Mechanized Verification of Circuit Descriptions using the Larch Prover

Jørgen Staunstrup,[1] Stephen J. Garland,[2] and John V. Guttag[3]

[1]Technical University of Denmark, Department of Computer Science, Building 344, DK-2800 Lyngby, Denmark. E-mail: jst@id.dth.dk

[2]The Hebrew University of Jerusalem and MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA. E-mail: garland@lcs.mit.edu

[3]MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA. E-mail: guttag@lcs.mit.edu

Abstract

This tutorial describes a circuit design technique that features using a mechanical theorem prover as a design tool, both to locate errors in prelimary designs and to verify certain aspects of completed designs. The design technique is based on SYNCHRONIZED TRANSITIONS, a high-level notation for describing circuits as collections of simple concurrent processes. It uses several different compilers to translate high-level circuit designs into formats suitable for circuit synthesis, simulation, and verification.

Keyword Codes: F.3.1; B.7.1; D.1.3
Keywords: Specifying and Verifying and Reasoning about Programs; Types and Design Styles; Concurrent Programming

1 Introduction

A variety of computer based tools help designers master the complexity of large integrated circuits. Many of these tools handle designs at a low level, for example, at the layout level. We see a potential for tools that assist the designer at higher levels of abstraction. In this tutorial we describe a design technique that features using a mechanical theorem prover as a design tool, both to locate errors in preliminary designs and to verify certain aspects of completed designs.

Our design technique is based on SYNCHRONIZED TRANSITIONS, a high-level notation for describing circuits as collections of simple concurrent processes. We use several different compilers to translate high-level circuit designs into formats suitable for circuit synthesis, simulation, and verification. By using *exactly the same circuit description* for

all translators, our approach eliminates a source of subtle errors that occur when transforming descriptions, for example, from a form suitable for synthesis into a form suited for verification.

In order to facilitate using a theorem prover to reason about the properties of circuit designs, we have

- developed the SYNCHRONIZED TRANSITIONS notation for expressing functional properties of circuits in addition to notations for describing their structure,
- developed an accompanying proof methodology, which involves extracting axioms and proof obligations from circuit designs, and
- built a compiler to extract these axioms and proof obligations in a format suitable for use with a general-purpose theorem prover.

What is most important in our approach is why and how we intend to use a theorem prover, not what theorem prover we use. Our aim is to use formal verification actively in the design process, not just to establish that a supposedly-debugged circuit is indeed correct. Because formal verification can be tedious and error prone, we use a theorem prover to assist with details and to guard against oversights. Because we do not want to place *a priori* limits on the properties of the circuits that we can reason about, we use a general-purpose theorem prover rather than one tailored to verifying particular properties. Because initial designs may not be correct, we use a theorem prover (the Larch Prover [?, ?]) that is designed to help users debug axioms and conjectures, not just help them prove theorems. And finally, to facilitate using LP, we have the SYNCHRONIZED TRANSITIONS compiler choose proof tactics particularly suited for our specific purpose.

Of the many properties that one may wish to establish for a circuit design, our design technique at present concentrates on two:

- verifying functional properties that can be expressed as invariants, and
- verifying refinements of abstract designs into concrete realizations.

There are many other properties that designers may wish to verify: that circuits operate at the desired speeds, that their power consumption is acceptable, or that they can be sold profitably. Hence our design technique does not provide guarantees against all design errors. Nonetheless, it helps considerably in catching design errors that cause circuits to give the wrong results.

2 SYNCHRONIZED TRANSITIONS: **Fundamental Notions**

SYNCHRONIZED TRANSITIONS is a language for designing integrated circuits, which it models as concurrent computations. A typical SYNCHRONIZED TRANSITIONS design consists of a collection of simple, concurrently executing parts. It closely models how a circuit operates, with several distinct sub-circuits cooperating to perform a joint computation.

The following example illustrates the fundamental concepts. Consider a very simple traffic light controller. Traffic lights are placed at the intersection of two roads, NS and

EW. A light called NSLight is visible on NS, another called EWLight is visible on EW. Each light can take one of three values: Green, Yellow, and Red. There are two sensors, called CarOnNS and CarOnEW, that indicate whether a car is waiting; CarOnEW is true if a car is waiting on EW.

To formalize the controller, we let `NSLight, EWLight, CarOnNS`, and `CarOnEW` be *state variables*, which may change value during a computation. A *transition* such as

$\ll$ `EWLight = Green AND CarOnNS` $\rightarrow$ `EWLight := Yellow` $\gg$

allows EWLight to change when the precondition `EWLight = Green AND CarOnNS` is true, that is, when the light is green and there is a car waiting in the opposite direction.

In general, a transition has the form $\ll$ *precondition* $\rightarrow$ *action* $\gg$, where *precondition* is a boolean expression and *action* is a multiple assignment that specifies the state transformation made by the transition. The action is executed only when the precondition is true. A multiple assignment

$\ll$ v_1, v_2, ..., v_k := e_1, e_2, ..., e_k $\gg$

specifies the simultaneous update of one or more state variables; all expressions on the right side are evaluated and then assigned to the variables on the left side, which must be distinct. When the precondition is the constant `true`, it can be omitted.

The following four transitions completely describe the operation of the traffic light controller.

```
≪ EWLight = Green AND CarOnNS → EWLight := Yellow ≫ ||
≪ EWLight = Yellow → EWLight, NSLight := Red, Green ≫ ||
≪ NSLight = Green AND CarOnEW → NSLight := Yellow ≫ ||
≪ NSLight = Yellow → NSLight, EWLight := Red, Green ≫
```

A transition describes a computation that is executed

- *concurrently*,
- *repeatedly*, being ready for another execution immediately upon completion,
- *atomically*, as indivisible operations, and
- *independently* of the order they appear in the circuit description.

There are strong similarities between SYNCHRONIZED TRANSITIONS and UNITY, as developed by Chandy and Misra [?]. Both describe a computation as a collection of atomic conditional assignments without any explicit flow of control. Chandy and Misra propose this as a general programming paradigm. SYNCHRONIZED TRANSITIONS is specialized to the development of application-specific integrated circuits.

It is important that the traffic light never allows cars to pass in both directions at the same time. This can be ensured if the light is always red in one of the two directions, that is, if it can be ensured that the assertion `EWLight = Red OR NSLight = Red` is always true. This condition is an example of an *invariant*. In general, an invariant is a predicate (on the values of the state variables) that is maintained by all transitions and that holds

for the initial values of the state variables. By induction, it follows that an invariant will hold throughout the computation. To verify that a given predicate I is an invariant, we must therefore show that:

1. I holds in the initial state, and
2. for each transition T, if I holds before executing T, then I also holds afterwards.

The second condition is established by assuming that the invariant holds for the values of the state variables prior to executing the transition (i.e., it holds in the *pre state*) and proving that the invariant holds for the values of the state variables afterwards (i.e., it holds in the *post state*). Consider, for example, the transition

```
≪ EWLight = Green AND CarOnNS → EWLight := Yellow ≫
```

Assuming the invariant holds in the pre state, and noting that the precondition ensures `EWLight = Green`, we conclude that `NSLight` must be red in the pre state. Since the transition does not change `NSLight`, the invariant will also hold in the post state.

A complete verification of a circuit requires proving a large number of such properties, and typical circuits have many more than the four transitions of the traffic light. For such applications, manual proofs, such as the one above, become very tedious and error prone. Furthermore, proofs must be redone every time a circuit description is changed. Usually, there is not much need for insight or creativity in these proofs; they are just time consuming. Hence a mechanical theorem prover is particularly suited for the task. In the next section we describe how to do this using the Larch Prover (LP), which has been integrated into the design tools supporting SYNCHRONIZED TRANSITIONS.

Following is a complete SYNCHRONIZED TRANSITIONS description of the traffic light.

```
CELL trafficlight
  STATIC Green = 0 Yellow = 1 Red = 2
  STATE
    EWLight, NSLight: INTEGER;
    CarOnEW, CarOnNS: BOOLEAN;
  INVARIANT EWLight = Red OR NSLight = Red
  INITIALLY EWLight, NSLight = Red, Green
BEGIN
  ≪ EWLight = Green AND CarOnNS → EWLight := Yellow ≫ ||
  ≪ EWLight = Yellow → EWLight, NSLight := Red, Green ≫ ||
  ≪ NSLight = Green AND CarOnEW → NSLight := Yellow ≫ ||
  ≪ NSLight = Yellow → NSLight, EWLight := Red, Green ≫
END trafficlight
```

The description is encapsulated in a *cell*, further details of which are given in Section ??. *Statics* are named quantities with fixed values. State variables must be declared and given specific types, which determine the operations that can be performed. They can also be given initial values. A list of transitions describes the computation performed by the cell. A cell is a textual description of a sub-circuit. A complete circuit consists of one or more cell instances. Several instances can be derived from the textual description of one cell. There is a similar distinction between the textual description of a transition and the instances of that transition.

3 Synopsis of LP, the Larch Prover

LP is a theorem prover for a subset of multisorted first-order logic. It is designed to work efficiently on large problems and to be used by relatively naive users. Among its uses are debugging formal specifications written in Larch [?], reasoning about algorithms involving concurrency [?], and establishing the correctness of hardware designs [?].

LP's applications motivate several departures in features and design from other theorem provers. First and foremost, LP is intended primarily for use as an interactive proof assistant or proof debugger, not as a fully automatic theorem prover. Its design is based on the assumption that initial attempts to state conjectures and prove them usually fail. As a result, LP is designed to carry out routine (and possibly lengthy) steps in a proof automatically and to provide useful information about why proofs fail. Unlike the Boyer-Moore prover [?, ?], it does not use heuristics to formulate additional conjectures in a search for a proof that may not exist. Strategic decisions, such as trying induction, must appear as explicit LP commands (either entered by the user or generated by an application-specific front-end such as the SYNCHRONIZED TRANSITIONS compiler).

LP provides a simple logical system based on familiar notions from first-order logic. Its deductive apparatus consists of a set of standard proof tactics (e.g., proofs by rewriting, cases, contradiction, and induction) and simple mechanisms for controlling the application of these tactics. Unlike LCF [?] and Isabelle [?], it does not allow users to define their own proof tactics. Instead, it enables users to record proof attempts, which it annotates to indicate their progress. Users can edit and replay these proof attempts, for example, to transform them into proofs of similar conjectures. LP checks its annotations during replay to keep edited proofs from getting "out of sync," for example, when some step succeeds with less user guidance than expected or to require more guidance.

Simplicity makes LP easy to learn and use. It also enables LP to perform powerful forward inferences efficiently. Many proofs succeed with a minimum of user interaction. When proof attempts fail, LP usually provides quick feedback. When time-consuming proof attempts occur, they are generally a sign that something is wrong. Because LP is efficient, it has been used to verify circuits described by several hundred (large) equations.

Finally, LP is designed for ease of user interaction. This is required during design both to find proofs that are too complicated given the current state of theorem-proving technology and, more importantly, to correct flaws in designs so that they can be proven correct.

3.1 LP's logical system

LP allows users to axiomatize theories and prove conjectures in the subset of multisorted first-order logic consisting of universal-existential ($\forall\exists$) formulas. Operationally, the basis for proofs in LP is a logical system that contains a small set of built-in axioms concerning booleans and conditionals, together with user-supplied axioms and theorems of the following kinds.

- *Equations* of the form $t_1 == t_2$, where t_1 and t_2 are terms. Equations of the form $t == true$ can be written simply as t. LP treats as inconsistent the equation *false* and all equations of the form $x == t$ or $not(x = t)$, where x is a variable and t is

a term not containing x.[4] LP uses inconsistencies to establish subgoals in proofs of implications and in proofs by cases and contradiction.

- *Rewrite rules* of the form $t_1 \rightarrow t_2$, which have the same logical meaning as equations, but that behave differently operationally. LP converts equations into rewrite rules by various ordering procedures that attempt to guarantee the termination of the resulting set of rules. It uses rewrite rules to reduce conjectures and facts in its logical system to normal forms.

- *Operator theories*, which assert that certain operators are commutative or both associative and commutative. Logically, operator theories are sets of equations. Operationally, LP uses them in equational term-rewriting to avoid nonterminating rewrite rules such as $x + y \rightarrow y + x$.

- *Deduction rules*, which LP uses to infer new equations from existing equations and rewrite rules. For example, the built-in deduction rule

  when $not(x) == true$ **yield** $x == false$

  enables LP to replace the rewrite rule `not(Red = Green) -> true` by the simpler rule `Red = Green -> false`. More importantly, deduction rules such as

  when (**forall** x) $I(x, pre)$, `Action`(y) **yield** $I(y, post)$

  are equivalent to universal-existential formulas such as

  $\forall y[\, \forall x[I(x, pre)$ & `Action`$(y)] \Rightarrow I(y, post)\,]$

- *Induction schemes*, such as *Nat* **generated by** 0, *plus1*, which LP uses to generate subgoals to be proved for the basis and induction steps in proofs by induction.

LP provides both forward and backward rules of inference, some of which it applies automatically, and others of which must be invoked explicitly by the user. Among the inference rules in LP are the following:

- *Reduction to normal form:* a forward rule, applied automatically to LP's logical system whenever a new rewrite rule or operator theory is asserted or deduced. Also a backward rule, applied automatically to reduce conjectures either to identities (in which case they are proved) or to subgoals that must be established by other rules of inference.

- *Deduction:* a forward rule, applied automatically to equations and rewrite rules.

- *Induction:* a backward rule invoked by users, for which LP generates subgoals (i.e., lemmas to be proved, usually with the aid of additional hypotheses) from the induction schemes.

[4]Thus, LP is designed for reasoning about models in which every sort has at least two elements. In practice, axioms that constrain a sort to be empty or to have a single element are almost always either intentionally inconsistent or result from mistaken formalizations. Hence LP chooses to treat them as inconsistent to protect users from mistakes, and to make some proofs go faster, rather than to provide a more general logical framework.

- *Proofs by cases and contradiction:* backward rules invoked by users, for which LP generates subgoals.
- *Instantiation:* a forward rule invoked by users, applicable to rewrite rules, equations, and deduction rules.
- *Proofs of deduction and induction rules:* backward rules, applied automatically.
- *Critical pairs:* a forward rule invoked by users, which produces equational consequences (such as $y/1 == y$) from pairs of rewrite rules (such as $1 * x \rightarrow x$ and $x * (y/x) \rightarrow y$) that do not follow by rewriting (because $y/1 == y$ is already fully reduced). The Knuth-Bendix completion procedure [?, ?] is the closure of the critical pair computation.

Details concerning LP can be found in [?].

3.2 Proving invariance using LP

The goal of an invariance proof is to show that some predicate involving state variables is maintained by all transition instances. When using LP to carry out such a proof, we have considerable latitude in how to formalize invariants and transitions. In this section, we illustrate one way that LP can be used to verify that the traffic light controller preserves its invariant.

First, we formalize all non-boolean values that state variables can take by introducing appropriate sorts. For example, given the declarations

```
declare sort Color
declare operators Green, Yellow, Red: -> Color
```

that introduce constants (i.e., 0-ary functions) for the colors, we axiomatize the properties of colors using the statements

```
set name color
assert not(Green = Red), not(Yellow = Red), not(Green = Yellow)
```

to introduce named equations that axiomatize properties of these constants. We also formalize state variables as constants of appropriate sorts:

```
declare sorts ColorVar, BoolVar
declare operators EWLight, NSLight: -> ColorVar
declare operators CarOnEW, CarOnNS: -> BoolVar
```

To distinguish the values of state variables in the pre and post states, we introduce additional operators

```
declare sort State
declare operators
  pre, post:                -> State
  .:         ColorVar, State -> Color
  .:         BoolVar, State  -> Bool
```

that enable us to write `EWLight.pre` and `EWLight.post` for the values of `EWLight` in the two states.[5]

Now we are ready to formalize the invariant and the transitions of the traffic light controller. We formalize the invariant as a unary predicate, defined by a single equation:

```
set name Invariant
declare operator I: State -> Bool
declare variable s: State
assert I(s) == EWLight.s = Red | NSLight.s = Red
```

The operator | is boolean OR. We formalize the transitions using deduction rules:

```
set name Actions
declare operators Action, Action1, Action2, Action3, Action4: -> Bool
assert
  when Action yield
    Action1 | Action2 | Action3 | Action4
  when Action1 yield
    EWLight.pre = Green                    % Preconditions
    CarOnNS.pre
    EWLight.post = Yellow                  % Action
    U(NSLight), U(CarOnNS), U(CarOnEW)     % Unchanged
  when Action2 yield
    ⋮
```

The first deduction rule asserts that there are four transition instances, any one of which can be selected (nondeterministically) for execution. The second causes LP to deduce six facts about the transition whenever it discovers that `Action1` is true. The first two conclusions assert that the precondition `EWLight.pre = Green AND CarOnNS.pre` is true in the pre state, the third that `EWLight` has a new value in the post state, and the last three that the values of the other state variables do not change. The properties of the (overloaded) operator `U` (for Unchanged) are axiomatized by the declarations and equations[6]

```
declare operators U: ColorVar -> Bool, U: BoolVar -> Bool
declare variables c: ColorVar, b: boolVar
assert
  U(c) == c.post = c.pre
  U(b) == b.post = b.pre
  ..
```

To verify that the invariant is preserved, we introduce a conjecture and direct LP to prove it by assuming the hypothesis of the implication and then doing a case analysis:

[5] LP uses context to disambiguate overloaded operators (e.g., .).

[6] LP uses the symbol .. to denote the end of multiline commands.

```
prove (I(pre) & Action) => I(post)
   resume by =>
   resume by cases Action1, Action2, Action3, Action4
   qed
```

LP carries out the details of the proof without any further user interaction. In response to `resume by =>`, LP tries to prove the subgoal `I(post)` using the additional axiom `I(pre) & Action`, from which it deduces `I(pre)` and `Action1 | ... | Action4`. In response to the`resume by cases`, LP first verifies that the cases are mutually exhaustive; then it proceeds to prove the subgoal `I(post)` from each of the additional axioms `Action1`, ..., `Action4`. The first case is typical. LP applies the deduction rule defining `Action1` to derive six new equations, which it orients into rewrite rules. The first of these, together with the rewrite rule `Green = Red -> false` deduced from axiom `color.1`, reduce `I(pre)` to `NSLight.pre = Red`; LP transforms this equation and `U(NSLight)` into rewrite rules that reduce the subgoal `I(post)` to an identity.

Proofs such as this illustrate the power of LP's rewriting strategy. By reducing axioms as well as conjectures to normal forms, LP is often able to deduce enough additional rewrite rules to reduce a conjecture to an identity.

3.3 Automatic translation

To assist in formalizing proofs such as the one just given, we have developed a translator from SYNCHRONIZED TRANSITIONS to LP that extracts appropriate declarations for state variables, axioms defining transitions and invariants, and proof obligations from circuit descriptions. The translator takes care of routine details, such as generating clauses for state variables that are unchanged by a transition.

The translator also selects idioms that we have found to be the most practical, either because they make proofs easier to understand and control, or because they enable proofs to succeed more quickly and with less user interaction. Some idioms involve the details of formalization; others involve proof tactics.

When formalizing actions, the translator uses an idiom based on deduction rules rather than equations. Although the equation $actionA \Rightarrow (t_1 \ \& \ t_2 \ \& \ t_3) == true$ and the deduction rule **when** *actionA* **yield** t_1, t_2, t_3 have the same logical meaning, they have different effects operationally. The equation, when converted into a rewrite rule, is applicable in more settings: to equations such as $action == actionA \mid actionB$, and not just to the equation $actionA == true$. This greater applicability, however, is a hindrance in SYNCHRONIZED TRANSITIONS proofs, because it expands equations prematurely, making them less readable and making subsequent reductions more costly (because the cost of normalizing a conjunction $t_1 \ \& \ t_2 \ \& \ t_3$ exceeds the cost of separately normalizing each of its conjuncts).

When formalizing conjectures, the translator can often generate fruitful idioms for the proof. For example, when formalizing invariance proofs, the translator generates an LP command to initiate a proof by cases on the transition selected for execution. As a result of these idioms, LP often verifies invariants of SYNCHRONIZED TRANSITIONS programs directly from the output of the translator without any manual assistance.

We have used this technique to verify several circuits, for example, an arbiter, an oscillator, and an efficient adder. To describe these and other nontrivial circuits, it is necessary to use more of the constructs available in SYNCHRONIZED TRANSITIONS, some of which are introduced in the next section.

4 SYNCHRONIZED TRANSITIONS: Structuring Concepts

To make the design of large circuits tractable, SYNCHRONIZED TRANSITIONS provides mechanisms for decomposing designs into independent parts that can be considered separately. These mechanisms were chosen to support both efficient circuit realization and formal verification.

The following example illustrates some of the structuring mechanisms. Let `s` be a string of characters, that is, a state variable of type `ARRAY[1..n] OF CHAR`. Initially, the first m characters of `s`, where $0 \leq m \leq n$, are in {'a'..'z'}, and the last $n - m$ characters are the special end marker `EOS`. We describe a circuit that removes adjacent duplicate characters from `s`, appending `EOS` characters at the end as necessary. For example, "bbannnaanaa" eventually becomes "banana".

Consider three adjacent elements, `a`, `b`, and `c`, of `s`. If `a` and `b` contain the same character, the one in `b` can be removed; otherwise `b` should be left unchanged. One way to do this is with the following named transition:

```
TRANSITION copy(a, b, c:  CHAR)
  ≪ a = b → b := c ≫
```

This transition needs to be applied to every position in the string, which we do by using the following SYNCHRONIZED TRANSITIONS notation to create $n - 2$ distinct instances of the transition, each responsible for removing possible duplicates in one position of the string, and all executing concurrently:

```
{|| j: [1..n-2] | copy(s[j], s[j+1], s[j+2]) }
```

Another transition instance

```
≪ s[n-1] = s[n] → s[n] := EOS ≫
```

completes the circuit by placing an end marker `EOS` at the end of the string when needed.

4.1 Cells

A *cell* encapsulates part of a design. For example, the following cell encapsulates the circuit for removing duplicate elements from a string.

```
TYPE
  chars = ARRAY range OF CHAR
  range = [1..n]
CELL nodup(STATIC n: INTEGER; STATIC EOS: CHAR; s: chars)
  TRANSITION copy(a, b, c: CHAR)
    ≪ a = b → b := c ≫
BEGIN
  {|| j: [1..n-2] | copy(s[j], s[j+1], s[j+2]) } ||
  ≪ s[n-1] = s[n] → s[n] := EOS ≫
END nodup;
```

The cell has three parameters. The first, `n`, determines the size of the array containing the string. This parameter is static, which means that its value is fixed for each instance of the cell `nodup`. It is possible, however, to create different instances of the cell that operate on arrays of different sizes. `EOS` is also a static parameter. The third parameter, `s`, is a state parameter; inside the cell, it is treated like any other state variable. By passing state variables as actual parameters, several cell instances can communicate and cooperate. From the viewpoint of the circuit, state parameters are connections to other cell instances.

The internal structure of a cell, which may contain local state variables, transitions, and subcells, is hidden from the rest of the design. The scope of state variables (including formal parameters) is defined to be the cell in which they are declared; the scope does not include inner subcells. Cell instances communicate via parameters to coordinate their computations. Instantiation of a cell is specified by giving its name and an actual parameter list, for example, `nodup(100, '\0', str)`. An actual parameter may be read and written by both the instantiating and the instantiated cell unless access to the parameter is restricted by one of the mechanisms described in Section ??.

A single state variable must not be known under two (or more) different names in a single cell. This is ensured by requiring that all actual parameters passed to a cell instance are distinct.

4.2 Hierarchies of cells

Cells may be nested to describe a hierarchy of subcircuits. In general, a circuit is a set of transition instances, which is obtained by instantiating all its cells and all elements in the bodies of these cells. This can lead through many levels of cell instantiations, until one reaches the innermost cells with only transitions to instantiate.

As an example of a hierarchical circuit, we consider an *arbiter* for providing indivisible access to some shared resource, for example, a bus or a peripheral. This particular arbiter is implemented as a tree in which all nodes (including the root and the leaves) are identical. The arbitration algorithm is based on passing a unique token around the tree. An external process, connected to a leaf, may use the resource only when that leaf has the token.

Each node has three pairs of connections, one for its parent and one for each of its children. A connection pair consists of two signals, `req` and `gr`, standing for "request" and "grant." Such a pair is used according to the following four-phase protocol.

1. A node raises a `req` signal to its parent to indicate that it wants the token.
2. The parent passes the token to a child by raising the child's `gr` signal.
3. A node indicates that it is done with the token by lowering its `req` signal.
4. Its parent takes back the token by lowering the child's `gr` signal, thereby enabling it to grant a new request.

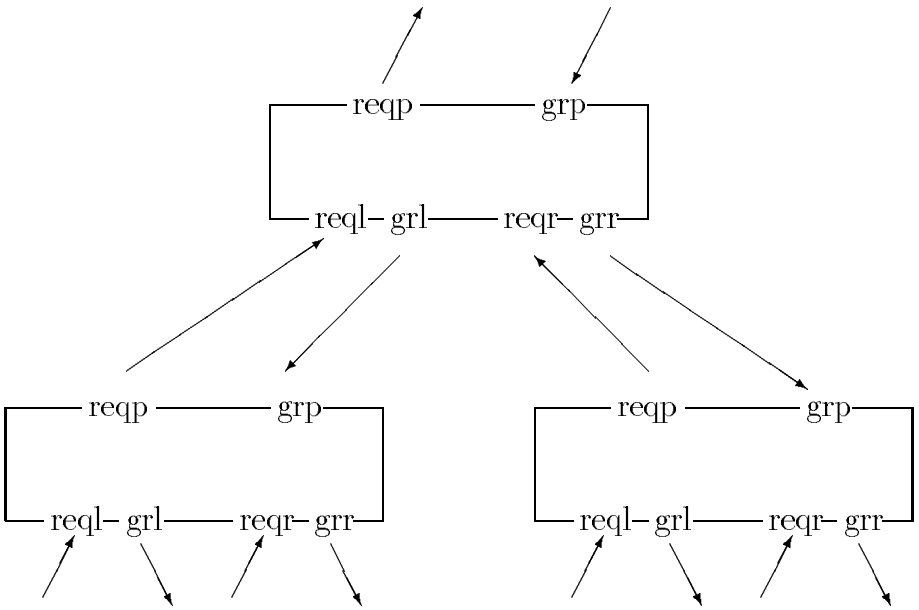

Figure 1: Two levels of arbiter tree

Figure ?? shows a few nodes and their interconnections. An external process requests the resource by setting `reqi` (where `i` is `l` or `r`, depending on the leaf to which the process has been assigned). When the corresponding `gri` is set (by the arbiter), the process may go ahead and use the resource. The signals `reqp` and `grp` are connected to `reqi` and `gri` at the next (higher) level of the tree. When both children of a particular node request the resource, it is given first to the left child; when that child releases the resource, it is given to the right child. The `reqp` and `grp` of the root node are connected. This means that the root is able to grant a request immediately.

The following cell describes the structure of the arbiter using SYNCHRONIZED TRANSITIONS, but without showing the processes attached to the leaves of the tree. The conditional instantiation { `level` < `top` | ...} is performed only when the value of the static parameter `level`, which increases as cells nest more deeply, is less than that of `top`. The invariant of the cell states that an arbiter node grants access to at most one child, and only when it itself has requested and been granted access.

```
CELL arbiter(grp, reqp: BOOLEAN; STATIC level, top: INTEGER)
  STATE grl, reql, grr, reqr: BOOLEAN
  INVARIANT
    ¬ (grl ∧ grr) ∧ ( (grl ∨ grr) => (grp ∧ reqp) )
  TRANSITION requestparent
    ≪ ¬ grp ∧ (reql ∨ reqr) → reqp := TRUE ≫
  TRANSITION grantleft
    ≪ grp ∧ reqp ∧ reql ∧ ¬ grr → grl := TRUE ≫
  TRANSITION grantright
    ≪ grp ∧ reqp ∧ reqr ∧ ¬ grl ∧ ¬ reql → grr := TRUE≫
  TRANSITION doneleft
    ≪ ¬ reql → grl := FALSE ≫
  TRANSITION doneright
    ≪ ¬ reqr → grr := FALSE ≫
  TRANSITION done
    ≪ grp ∧ ¬ grl ∧ ¬ grr → reqp := FALSE ≫
BEGIN
  requestparent || grantleft || grantright || doneleft || doneright || done ||
  { level < top | arbiter(grl, reql, level+1, top) ||
                  arbiter(grr, reqr, level+1, top) }
END arbiter
```

4.3 Restricting the use of state variables

Ordinarily, a state variable passed as an actual parameter may be read and written both by the instantiating and by the instantiated cell. We can supplement the scope rules in SYNCHRONIZED TRANSITIONS, which hide state variables from inner cells unless passed as actual parameters, by stating one of three additional restrictions on state variables (including formal parameters).

- External write: the variable may be changed only externally, that is, in a surrounding cell instance. This restriction is only meaningful for formal parameters.

- Local write: the variable may be changed in this scope only; it may not be changed in either internal or external cell instances.

- Internal write: the variable may be changed in internal cell instances only, that is, by instances to which it is passed as an actual parameter.

These restrictions facilitate formal verification and optimization of circuits synthesized from designs in SYNCHRONIZED TRANSITIONS. They are written among the declarations for a cell. For example, the arbiter cell might contain the following restrictions.

```
RESTRICTION
  grp: EXTERNAL;
  reqp, grl, grr: LOCAL;
  reql, reqr: INTERNAL;
```

When a variable passed as an actual parameter is restricted to be externally or locally written, the corresponding formal parameter (in the cell instance to which the parameter is passed) must be declared to be external.

4.4 Protocols

Restrictions on formal parameters provide a partial protocol for using a cell, telling which parameters can be written by the cell, which by the instantiating cell, and which (by omission) by both. SYNCHRONIZED TRANSITIONS allows designers to express more complete protocols, which describe the transitions performed by the cell solely in terms of its formal parameters, hiding the internal structure and operation of the cell. These protocols play a crucial role in reasoning about circuits, as they express the intent of the design rather than a particular implementation of that intent.

In the case of the arbiter, SYNCHRONIZED TRANSITIONS enables us to formalize the intended four-phase protocol by saying that `reqp` can change only when it is the same as `grp`, and `grp` can change only to the value of `reqp`.

```
PROTOCOL
  (reqp.post = ¬reqp.pre ⇒ reqp.pre = grp.pre) ∧
  (grp.post = ¬grp.pre ⇒ grp.post = reqp.pre)
```

Protocols differ from invariants in that they can refer to both the pre and the post state, and also in that they cannot refer to local state variables.

5 Hierarchical Verification

Given a description of a circuit as a hierarchical composition of cells, we would like to reason about its behavior in an analogous hierarchical fashion. Associated with each cell instance x is an invariant $I(x, s)$, descriptions $T_i(x)$ of the cell's transitions, and a protocol $P(x)$. The difference between T_i and P is that T_i describes a single transition and can mention local state variables, whereas P summarizes all transitions and mentions only the formal parameters of the cell. What we would like to show is that any transition performed by any cell obeys the protocol of that cell and preserves the invariants of all cells.

A global proof strategy begins by assuming $\forall x\, I(x, pre)$ (i.e., that the invariant holds in all cell instances) and $\exists n\, T_i(n)$ (i.e., that transition i occurs in some instance). It then attempts to show $\forall x\, I(x, post)$, that is, that the invariant still holds in all cell instances after the transition. Ideally, however, we would like to reason locally rather than globally.

A local, but inadequate, proof strategy would involve showing

$$\forall x[I(x, pre) \wedge T_i(x) \Rightarrow I(x, post)]$$

for all i, that is, if the invariant holds at a cell instance where a transition occurs, then it still holds at that cell instance after the transition. The key difference between the local and global plan of attack is that (locally) we assume less and are required to prove less.

This strategy is inadequate when an invariant involves parameters, because it shows only that invariants expressed entirely in terms of locally written variables are preserved. To show that invariants involving formal or actual parameters are preserved, we must establish that these invariants are preserved by the transitions of both the instantiating cell (where the parameters are actual) and the instantiated cell (where the parameters are formal). This step, which is the key to localized invariance proofs, establishes the *noninterference* of neighboring cells.

Consider a cell instance x in the arbiter. It passes state variables as actual parameters to two subcells, $Part1(x)$ (its left child) and $Part2(x)$ (its right child), and it receives state variables as formal parameters from a supercell, $Parent(x)$. We must show that the transitions of these other cells preserve the invariant $I(x, s)$. For example, we must show that

$$\forall x[\, I(x, pre) \wedge I(Part1(x), pre) \wedge T_i(Part1(x)) \Rightarrow I(x, post)\,]$$

for each transition T_i of the arbiter. Informally, this proof obligation has us assume the invariants $I(x, pre)$ and $I(Part1(x), pre)$ of both cell instance x and its left child. If we can show that execution of each T_i in the left child preserves the invariant of the cell, then we have shown that transitions of the left cell do not interfere with x. Similar steps are required for $Part2(x)$ and $Parent(x)$.

In the case of the arbiter, it suffices to consider transitions in three neighboring cells (the parent and the children). Transitions in other cells cannot interfere with the invariant. In general, it is not true that only neighboring cells can interfere with the invariant. For example, parameters may be passed through several levels of cell instantiations.

Unfortunately, the number of proof obligations grows very rapidly with this approach: if there are n cell instances, and each instance has t transitions, each cell gives rise to nt proof obligations. Accordingly, we prefer to reason about the protocol $P(x)$, which captures the essential properties of a cell, rather than about the many transitions of the cell. We use this protocol when proving noninterference, thereby reducing the number of proof obligations from nt to n. Furthermore, by showing that the protocol of any cell is obeyed by all transitions of neighboring cells that obey their protocols, we need consider only neighboring cells when proving noninterference. The average number of neighboring cells is generally independent of the overall number of cell instances, so that the number proof obligations is reduced to a constant number (for each type of cell).

This analysis leaves us with the following proof obligations for showing that invariants are preserved and that protocols are obeyed.

First, each transition T_i of a cell instance x must preserve the invariant $I(x, s)$ of the cell and obey the protocol $P(x)$ of the cell and the protocols $P(Part_i(x))$ of its subcells. In short, we must show

$$I(i, pre) \wedge inCell(x) \wedge T_i(x) \Rightarrow PostObl(x)$$

for each transition T_i of the cell, where $inCell$ expresses the scope rules and restrictions for the cell and $PostObl(x)$ is defined to be

$$I(x, post) \wedge P(x) \wedge P(Part_1(x)) \wedge \ldots \wedge P(Part_n(x))$$

where n is the number of subcells.

Second, cells respect the protocols and invariants of their subcells and *vice versa*. Here we must show (1) that transitions of a cell made in accordance with its protocol respect each subcell's protocol and invariants, and (2) that transitions of subcells made in accordance with their protocols do the same for the cell. In short, we must show

$$1_i \quad \forall x\,[\,I(Part_i(x), pre) \wedge inCell(x) \wedge P(x) \Rightarrow PostObl(Part_i(x))\,]$$
$$2_i \quad \forall x\,[\,I(x, pre) \wedge inCell(Part_i(x)) \wedge P(Part_i(x)) \Rightarrow PostObl(x)\,]$$

As we have noted, this method of proof has several advantages. It specifies protocols that cell instances observe with respect to each of their neighbors. It reduces the number of noninterference conditions to verify for each cell instantiation (to two). And it enables us to show that changes made in accordance with the protocol for one neighboring cell instance are also in accordance with the protocols for other neighbors. Therefore, it circumvents the need to consider separately interferences that might result when parameters are passed through several levels, for example, when a grandchild of a cell instance can change an actual parameter supplied by the grandparent. A complete proof based on this approach [**?**] requires less user interaction to show that all cells of the hierarchical arbiter maintain the invariant

$$\neg(grl \wedge grr) \wedge ((grl \vee grr) \Rightarrow (grp \wedge reqp))$$

6 Mutual Exclusion in the Arbiter Tree

We also used LP to show that the entire arbiter tree, no matter what its size, ensures mutual exclusion, i.e., that two leaf nodes never have their grant signals true simultaneously. The proof proceeds by induction over the structure of the arbiter tree, i.e., by induction over the level of nodes in the tree. For this proof, we start with a theory that contains axioms for natural numbers and trees together with the invariant for the arbiter tree (proved in the last section). Included in this theory are the following equations that describe a tree with root node `Root` and infinitely many levels.

```
declare sort Instance
declare variables x, y, z: Instance
declare operators
  Root  :           -> Instance
  Part1 : Instance  -> Instance
  Part2 : Instance  -> Instance
  level : Instance  -> Nat
  ..
assert
  level(Root) == 0
  level(Part1(x)) == plus1(level(x))
  level(Part2(x)) == plus1(level(x))
  ..
```

We also assert that all nodes in the tree (equivalently, all subtrees of the tree) are generated by the operators `Root`, `Part1`, and `Part2`.

```
assert Instance generated by Root, Part1, Part2
```

This assertion provides LP with an inductive rule of inference. Now we issue the commands

```
prove ((level(y) = level(z)) & grp(y) & grp(z)) => (y = z)
  resume by induction on y
```

to begin an inductive proof of the mutual exclusion property. The proof requires some user assistance. LP first attempts to prove the basis case

```
((level(Root) = level(z)) & grp(Root) & grp(z)) => (Root = z)
```

for the induction. This requires a subsidiary induction on `z`, which we initiate by typing

```
resume by induction on z
```

From this, LP finishes the basis case without further help. It then generates the induction hypothesis

```
((level(yc) = level(z)) & grp(yc) & gr(z)) => (yc = z)
```

(which asserts that mutual exclusion occurs at the level of the node `yc`) and tries to prove the equations

```
((level(Part1(yc)) = level(z)) & grp(Part1(yc)) & grp(z)) => (Part1(yc) = z)
((level(Part2(yc)) = level(z)) & grp(Part2(yc)) & grp(z)) => (Part2(yc) = z)
```

(which assert that mutual exclusion occurs at the next level). Again, we must instruct LP to induct over `z`. The basis case for this induction goes through by reduction to normal form. The two induction steps for `z` then combine with the two for `y` to yield four lemmas that must proved:

```
((level(Part1(yc)) = level(Part1(zc))) & grp(Part1(yc)) & grp(Part1(zc)))
                 => (Part1(yc) = Part1(zc))
((level(Part1(yc)) = level(Part2(zc))) & grp(Part1(yc)) & grp(Part2(zc)))
                 => (Part1(yc) = Part2(zc))
((level(Part2(yc)) = level(Part1(zc))) & grp(Part2(yc)) & grp(Part1(zc)))
                 => (Part2(yc) = Part1(zc))
((level(Part2(yc)) = level(Part2(zc))) & grp(Part2(yc)) & grp(Part2(zc)))
                 => (Part2(yc) = Part2(zc))
```

To prove the first lemma, we partition the proof into cases with the commands

```
resume by case yc = zc
  resume by =>
```

which serve to reduce the proof to one nontrivial case. Finally, we complete the proof by entering the command

```
critical-pairs *Hyp with *
```

to compute critical pairs between rewrite rules that describe the nontrivial case and rewrite rules that describe the induction hypothesis and the invariant in the pre state. LP orders these critical pairs automatically into rewrite rules, which help reduce the lemma to an identity. The proof of the remaining three lemmas is identical in form.

7 Abstraction Functions

In addition to proving invariants, we also use LP to show *refinement*, i.e., that one design in SYNCHRONIZED TRANSITIONS, called a *concrete* circuit, implements another design, called an *abstract* circuit. To do this, we define an *abstraction function*, $\mathcal{A}$, that maps states of the concrete circuit to states of the abstract circuit, and we impose a proof obligation to ensure that a concrete circuit does only what the abstract circuit prescribes. We are concerned here only with safety properties, and we do not require that the concrete circuit does everything that is possible in the abstract one.

Formally, an abstraction function is a mapping $\mathcal{A} : S_c \mapsto S_a$ from the concrete state space S_c to the abstract state space S_a. S_c is the set of all possible concrete states, that is, all assignments of values to the concrete state variables. Similarly, S_a is the set of all possible abstract states.

Loosely speaking, we require that for every execution of a concrete transition instance T_C, there must be an abstract transition instance T_A whose execution would have the equivalent effect. This requirement captures the central idea of relating the behavior of a concrete implementation to its abstract specification: for each action in the concrete circuit, there is a corresponding action (an "explanation") in the abstract circuit.

Usually, the concrete circuit contains more details than the abstract. Therefore, the abstraction function maps many concrete states to the same abstract state. Such redundancy may be necessary for efficient or robust implementations. Often, the concrete circuit changes from one concrete state to another that represents the same abstract state, that is, $\mathcal{A}(pre_C) = \mathcal{A}(post_C)$. To allow for such invisible changes, which are not reflected in the abstract circuit, we must weaken the requirement we impose on the abstraction function. Instead of requiring that there is an abstract transition instance corresponding to *every* concrete one, we only require this for every transition instance making a visible state change.

These considerations are captured by the *abstraction condition*

$$\Big(\exists T_C : \{pre_C\}\ T_C\ \{post_C\}\Big) \Rightarrow$$
$$(\exists T_A : \{\mathcal{A}(pre_C)\}\ T_A\ \{\mathcal{A}(post_C)\}) \vee \mathcal{A}(pre_C) = \mathcal{A}(post_C)$$

Example: A Modulo-4 counter

Consider a simple modulo-4 counter, which is described abstractly as follows: d

```
CELL counter(incrA: BOOLEAN)
  STATE cA: INTEGER
BEGIN
  ≪ incrA → incrA, cA:= FALSE, (cA+1) MOD 4 ≫
END
```

The signal `incrA` indicates that an increment should be made. When it has been done, `incrA` becomes false again.

Such a counter can be realized with two bits `(lC, mC)` representing `cA`. `lC` is the least significant bit and `mC` the most significant in the binary representation of `cA`.

```
CELL counter(incrC: BOOLEAN)
  STATE lC, mC: BOOLEAN
BEGIN
  ≪ incrC ∧ ¬lC → incrC, lC := FALSE, TRUE ≫ ||
  ≪ incrC ∧ lC ∧ mC → incrC, lC, mC := FALSE, FALSE, FALSE ≫ ||
  ≪ incrC ∧ lC ∧ ¬mC → incrC, lC, mC := FALSE, FALSE, TRUE ≫
END
```

To use LP, we formalize the sets of concrete and abstract states as sorts, with a function `A` mapping one to the other. We also introduce auxiliary functions to evaluate state variables and to construct states out of their components.

```
declare sorts StateC, StateA
declare operators
  A: StateC -> StateA
  .: BoolVar, StateA -> Bool
  .: IntegerVar, StateA -> Integer
  .: BoolVar, StateC -> Bool
  I: Bool -> Integer
  makeA: Integer, Bool -> StateA
  ..
```

We define the properties of these functions with the following equations:

```
cA.makeA(i, b) == i
incrA.makeA(i, b) == b
makeA(cA.sA, incrA.sA) == sA
makeA(i1, b1) = makeA(i2, b2) == i1 = i2 & b1 = b2

I(true) == 1  I(false) == 0
A(sC) == makeA( (2*I(mC.sC)) + I(lC.sC), incrC.sC )
```

The abstract transition is defined by the equation

```
trA(preA, postA) ==
  incrA.preA & incrA.postA = false  & cA.postA = next(cA.preA)
```

where the function `next` is defined elsewhere as modulo 4 increment. The three concrete transitions are defined by the equations

```
tr1C(preC, postC) ==
  incrC.preC & not(lC.preC) &
  incrC.postC = false & lC.postC = true &
  mC.postC = mC.preC
tr2C(preC, postC) ==
  incrC.preC & lC.preC & not(mC.preC) &
  incrC.postC = false & lC.postC = false  &
  mC.postC = true
tr3C(preC, postC) ==
  incrC.preC & lC.preC & mC.preC &
  incrC.postC = false & lC.postC = false  &
  mC.postC = false
```

Using these definitions, we verify the abstraction condition by proving an implication of the following form, for each of the three concrete transition instances:

```
prove tr1C(preC, postC) => (A(preC) = A(postC) | trA(A(preC), A(postC)))
```

The only user assistance needed for this proof is to divide it into two cases.

8 Beyond Safety Properties

In this tutorial, we have shown how invariants and refinements can be verified and how the verification can be supported by mechanical tools. These aspects of circuit design are both examples of safety properties; they prevent "bad things from happening," for example, entering states not satisfying the invariant. Although we have not given any examples in this paper, requirements that an answer be produced in a fixed number of steps can be formulated as safety properties.

Much current research is directed towards techniques for verifying liveness properties, that is, for proving that certain events *must eventually* occur. We have not presented any work on liveness proofs, though LP can, and has, been used by others to reason about liveness.

One may well ask, "what is the point of verifying some safety properties, if we cannot ensure that the final circuit will finish its computation?" Safety proofs ensure that if a circuit ever delivers an output, that output has the verified properties. This is particularly important because, if a circuit delivers the wrong output, it may well go unnoticed. In contrast, if a circuit delivers no output, it is likely to be noticed rather soon. That is not to say that we ignore liveness considerations. Rather, we tend to rely on informal techniques to convince ourselves that they hold.

9 Experience

We have used LP and Synchronized Transitions together to verify safety properties of several small, but interesting designs such as the arbiter, Chandy and Misra's efficient adder (Section 6.9,[?]), and a set of building blocks for asynchronous circuits. We have

also used these tools when teaching courses. Students have verified a series of toy examples such as the `nodup` cell for eliminating duplicates in a string.

Designs described with SYNCHRONIZED TRANSITIONS can be manipulated and analyzed in a number of ways:

1. *Verification of invariants:* Proving that transitions maintain properties formulated as invariant assertions.

2. *Refinement:* Proving that a concrete program, the realization, is a correct implementation of another program, the abstraction.

3. *Timing analysis:* Verifying timing aspects of the transitions.

4. *Simulation:* Informally verifying a design by executing it with various input data.

5. *Synthesis:* Transforming a design into a circuit description, e.g., a netlist or a layout.

6. *Verification of implementation conditions:* Checking restrictions required for the realization to operate correctly.

Separate experimental tools support (??–??), (??), (??), and (??). This tutorial has concentrated on (??) and (??), for which we use LP together with a compiler that translates a source program in SYNCHRONIZED TRANSITIONS into a set of axioms and proof obligations in a format appropriate for LP. Simulation of a program in SYNCHRONIZED TRANSITIONS can be done using a compiler which produces an executable C program, other compilers make it possible to automatically synthesize circuits.

Our initial experience with LP and SYNCHRONIZED TRANSITIONS is encouraging. We believe that it is currently possible to design moderate-sized circuits in a high-level language supported by both verification and synthesis tools. The principal advantage of using systems like LP to construct machine-based proofs is that such proofs are more reliable. They are harder to construct than manual proofs primarily where manual proofs are vague, and therefore suspect. An additional benefit of a machine-based proof is that it makes it easier to see exactly what is required to ensure the desired properties. This benefit has led us to simplify several circuits and the invariants used to reason about them. A final benefit of machine-based proofs is that they are easily repeatable. This allows us to perform regression tests when we change circuits or restate their invariants.

Current techniques and tools for formal verification are not sufficiently refined for widespread application, but the potential benefits are so significant that we think they must be investigated seriously. When formal verification works, it corresponds to an exhaustive simulation: a certain property is ensured for all combinations of inputs in all states. Since exhaustive simulation is rarely feasible in practice, formal verification has the potential of improving the quality of integrated circuit design.

10 Availability of Tools

LP is written in CLU and runs under Unix. Release 2.2 is currently available free of charge, and without a license, by anonymous ftp from larch.lcs.mit.edu. Prospective users can

retrieve an executable version of LP, along with a supporting run-time library containing sample axiom sets and proofs, for DECstations, Sparcstations, MIPS machines, Vaxes, or Sun-3 workstations. Source code is also available.

The tools supporting SYNCHRONIZED TRANSITIONS are still undergoing development. They are written in C and run under Unix. Currently, they are not available for public use, but individual access to use them can be arranged by contacting the first author (jst@id.dth.dk).

Ackowledgement

Research supported in part by the Danish Technical Research Council and in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-89-J-1988, and by the National Science Foundation under grant CCR-8910848.

The work reported in this tutorial has been influenced by many discussions with Jim Saxe, Jim Horning, Mark Greenstreet and Niels Mellergaard. We are grateful for their help.

References

[1] Robert S. Boyer and J Strother Moore. *A Computational Logic*. New York: Academic Press, 1979.

[2] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. New York: Academic Press, 1988.

[3] K. M. Chandy and J. Misra. *Parallel Program Design: a Foundation*. Addison-Wesley, 1988.

[4] Stephen J. Garland and John V. Guttag. Inductive methods for reasoning about abstract data types. In *Proceedings of the 15th ACM Conf. on Principles of Programming Languages*, 1988, 219–228.

[5] Stephen J. Garland, John V. Guttag, and James J. Horning. Debugging Larch Shared Language specifications. *IEEE Transactions on Software Engineering* 16(9):1044–1057, September 1990.

[6] Stephen J. Garland and John V. Guttag. *A Guide to LP, The Larch Prover*. Digital Equipment Corporation Systems Research Center Report 82, December 1991.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985.

[8] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297, Pergamon Press, 1969.

[9] Niels Mellergaard. Generating Proof Obligations for Circuits. Technical report, Department of Computer Science, Technical University of Denmark, August 1991.

[10] Larry C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge University Press, Cambridge, 1987.

[11] Larry C. Paulson. The foundation of a generic theorem prover. Technical Report No. 130, University of Cambridge Computer Laboratory, March 1988.

[12] G. L. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM* 28(2):233–264, April 1981.

[13] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. *Proceedings from the International Workshop on Designing Correct Circuits.* J. Staunstrup and R. Sharp (ed.) Elsevier 1992.

[14] Jørgen Staunstrup, Stephen J. Garland, and John V. Guttag. Localized verification of circuit descriptions. In *Proceedings of an International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, *Lecture Notes in Computer Science* 407, pages 349–364, Springer-Verlag, 1989.