Localized Verification of Circuit Descriptions

Jørgen Staunstrup *
Technical University of Denmark

Stephen J. Garland and John V. Guttag †
Massachusetts Institute of Technology

1 Introduction

A variety of computer based tools help designers master the complexity of large VLSI circuits. Many of these tools handle designs at a low level, e.g., at the layout or transistor level. We see a potential for tools that assist the designer at higher levels of abstraction. In this paper we describe some experiments in which we used an automatic theorem prover to verify properties of circuits.

Our approach is based on a high-level notation, called Synchronized Transitions, for describing circuits as collections of simple concurrent processes. The tools supporting this notation rely on several compilers that 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: verifying a description that is a manually constructed "abstraction" of the circuit.

This paper discusses one aspect of circuit verification, namely, how to localize a correctness proof to reflect the structure of a circuit designed as a hierarchy of subcircuits. We formalize properties of a circuit as invariants, and we use a theorem prover both to show that the circuit design preserves the invariants and to derive interesting consequences from the invariants. We have also been able to show that circuit designs meet various constraints required for them to be implemented in a particular technology.

Our initial experience with this approach was encouraging. In [3] we observed that the principal advantage of machine-based proofs is that they 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 our circuits and the invariants used to reason about them. A final benefit of

*Research supported by the Danish Research Council. Address: Computer Science Department, Building 344, DK-2800 Lyngby, Denmark. E-mail: jst@iddth.dk

†Research supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125, by the National Science Foundation under grant CCR-8706652, and by NYNEX. Address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA. E-mail: garland@lcs.mit.edu, guttag@lcs.mit.edu

machine-based proofs is that they are easily repeatable. This allowed us to do regression testing when we changed the circuits or the invariants used to reason about them.

The first part of this paper (Sections 2–5) contains an introduction to our verification technique; further details can be found elsewhere [3, 4]. Readers already familiar with our approach based on LP and Synchronized Transitions may wish to go directly to the second part.

The second part of this paper (Sections 6–7) describes mechanisms for decomposing designs and correctness proofs into independent parts that can be considered separately. Although Synchronized Transitions has mechanisms for decomposition, previous correctness proofs [3] have been based on "flat" circuit descriptions from which all structure has vanished. While circuit synthesis necessarily flattens a design, it is not satisfactory for circuit verification to do the same.

2 Synchronized Transitions

A circuit consists of registers and combinational functions. To structure a design, registers are grouped into state components changed by the combinational functions. In the Synchronized Transitions notation, state components are denoted by state variables and combinational functions by transitions.

The computational model underlying the Synchronized Transitions notation is to view a circuit as a collection of asynchronously computing automata. Each step of a computation is performed by a transition, which may involve any number of automata. Viewed from any one of them, a transition takes the automaton from one state to another.

Transitions specify state changes using an imperative guarded-command notation similar to that found in many high-level programming languages. For example, the transition

$$\ll \neg reql \rightarrow grl := false \gg$$

is performed only when its precondition $\neg reql$ holds, and its action leads to a state in which $grl = false$. In general, the precondition of a transition is a boolean expression and the action is a simultaneous assignment.

A large number of transitions are needed to describe any nontrivial circuit. Each transition corresponds to a piece of circuitry. Transitions are atomic, as indicated by the notation $\ll \ldots \gg$; thus each transition appears to be executed indivisibly. In [9, 8] it is explained how the atomicity of transitions is preserved in circuit implementations where many transitions are performed simultaneously. All transitions are independent; thus, there is no global thread of control determining the order of execution. This is an abstraction of a circuit, which is capable of simultaneously executing all its parts.

A transition need not be performed immediately after its precondition becomes satisfied, and there is no upper bound on when it occurs. This is an abstraction for delays within a circuit. For example, $\ll true \rightarrow y := a \vee b \gg$ describes an *or* gate. The precondition *true* specifies that the transition is always allowed to set the output to the *or* of the inputs; however, an arbitrary delay may elapse between a change in the inputs and a change in the output.

The Synchronized Transitions notation has been used on a variety of circuits, both synchronous [8] and asynchronous [9]. Several aspects of the notation have been omitted from this brief description. Further details may be found in [8]. There are strong similarities between Synchronized Transitions and UNITY, as developed by Chandy and Misra [1]. Both describe a computation as a collection of atomic conditional assignments without any explicit flow of control. UNITY proposes this as a general programming paradigm. The goal of Synchronized Transitions is more restricted, namely, development of special purpose VLSI circuits.

3 An example

An *arbiter* is a circuit that provides indivisible access to some shared resource, e.g., a bus or a peripheral. The arbiter described here is implemented as a binary 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 requests the token by setting *req* to *true*.
2. When *gr* becomes *true*, the node has the token and may pass it down the tree.
3. The token is handed back by setting *req* to *false*.
4. When *gr* becomes *false*, a new request can be made.

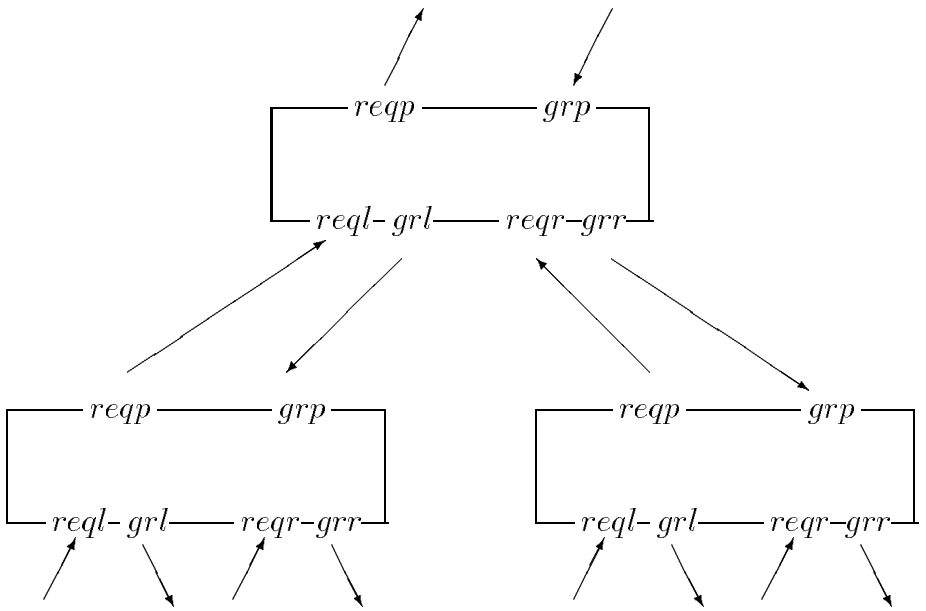

Figure 1: Two levels of arbiter tree

Figure 1 shows a few nodes and their interconnections. The signals *reqp* and *grp* at each node are connected to *reql* and *grl*, or to *reqr* and *grr*, 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. A transition at the root copies *reqp* to *grp* so that the root is able to grant its own request immediately.

The following invariant states that an arbiter node grants access to at most one child, and that access is granted to a child only when the parent has both requested and been granted access.

$$\forall\, n \in nodes:\ I(n) = I_1(n) \wedge I_2(n)$$

$$I_1(n):\ \neg(n.grl \wedge n.grr)$$
$$I_2(n):\ (n.grl \vee n.grr) \Rightarrow (n.grp \wedge n.reqp)$$

The arbiter operates by performing the following transitions at its nodes.

$$(*\ requestparent\ *) \quad \ll \neg grp \wedge (reql \vee reqr) \rightarrow reqp := true \gg$$

$$(*\ grantleft\ *) \quad \ll grp \wedge reqp \wedge reql \wedge \neg grr \rightarrow grl := true \gg$$

$$(*\ grantright\ *) \quad \ll grp \wedge reqp \wedge reqr \wedge \neg grl \wedge \neg reql \rightarrow grr := true \gg$$

$$(*\ doneleft\ *) \quad \ll \neg reql \rightarrow grl := false \gg$$

$$(*\ doneright\ *) \quad \ll \neg reqr \rightarrow grr := false \gg$$

$$(*\ done\ *) \quad \ll grp \wedge \neg grl \wedge \neg grr \rightarrow reqp := false \gg$$

This arbiter can be modified to make it fair. However, the simple version shown here is sufficient to illustrate the points made in this paper. Furthermore, since this paper does not describe how circuits are initialized, our proofs only show that invariants are preserved, i.e., once they hold they continue to hold after execution of any transition.

4 Synopsis of LP: the Larch Prover

LP is a theorem prover, based on equational term-rewriting, for a fragment of first-order logic. It has been used to analyze formal specifications written in Larch [5], to reason about algorithms involving concurrency [3], and to establish the correctness of hardware designs [3, 7]. The intended applications of LP motivate several departures in features and design from other term-rewriting programs and theorem provers.

- LP permits users to define theories by means of equations, induction schemes, and other (nonequational) rules of deduction. Induction schemes, for example, enable us to prove that the mutual exclusion property holds for an arbiter with arbitrarily many levels.
- LP is designed to work efficiently with large sets of large equations. It has been used to verify circuit designs described by several hundred equations.
- LP provides facilities that allow users to establish subgoals and lemmas during the proof of a theorem. It supports a variety of proof methods beyond those found in conventional equational term-rewriting. In the intended applications of LP, rewriting techniques such

as normalization, completion [6], and proof by consistency [2] are often inapplicable, awkward, or computationally too expensive. Other techniques, such as proofs by cases or induction, often lead to better results.

- LP is most often used to debug a specification or a set of invariants. Hence, it is more important to report when and why a proof breaks down than to automatically explore many avenues for pushing a proof through to a successful conclusion. For this reason, LP does not employ heuristics to derive subgoals automatically from conjectures. Instead, it relies largely on forward rather than backward inference, with the user rather than the program being responsible for inventing useful lemmas.

Details concerning LP can be found in [2, 4].

5 Invariance proofs

The goal of an invariance proof is to show that a predicate I involving state variables is preserved by all transitions. Since all transitions are atomic, this is equivalent to showing $\{I\}T\{I\}$ for every transition T, i.e., that I holds after T provided it holds before.

There is considerable latitude in setting up proofs within this framework. When using LP, one begins with a library of common types, e.g., booleans and natural numbers. Associated with each type is a set of operators, together with axioms and rules of inference, that can be used for reasoning about predicates built using those operators. This establishes the basic theory in which proofs are done.

To prove that a transition preserves an invariant, we axiomatize the transition by a deduction rule that describes the relation between the state before the transition occurs (the *pre* state) and the state after it has occurred (the *post* state). To distinguish between the values of a state variable in the pre and post states, we append *pre* or *post* to the variable, e.g., $(n.grl).pre$ and $(n.grl).post$. Using this notation, we can define the effect of the *done* transition being executed at node n in the arbiter tree by the LP deduction rule

when	$T(n)$	
yield	$(n.grp).pre \wedge \neg(n.grl).pre \wedge \neg(n.grr).pre$	% Precondition
	$\neg(n.reqp).post$	% Action
	$(x.grp).post = (x.grp).pre$	% Unchanged
	$(x = n) \vee ((x.reqp).post = (x.reqp).pre)$	

which causes LP to deduce four conclusions describing the *done* transition whenever it discovers that $T(n)$ is true, e.g., whenever the user asserts $T(n)$.

We can also formulate the fact that the invariant I holds at node x in state s by the LP equation

$$I(x,s) \;==\; \neg((x.grl).s \wedge (x.grr).s) \wedge ((x.grl).s \vee (x.grr).s) \Rightarrow ((x.grp).s \wedge (x.reqp).s)$$

To show that each transition preserves the invariant, we must show

$$[\forall x\ I(x, pre) \wedge \exists n\ T(n)] \Rightarrow \forall x\ I(x, post)$$

A proof of this fact is initiated by the LP commands

add $I(x, pre)$
prove $T(n) \Rightarrow I(x, post)$

We have used this technique to verify that several circuits (such as the arbiter) preserve their invariants [3]. Until recently, we have translated invariants and transitions, expressed in Synchronized Transitions notations, by hand into LP commands. We are currently developing a translator that generates these commands from circuits described with Synchronized Transitions. A primary goal is to have the structure of a proof in LP follow the structure of the circuit description. The rest of this paper shows how this can be achieved for Synchronized Transitions programs described as hierarchies of cells.

6 Composition of cells

To make the design and verification of large circuits tractable, it is necessary to have mechanisms for decomposing designs and correctness proofs into independent parts that can be considered separately. This section presents the *cell* construct, which is used to modularize programs in Synchronized Transitions. The next section describes an associated proof strategy.

A cell encapsulates part of a design. It may contain local state variables, transitions, and subcells. The following cell defines the behavior of a subcircuit *part* that sets its local state variable x to $false$.

$CELL\ part;$
  $STATE\ x:\ BOOLEAN;$
$BEGIN$
  $\ll x \rightarrow x := false \gg$
$END\ part;$

The internal structure of a cell is hidden from the rest of the circuit. The scope of state variables (and formal parameters) is defined to be the cell in which they are declared; the scope does not include subcells. Cells communicate through parameters to coordinate their computations. For example, the following cell communicates through the boolean parameter *reset*.

$CELL\ part(reset : BOOLEAN);$
  $STATE\ local : BOOLEAN;$
$BEGIN$
  $\ll reset \rightarrow local := false \gg$
$END\ part;$

An instance of a cell is created by instantiating the formal parameters of the cell by actual parameters located in the body of another cell, e.g.,

$BEGIN$
  $part(init)$
END

An actual parameter may be read and written by both the instantiating and the instan-

tiated cell, unless access to the parameter is restricted by the mechanism described in Section 6.2.

A single state variable must not be known under two (or more) different names in a single cell. This can be ensured by requiring that all actual parameters passed to a cell instance are distinct. It is outside the scope of this paper to discuss how this is verified.

Cells may be nested to describe a hierarchy of subcircuits. For example,

$$
\begin{array}{l}
CELL\ component(reset, in, out : BOOLEAN); \\
\quad STATE\ temp : BOOLEAN; \\
\quad CELL\ part(reset, o, t : BOOLEAN); \\
\quad BEGIN \\
\quad\quad \ll \neg reset \rightarrow o := t \gg \\
\quad END\ part; \\
BEGIN \\
\quad part(reset, out, in) \parallel part(reset, temp, in) \\
END\ component;
\end{array}
$$

describes a cell with an internal subcell that is instantiated twice with different actual parameters. In general, a cell defines the set of transitions obtained by instantiating all elements in the body of the cell.

6.1 Conditional instantiation and recursion

Conditional instantiations can be used to control whether or not a particular subcell is instantiated. A conditional instantiation has the form $\{B \mid subcell\}$, where B is a boolean expression. Since it must be possible to determine the value of B statically, B may not involve state variables. Instead B involves *static parameters*, which have values that can be determined at compile time for each instance of a cell. In the case of the arbiter, a static parameter could be the level of the instance in the arbiter tree.

Cells may be described recursively, i.e., a cell may instantiate a local copy of itself. This allows for a very compact description of the arbiter introduced in Section 3.

$$
\begin{array}{l}
CELL\ arbiter(grp, reqp : BOOLEAN;\ STATIC\ level, top : INTEGER); \\
\quad STATE\ grl, reql, grr, reqr : BOOLEAN; \\
BEGIN \\
\quad (*\ requestparent\ *) \quad \ll \neg grp \wedge (reql \vee reqr) \wedge \rightarrow reqp := true \gg \quad \parallel \\
\quad ... \\
\quad \{\ level < top \mid arbiter(grl, reql, level + 1, top) \parallel arbiter(grr, reqr, level + 1, top)\ \} \\
END\ arbiter;
\end{array}
$$

Appendix A contains a complete description of the arbiter. It includes descriptions for all transitions, the invariant, and additional specifications described in the following sections.

6.2 Restricting the use of state variables

A state variable passed as an actual parameter may generally be read and written both by the instantiating and instantiated cell. Access to selected state variables may be restricted

in one of three ways to prevent them from being written in a particular scope.

- *External write.* A formal parameter may be changed only externally, i.e., only in a surrounding cell instance.
- *Local write.* A state variable or formal parameter may be changed in this scope only, i.e., in neither internal nor external cell instances.
- *Internal write.* A state variable or formal parameter may be changed in internal cell instances only, i.e., in instances to which it is passed as an actual parameter.

These restrictions facilitate formal verification and optimization of circuits synthesized from programs in Synchronized Transitions. They are written among the declarations in a "restriction part," e.g,

$$
\begin{array}{l}
RESTRICTION \\
\quad grp : EXTERNAL; \\
\quad reqp, grl, grr : LOCAL; \\
\quad reql, reqr : INTERNAL;
\end{array}
$$

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.

7 Proof rules for cell instantiations

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. Ideally, we would like to associate invariants with cells and to show locally, rather than globally, that all transitions preserve the invariants.

We refer to the arbiter to illustrate several aspects of this plan of attack. The global proof strategy in Section 5 involved assuming $\forall x\, I(x, pre)$, i.e., that the invariant holds in all cell instances, and attempting to show $\exists n\, T(n) \Rightarrow \forall x\, I(x, post)$, i.e., that the invariant still holds in all cell instances after a transition T of cell instance n. A local, but inadequate, proof strategy would involve showing $\forall n[I(n, pre) \wedge T(n) \Rightarrow I(n, post)]$, i.e., that if the invariant holds at a cell instance where a transition occurs, then it still holds at that cell instance after the transition. A local proof strategy would also describe each transition T in terms of its effect on a single cell rather than in terms of its effect on the entire circuit. The key difference between the local and global plan of attack is that (locally) we assume less and are required to prove less.

There are two problems with this approach. One is that we may not be able to assume less just because we need to prove less. Hence we may need to strengthen invariants if we are going to assume that they hold at fewer cell instances, and then we may need to work harder to show that the stronger invariants are preserved. This problem does not materialize in the case of the arbiter.

The second problem arises when an invariant involves parameters. The method just outlined suffices to establish *local invariance,* i.e., to show that invariants expressed solely

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 i in the arbiter. It passes state variables as actual parameters to two subcells, $arbiterPart1(i)$ (its left child) and $arbiterPart2(i)$ (its right child), and it receives state variables as formal parameters from a supercell, $arbiterParent(i)$. We must show that the transitions of these other cells preserve the invariant $I(i, s)$. For example, we must show that

$$\forall i[\, I(i, pre) \wedge I(arbiterPart1(i), pre) \wedge T(arbiterPart1(i)) \Rightarrow I(i, post)\,]$$

for each transition T of the arbiter. Similar steps are required for $arbiterPart2(i)$ and $arbiterParent(i)$.

Unfortunately, the number of proof obligations grows very rapidly with this approach: if each cell has n neighbors and t transitions, then there are nt proof obligations to discharge. Accordingly, we prefer to abstract a common protocol from the many transitions of a cell and use this protocol when proving noninterference, thereby reducing the number of proof obligations from nt to n. The protocol captures the essential properties of the cell, for example, that a node in the arbiter observes the four-phase protocol described in Section 3. In more detail, we establish noninterference as follows.

7.1 Scope rules

The scope rules of Synchronized Transitions ensure that variables not passed as parameters are invisible outside a cell. For example, $arbiterPart1(i)$ receives only grl and $reql$ as actual parameters from its parent cell i, and hence the variables $i.grp$, $i.reqp$, $i.grr$, and $i.reqr$ are invisible to $arbiterPart1(i)$. The fact that these variables cannot be changed by any transition of $arbiterPart1(i)$ can be formalized in LP by the following deduction rules and equation, which also define several overloaded boolean-valued functions *unchanged* that simplify stating other facts about circuits.

when $inArbiterPart1(i)$ **yield** $unchanged(i.grp, i.reqp, i.grr, i.reqr)$
when $unchanged(sv1, sv2, sv3, sv4)$ **yield**
$unchanged(sv1, sv2), unchanged(sv3, sv4)$
when $unchanged(sv1, sv2)$ **yield** $unchanged(sv1), unchanged(sv2)$

$unchanged(sv) == (sv.post) \Leftrightarrow (sv.pre)$

7.2 Restrictions

The restrictions on the use of variables in a cell (see Section 6.2) also provide information for use in a proof of noninterference. These restrictions can be formalized for the arbiter

in LP by the following deduction rules.

when $arbiterLocalsUnchanged(i)$ **yield** $unchanged(i.reqp, i.grl, i.grr)$
when $arbiterInternalsUnchanged(i)$ **yield** $unchanged(i.reql, i.reqr)$
when $arbiterExternalsUnchanged(i)$ **yield** $unchanged(i.grp)$
when $inArbiterCell(i)$ **yield** $arbiterExternalsUnchanged(i)$
$arbiterInternalsUnchanged(i)$
when $inArbiterParent(i)$ **yield** $arbiterLocalsUnchanged(i)$
$arbiterInternalsUnchanged(i)$
when $inArbiterPart1(i)$ **yield** $inArbiterCell(arbiterPart1(i))$
$arbiterExternalsUnchanged(i)\ arbiterInternalsUnchanged(i)$

7.3 Protocols

The three pairs of state variables $(grp, reqp)$, $(grl, reql)$ and $(grr, reqr)$ all follow a four-phase protocol, which can be defined in LP as holding for the formal parameters of cell instance i by the equation

$$P(i) == if(same(i.reqp, i.grp, post), unchanged(i.reqp), unchanged(i.grp))$$

where *same* is defined by the equation $same(sv1, sv2, s) == (sv1.s) \Leftrightarrow (sv2.s)$. The definition of P captures the four-phase protocol, i.e., that grants follow requests: if grp is the same as $reqp$ after a transition, then it must have changed (because $reqp$ is unchanged); if it differs, then $reqp$ must have changed.

7.4 Proof obligations

Each transition T of a cell instance i must preserve the invariant of the cell and obey the protocol of the cell and its subcells. These proof obligations are defined by the following LP equation.

$$PostObl(i) == I(i, post) \wedge P(i) \wedge P(arbiterPart1(i)) \wedge P(arbiterPart2(i))$$

To show local invariance, we must verify

$$\forall i[\, I(i, pre) \wedge inArbiterCell(i) \wedge T(i) \Rightarrow PostObl(i)\,]$$

for each transition T of the arbiter.

Showing noninterference consists of two steps: (1) verifying that transitions of a parent do not interfere with the cell, and (2) verifying that transitions of subcells do not interfere with the cell. In both steps we must show that $PostObl(i)$ is satisfied. To do this, we can assume that the invariant holds in the prestate, that the scope rules and restrictions are obeyed, and that the transitions obey the protocol of the cell (in step 1) or the protocol of the subcells (in step 2). Thus, we must show the following:

1 $\forall i[\, I(i, pre) \wedge inArbiterParent(i) \wedge P(i) \Rightarrow PostObl(i)\,]$
2*a* $\forall i[\, I(i, pre) \wedge inArbiterPart1(i) \wedge P(arbiterPart1(i)) \Rightarrow PostObl(i)\,]$
2*b* $\forall i[\, I(i, pre) \wedge inArbiterPart2(i) \wedge P(arbiterPart2(i)) \Rightarrow PostObl(i)\,]$

This method of proof has several additional 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 the number of neighbors of that cell). 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, e.g., when a grandchild of a cell instance can change an actual parameter supplied by the grandparent.

Appendix B contains a complete set of LP commands for using this method to verify that the arbiter preserves its invariants. Although the set of commands was constructed by hand, it could have been generated by a translator from the program shown in Appendix A.

8 Conclusion

Automated theorem provers can provide substantial assistance when verifying that circuits described with Synchronized Transitions preserve invariants. To make verification practical for large circuits, Synchronized Transitions allows circuits to be described as hierarchies of subcircuits. Protocols defined for each subcircuit permit them to be verified one at a time. This localized proof technique factors the verification of a circuit into manageable pieces, making machine assisted verification both simpler and faster.

Acknowledgements

James Saxe, Hans Henrik Løvengreen, and Mark Greenstreet made helpful comments on preliminary versions of this paper and on the use of LP for this type of proof.

References

[1] K. M. Chandy and J. Misra, *A Foundation of Parallel Program Design*, Prentice Hall, 1987.

[2] S. J. Garland and J. V. Guttag, "Inductive Methods for Reasoning about Abstract Data Types," *Proceedings of the 15th ACM Conf. on Principles of Programming Lang.*, 1988.

[3] S. J. Garland, J. V. Guttag and J. Staunstrup, "Verification of VLSI Circuits using LP," *Proceedings of the IFIP WG 10.2, The Fusion of Hardware Design and Verification*, North Holland 1988.

[4] S. J. Garland and J. V. Guttag, "An Overview of LP: the Larch Prover," *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, Springer-Verlag, 1989.

[5] J. V. Guttag and J. J. Horning, "Report on the Larch Shared Language" and "A Larch Shared Language Handbook," *Science of Comp. Prog.* 6:2 (Mar. 1986), 103–157.

[6] D. E. Knuth and P. B. Bendix, "Simple Word Problems in Universal Algebras," in *Computational Problems in Abstract Algebra*, J. Leech (ed.), Pergamon Press, Oxford, 1969, 263–297.

[7] J. Saxe, Private communication.

[8] J. Staunstrup and M. R. Greenstreet, "From High-level Descriptions to VLSI Circuits", *BIT* 28:3, 1988.

[9] J. Staunstrup and M.R. Greenstreet, "Designing Delay Insensitive Circuits using Synchronized Transitions. Part I: Introduction and Motivation, and Part II: The Formal Model," submitted for publication.

Appendix A. Description of arbiter

Following is a complete recursive description of an arbiter using Synchronized Transitions.

```
CELL arbiter(grp, reqp: BOOLEAN; STATIC level, top: INTEGER);

  STATE grl, reql, grr, reqr: BOOLEAN;

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

  INVARIANT
    not(grl AND grr)  AND  ( (grl OR grr ) => (grp AND reqp) )

  PROTOCOL
    if(same(reqp, grp, post), unchanged(reqp), unchanged(grp))

BEGIN

  (* requestparent *)
  << NOT grp AND (reql OR reqr) -> reqp := TRUE >> ||

  (* grantleft *)
  << grp AND reqp AND reql AND NOT grr -> grl := TRUE >> ||

  (* grantright *)
  << grp AND reqp AND reqr AND NOT grl AND NOT reql -> grr := TRUE>> ||

  (* doneleft *)
  << NOT reql -> grl := FALSE >> ||

  (* doneright *)
  << NOT reqr -> grr := FALSE >> ||

  (* done *)
  << grp AND NOT grl AND NOT grr -> reqp := FALSE >> ||

  { level < top | arbiter(grl, reql, level+1, top) ||
                  arbiter(grr, reqr, level+1, top)
  }

END arbiter;
```

Appendix B. Localized invariance proof for arbiter

The following input to LP produces a proof that the arbiter preserves its invariants. Comments indicate how this input could be produced by a compiler from the Synchronized Transitions description in Appendix A.

```
%------------------- Generic output from ST compiler
set name definitions
declare
  pre, post  :                                          -> State
  unchanged  :  StateVar                                -> Bool
  unchanged  :  StateVar, StateVar                      -> Bool
  unchanged  :  StateVar, StateVar, StateVar            -> Bool
  unchanged  :  StateVar, StateVar, StateVar, StateVar  -> Bool
  same       :  StateVar, StateVar, State               -> Bool
  on, off, . :  StateVar, State                         -> Bool
  sv, sv1    :: StateVar
  sv2, sv3   :: StateVar
  s, s1      :: State
  ..
add
  unchanged(sv)    == (sv.post) <=> (sv.pre)
  same(sv, sv1, s) == (sv.s) <=> (sv1.s)
  on(sv, s)        == sv.s
  off(sv, s)       == not(sv.s)
  ..
add-deduction-rules
  when unchanged(sv, sv1) yield unchanged(sv) unchanged(sv1)
  when unchanged(sv, sv1, sv2) yield unchanged(sv) unchanged(sv1, sv2)
  when unchanged(sv, sv1, sv2, sv3) yield
       unchanged(sv, sv1) unchanged(sv2, sv3)
  ..
%----------------- Declarations derived from ST program for arbiter
declare
  .                :  ArbiterInstance, VarId           -> StateVar
  I                :  ArbiterInstance, State           -> Bool
  arbiterPart1,
  arbiterPart2     :  ArbiterInstance                  -> ArbiterInstance
  inArbiterSubcell :  ArbiterInstance, ArbiterInstance -> Bool
  i, sub, super    :: ArbiterInstance
  P, PostObl, inArbiterCell, inArbiterParent, inArbiterPart1,
  inArbiterPart2, actionTaken, done, doneleft, doneright,
  grantleft, grantright, requestparent, parentAssumptions,
  part1Assumptions, part2Assumptions, arbiterLocalsUnchanged,
  arbiterInternalsUnchanged, arbiterExternalsUnchanged
                                  : ArbiterInstance -> Bool
  grp, grl, grr, reqp, reql, reqr : -> VarId
  ..
```

```
%------------------ Bindings of actual/formal parameters
set name instantiations
add
  % instantiation of arbiter(grl, reql) within arbiter instance i
    arbiterPart1(i).grp  == i.grl
    arbiterPart1(i).reqp == i.reql
  % instantiation of arbiter(grr, reqr) within arbiter instance i
    arbiterPart2(i).grp  == i.grr
    arbiterPart2(i).reqp == i.reqr
  ..
%------------------ Consequences of scope rules and restrictions
set name restrictions
add-deduction-rules
  % Derived from restrictions section
    when arbiterLocalsUnchanged(i) yield unchanged(i.reqp, i.grl, i.grr)
    when arbiterInternalsUnchanged(i) yield unchanged(i.reql, i.reqr)
    when arbiterExternalsUnchanged(i) yield unchanged(i.grp)
    when inArbiterCell(i) yield
         arbiterInternalsUnchanged(i) arbiterExternalsUnchanged(i)
    when inArbiterParent(i) yield
         arbiterLocalsUnchanged(i) arbiterInternalsUnchanged(i)
    when inArbiterSubcell(sub, super) yield
         arbiterLocalsUnchanged(super) arbiterExternalsUnchanged(super)
         inArbiterCell(sub)
  % Consequences of scope rules and restrictions
    when inArbiterParent(i) yield
         unchanged(i.grl, i.reql, i.grr, i.reqr)     % From scope rules
    when inArbiterPart1(i) yield
         unchanged(i.grp, i.reqp, i.grr, i.reqr)     % From scope rules
         inArbiterSubcell(arbiterPart1(i), i)        % From restrictions
    when inArbiterPart2(i) yield
         unchanged(i.grp, i.reqp, i.grl, i.reql)     % From scope rules
         inArbiterSubcell(arbiterPart2(i), i)        % From restrictions
  ..
%------------------ Definitions of invariant, protocol
set name invariant
add
  I(i, s) == (off(i.grl, s) | off(i.grr, s)) &
             ( (on(i.grl, s) | on(i.grr, s)) =>
               (on(i.grp, s) & on(i.reqp, s)) )
  P(i) == if(same(i.reqp, i.grp, post),
             unchanged(i.reqp),
             unchanged(i.grp))
  PostObl(i) == I(i, post)              &     % Invariant
                P(i)                    &     % Protocol for cell
                P(arbiterPart1(i))      &     % Protocol for part 1
                P(arbiterPart2(i))            % Protocol for part 2
  ..
```

```
%------------------ Definitions of transitions
set name transitions
add-deduction-rules
  when done(i) yield
       on(i.grp, pre) off(i.grl, pre) off(i.grr, pre)   % Precondition
       off(i.reqp, post)                                % Action
       unchanged(i.grl, i.grr)                          % Unchanged
  when doneleft(i) yield
       off(i.reql, pre)                                 % Precondition
       off(i.grl, post)                                 % Action
       unchanged(i.reqp, i.grr)                         % Unchanged
  % Other transitions omitted
  when actionTaken(i) yield
       inArbiterCell(i)
       I(i, pre)
       done(i) | doneleft(i)   % other transitions omitted
  ..
%------------------ Proof obligations for local invariance
prove actionTaken(i) => PostObl(i) by case actionTaken(i)
  resume by cases done(c_i) doneleft(c_i)  % Generated by compiler
  resume by case on(c_i.grr, pre)          % Not generated by compiler
%------------------ Proof obligations for noninterference
set name noninteference
add-deduction-rules
  when part1Assumptions(i) yield
       I(i, pre)
       P(arbiterPart1(i))
       inArbiterPart1(i)
  when part2Assumptions(i) yield
       I(i, pre)
       P(arbiterPart2(i))
       inArbiterPart2(i)
  when parentAssumptions(i) yield
       I(i, pre)
       P(i)
       inArbiterParent(i)
  ..
prove part1Assumptions(i) => PostObl(i) by case part1Assumptions(i)
prove part2Assumptions(i) => PostObl(i) by case part2Assumptions(i)
prove parentAssumptions(i) => PostObl(i) by case parentAssumptions(i)
  resume by case on(c_i.grl, pre) | on(c_i.grr, pre)  % Not generated
```