A PROLOG-BASED CONNECTIVITY VERIFICATION TOOL

Alexander C. Papaspyridis

Information Engineering Section, Department of Electrical Engineering,
Imperial College, London SW7 2BT, England

ABSTRACT

A connectivity verification program implemented in Prolog is presented in this paper. The major advantage of this program, called VERCON, over existing approaches is that it always works, irrespective of circuit topology. VERCON's approach to connectivity verification is to extract all the different designer specified subcircuits from the flat transistor description. Verification is achieved when the top-level object is extracted and there are no transistors which were not used to form the top-level object. Although VERCON is a research prototype, several valuable conclusions have been drawn that will aid the design of a new connectivity verification program written in C.

1. INTRODUCTION

Connectivity verification can be summarised as the task of proving that the structure of two circuits is identical. This is a subtask of layout verification, in which the layout of a circuit is validated against its transistor level description in order to confirm that all transistors are present and connected as specified. Such verification is sought, for example, to confirm that a human designer has laid-out a circuit from its transistor level specification correctly. Connectivity verification is a very important stage in the overall validation of VLSI circuits, because layout design is often manually performed and is therefore prone to errors. Such errors are difficult to locate, because of the high complexity of VLSI circuits.

The task of checking a transistor netlist extracted from a layout against its specification, which was entered either graphically using a schematic capture editor, such as GRACE [1], or textually using a language such as *netlist*, is known by various names like network comparison, netlist comparison, interconnection check and connectivity audit. In this paper the term connectivity verification has been adopted, because it correctly emphasizes connectivity as the circuit attribute under examination.

To verify connectivity, the most common approach has been to simulate the extracted circuit and compare the results with those obtained for its specification. However, this approach has the disadvantage that it cannot show conclusively whether any difference in results can be attributed to connectivity errors rather than parasitics present in the layout. On the other hand, use of dedicated connectivity verification tools has been limited, mainly due to their lack of robustness.

Various attempts at connectivity verification have been made and Spickelmier [2] and Barke [3] provide several references to previous work. Most of the existing connectivity verification tools are based on the so-called graph traversal algorithms. These view the circuit as a connected graph and check connectivity by proving isomorphism between the graphs of the extracted and the specified circuit. The important ingredient of a graph traversal algorithm is that of determining, given a node on the graph, which branch of the graph will be followed next. To decide on that, the algorithms look either to adjacent nodes, adjacent devices, or a mixture of both.

The two basic algorithms that are currently employed in connectivity verification are *signature calculation* and *path tracing* [4,5]. Two examples that use these approaches are the programs GEMINI [6] and WOMBAT [2]. GEMINI uses a graph isomorphism algorithm that partitions graphs using vertex invariants and then iteratively refines the partitioning until all partitions consist of a single node. The YNCC_{FLT} [7] system follows, generally, a graph traversal algorithm, but uses some additional criteria to filter-out subcircuits, that although do not affect the logical operation of the circuit, they do not match topologically to the specification. However, this approach should not be seen as a new one, but rather as complementing the conventional one, particularly for semi-custom design styles. The NECOM program [3] also employs a graph traversal algorithm based on heuristics for partitioning the circuit graph. The approach taken there was to assign a weight to each node, the weight depending on the number and type of devices connected to it. The algorithm then partitions the circuit graph using the node weights information. According to NECOM's designer, this approach was aimed at and is particularly suited to analog bipolar circuits.

The graph traversal algorithms are known [3,4] to deteriorate under certain circumstances, usually when examining highly symmetric circuits. Two such cases occur when terminal permutable objects such as NOR gates or parallel paths such as RAMs are encountered. To overcome these drawbacks, a different approach was followed in the CV program [4,5]. CV is a rule-based system, implemented in OPS5, that takes a global approach by comparing groups of elements. To achieve this, it uses the designer's hierarchy to form rules that match equivalent groups of elements. However, certain assumptions were made which in the author's opinion can affect its correct operation. A more extensive treatment of CV's limitations will be presented later.

The global comparisons approach was judged to offer a significant advantage over conventional local comparisons methods, and therefore it has also been adopted here for a system called VERCON (VERify CONnectivity). VERCON does not compare two transistor level circuits, but instead uses the designer's hierarchical decomposition of the circuit to form Prolog rules describing the connectivity of all different subcircuits. Each of the rules is then matched against the database of facts that represent the transistor level description. Whenever such a match is achieved, it means that a particular combination of objects form one of the user defined subcircuits, and so this subcircuit is extracted from the database of facts. Verification of the layout is achieved when the highest-level object in the hierarchy can be extracted.

The program is written in Quintus Prolog, which is based on Edinburgh Prolog [8]. Prolog was chosen for several reasons: First, its high expressivity makes it an ideal vehicle for prototyping. Since the investigation had an exploratory nature, it was important that various ideas should be implemented rapidly, and various alternatives investigated with minimum programming effort. Second, its rule-base orientation and its pattern matching capability are essential elements of the connectivity verification process. Prolog has been used to implement a variety of CAD systems for VLSI [9, 10, 11, 12, 13, 14, 15], but it is thought to be the first time it has been applied to the connectivity verification area.

The structure of this paper is as follows: In Section 2, a detailed description of each of the system's components will be presented. In Section 3, VERCON's performance with various circuits will be presented in terms of correctness of operation and speed of execution. The achievements and limitations of the system will be discussed in Section 4, together with the extensions needed for the development of a connectivity verification tool that can handle VLSI circuits. Finally, in Section 5 the major points of this work will be summarised.

2. SYSTEM DESCRIPTION

A structure chart of VERCON is shown in Figure 1. VERCON's operation can be decomposed into three parts: data preprocessing, subcircuit extraction and postprocessing of the extracted subcircuits.

The preprocessors convert the transistor netlist and the hierarchical description of subcircuits into Prolog format and also extract the underlying circuit hierarchy. In the following stage, a Prolog program first extracts inverters, NOR and NAND gates, and subsequently another Prolog program extracts all other objects specified in the hierarchical description of subcircuits file. Finally, a preprocessor checks that any objects extracted for the top-level circuit description do correspond to the transistor netlist. In this Section, the operation of each of VERCON's modules will be described.

2.1 Preprocessors

The first operation performed by VERCON is to translate the transistor netlist file and the hierarchical description of subcircuits file into Prolog rules and facts. The first preprocessor (module A, Figure 1), written in C, converts the transistor netlist, described in *.sim* format, to Prolog facts. For example the transistor:

p qbar 1 vdd 2.00 4.00 r 0 0 8.00

will be converted to the Prolog fact:

ptrans(2, qbar, 1, vdd).

where 2 is the reference number for this particular transistor.

A second preprocessor, written in C, converts the hierarchical circuit structure, to Prolog rules and extracts the underlying hierarchy (modules B and C in Figure 1). For example, the description of a latch constructed out of NOR gates in netlist is:

```
(macro latch (q qbar r s)
     (nor q r qbar)
     (nor qbar s q))
```

This will be converted to the Prolog rule:

```
latch(Q, QBAR, R, S) :-
     nor(Q, R, QBAR),
     nor(QBAR, S, Q).
```

. . . and the associated fact:

requires(latch, nor).

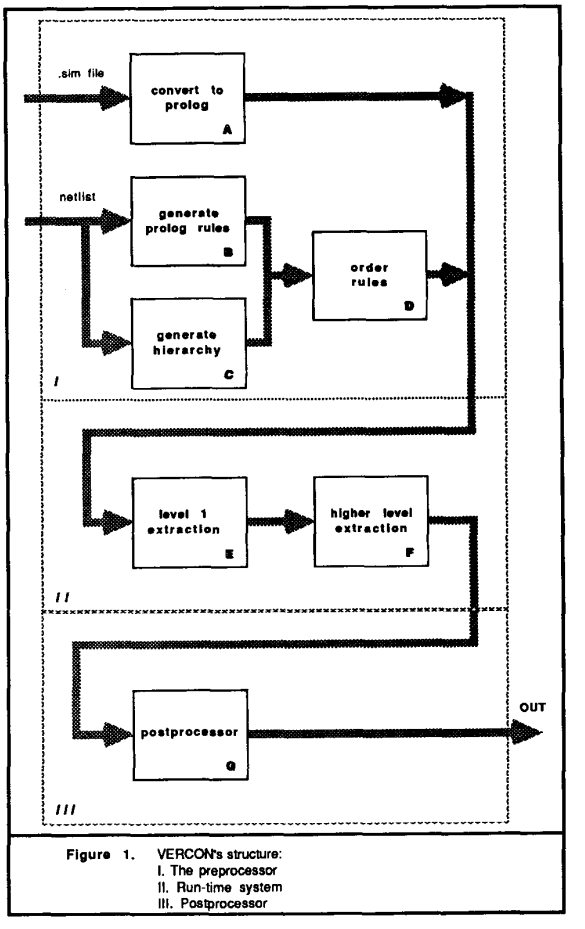

Figure 1. VERCON's structure:
I. The preprocessor
II. Run-time system
III. Postprocessor

It should be noted that in Quintus Prolog the terms that start from lower case are constants, while the terms ones that start from upper case are variables.

The semantics of the Prolog rule above follow closely the netlist semantics and is as follows: there exists an object **latch** with terminals **Q, QBAR, R, S** *if* there is an object **nor** with terminals **Q, R, QBAR** *and* another object **nor** with terminals **QBAR, S, Q**. The notation adopted here represents a device as a fact, with predicate name the device name, while the terminals of the device are shown as the fact's arguments. The structure of an object is represented as the conditions to a rule having the object's name. Inside the rules, connectivity between devices is represented by the use of the same variable for the terminals of the devices that are connected together. This approach has also been used by other researchers like Woo [15], and Clocksin [16], where it is referred to as the *definitional method.*

From the discussion above, it can be seen that the topological description of circuits mapps elegantly to Prolog rules and facts, which means that Prolog, with no extensions, can be readily used as a language to describe hardware structure. Figure 2 illustrates VERCON's operation through the use of a simple example of a latch constructed out of NOR gates, by showing each of the steps taken in verifying connectivity. In the upper part of Figure 2, the action of the two preprocessors is shown.

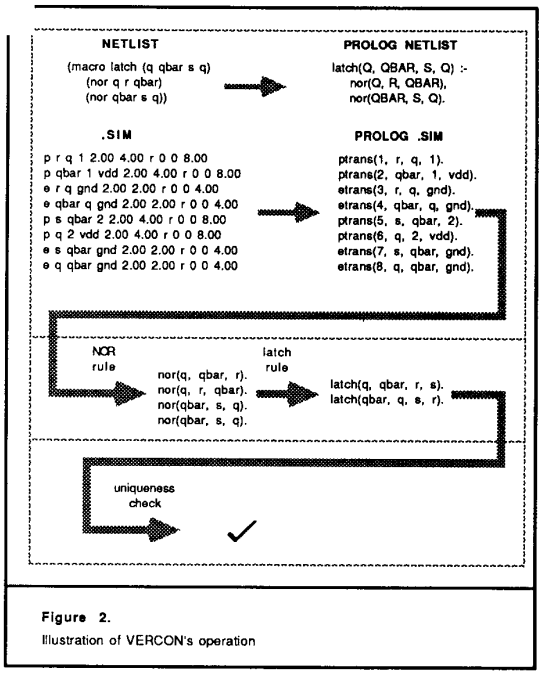

Figure 2.
Illustration of VERCON's operation

2.2 An Initial Implementation of a Connectivity Verification Tool

After preprocessing, both the flat circuit description as a set of transistors, and the hierarchical decomposition of the circuit in terms of smaller subcircuits are expressed in Prolog. Connectivity verification can then be simply performed by using Prolog's deductive mechanism to show that the top-level description of the circuit can be proved. Therefore, the connectivity verification program could simply consist of the single query:

?- circuit(*external_connections*).

where **circuit** is assumed to be the top-level description.

Alas, our connectivity verification program has to be more complex than this single-line query for a number of reasons: Firstly, proof of existence for the top-level object, generated from the query above, is not a necessary and sufficient condition for connectivity verification. Secondly, the query may generate as answer a configuration that may not be physically realizable, for example by using the same object twice. Finally, a third reason is that for computational efficiency it would be preferable to store the intermediate solutions of the extraction process.

Proof of existence for the top-level object is not a necessary and sufficient condition for connectivity verification, because besides the top level object, some other objects may also be present and hence isomorphism between the two circuits cannot be proved. In other words, it is possible that the top-level description of the circuit can be proved, but at the same time the database may contain some extra transistors which are not used to form this top-level object.

A configuration that is not physically realizable is the three transistor configuration of a 2-input NMOS NOR gate can also be interpreted as an inverter with a pulldown transistor at its output. It is clear that both interpretations of this circuit cannot hold simultaneously, but Prolog's deductive mechanism does not prevent this from occurring.

Computational efficiency considerations dictate the use of a forward-chaining control mechanism instead of a backward-chaining one, since any subcircuits extracted are stored in the database, and thus the extraction process does not have to be repeated. Another advantage is that a post-mortem analyser can make use of the extracted objects to determine the parts of the transistor description that violate the structural specification of the circuit.

2.3 The Ordering Program

For the reasons mentioned above, a forward chaining type of approach is taken in VERCON. This calls for extracting all different types of objects, starting from the ones that are lowest in the hierarchy. In terms of Prolog's operation, the essential element of this process is that only facts may be used to prove a rule and not other rules.

VERCON orders the subcircuits describing rules according to the level of hierarchy they belong to, by using the **requires** relation described before (module D in Figure 1). This ordering process cannot use an ordinary sorting program for this purpose, as the order of the subcircuits is not *linear*. For two circuits A and B, if A uses B, either directly, or through a subcircuit of it, then A is higher than B. However, if A does not use B and B does not use A, then it cannot be inferred that A = B, as would be the case for *linear* ordering. The method used here to order the subcircuits employs depth-first traversal of the circuit hierarchy tree. Each node is then assigned a value corresponding to its depth. If a node is assigned a number of values, by being encountered again, then the effective value is the largest. When traversal of the tree is complete, subcircuits are grouped according to their depth, starting from the ones with the largest.

2.4 Extraction of level 1 Objects

Subsequently, all inverters, NOR and NAND gates, termed *level 1 objects*, are extracted, using inherent rules (module E in Figure 1). Inherent rules are used for two reasons: Firstly, netlist descriptions usually do not contain level 1 object definitions. Secondly, since level 1 descriptions are used in almost all designs, significant time is saved in execution if the rules are tuned for computational efficiency. It is estimated that execution time for this stage has been reduced by 40% in this way. Extraction of level 1 objects is illustrated in the lower section of Figure 2, where NOR gates have been extracted. Note also that for each NOR gate, two objects are stored; this signifies that the gate has permutable inputs. For example, the following two Prolog facts refer to the same object:

nor(q, r, qbar).
nor(q, qbar, r).

2.5 Higher Level Extraction

After extraction of level 1 objects, higher order extraction follows (module F in Figure 1). As explained previously, each of the rules that describe the different subcircuits is matched against the database of facts. Whenever a new object is matched, its description is entered into the database together with information as to what objects it comprises, the latter is exploited by the postprocessor.

The matching process will now be explained briefly, using the **latch** rule shown in Figure 2. To satisfy the goal which is shown as the head of the clause, i.e. **latch(Q, QBAR, R, S)**, the conjunction of clauses in the body of the rule must be satisfied. So Prolog attempts to satisfy the first clause, i.e. **nor(Q, R, QBAR)**, all arguments of which are free (not bound). This clause is matched to the first fact available, i.e. **nor(q, qbar, r)**. This means that the following variable unifications have occurred: **Q** to **q**, **R** to **qbar**, and **QBAR** to **r**.

Subsequently, Prolog attempts to satisfy the second clause, i.e. **nor(QBAR, S, Q)**, but due to the above unifications, Prolog tries to satisfy **nor(r, S, q)**, which is not satisfied by any clause. Prolog then backtracks to the first subgoal and attempts to resatisfy it. The following match is then made: **nor(q, r, qbar)**. Then, as before, the second subgoal is attempted, i.e. now **nor(qbar, S, q)** that is satisfied by **nor(qbar, s, q)** and therefore the parent goal is satisfied:

latch(q, qbar, r, s).

Prolog then attempts to resatisfy the original goal by backtracking, which yields the following match:

latch(qbar, q, s, r).

Although these two descriptions are equivalent, they are both available for matching by higher level rules, and therefore the fact that **latch** object is terminal permutable does not affect the verification process.

It should be stressed that VERCON automatically handles *topologically* permutable objects, but not electrically permutable ones. The reason is that electrical permutability cannot be deduced from connectivity information, but requires knowledge of the electrical properties of the circuit devices. Although NOR and NAND gates are not topologically, but only electrically permutable, permutations of them are ostensibly created since they are frequently used with their inputs permuted.

2.6 Postprocessing of the Extracted Subcircuits

Once all rules describing the circuit structure have been matched, results are checked by the postprocessor (module G, Figure 1) for two things: First, to see whether a top level instantiation exists, and secondly, to ensure that no redundant objects are present. If both of the above can be confirmed, then the connectivity has been verified. The former calls for finding the top level macro and consequently to check whether any such macro has been extracted. If this is confirmed, then another procedure decomposes the top level macro in terms of subcircuits it uses, and then recursively each of the subcircuits until all the transistors that form the top level macro are obtained. If no other transistors are left, then connectivity is verified.

3. SYSTEM PERFORMANCE

VERCON consists of about 700 lines of C code and 650 lines of **Quintus Prolog** code, structured in 45 rules, and runs on a **VAX-750** under **Unix**. This is the second implementation of VERCON, the first [17], being in **sigma-Prolog** . The current implementation has retained only the basic philosophy of the original version since the program was modified extensively to make use of the new built-in functions. A sample trace of VERCON's run for the example of Figure 2 is shown in Fig. 3.

There are two constituent aspects of performance: correctness of operation and speed of execution. As it concerns the former, VERCON has been used to verify the connectivity of a variety of circuits, the largest being about 1100 transistors. VERCON *always* worked correctly in a single run.

The preprocessors are fast requiring less than 9 CPU seconds even for the largest circuit. VERCON requires about 20 minutes of CPU time to verify the connectivity of a CMOS circuit of approximately 580 transistors, which means that VERCON is significantly slower than the other programs mentioned before. Compared to the initial implementation, VERCON's speed has been improved more than an order of magnitude.

The variation of computer time with circuit size, i.e. the order of the algorithm, is a very important aspect, since it determines whether the program will be useful for large circuits. In VERCON's case, it was found that the algorithm employed is no worse than those for CV, GEMINI and WOMBAT and is less than 2. This was determined by comparing the execution times for these programs using the information reported in [4, 5] for CV and WOMBAT and data obtained from running VERCON and GEMINI with various circuits by the author. It should be stressed that the comparison made is a very crude one, since only a small number of examples were available. The variation of the execution time was obtained only as a function of circuit size in transistors, since this is the major quantifiable aspect of circuit complexity. However, other aspects of circuit structure also affect significantly the execution time, and hence for any precise comparison of execution time, the same circuits must be used with the various programs, and several different types of circuits must be considered.

```
Quintus Prolog Release 1.6 (VAX Unix 4.2BSD)
Copyright (C) 1986, Quintus Computer Systems, Inc.  All rights reserved.

Thu Mar  3 15:30:50 GMT 1988
Directory: /titanic/research/alex/cv/package
[Using leashing stopping at [call,redo] ports]
[/titanic/research/alex/prolog.ini consulted (0.400 sec 44 bytes)]

|?-  run.
$Header: rest.pl,v 1.3 88/02/17 20:43:24 alex Exp $
*** Verifying circuit connectivity ***
$Header: sim2prol.c,v 1.6 88/02/16 16:19:51 alex Exp $
sim2prol: job completed
$Header: net2prol.c,v 1.3 88/02/17 20:28:17 alex Exp $
net2prol: job completed
[/titanic/research/alex/cv/package/simfile.pl consulted (0.983 sec 416 bytes)]
Performing level 1 extraction
$Header: l1xtr.pl,v 1.5 88/02/17 20:39:24 alex Exp $
Looking for inverters ...done
Looking for nor gates ...done
Looking for nand gates ...done
Level 1 extraction completed
[/titanic/research/alex/cv/package/simfile.pl consulted (0.884 sec 352 bytes)]
[/titanic/research/alex/cv/package/netfile.pl consulted (0.383 sec 172 bytes)]
[/titanic/research/alex/cv/package/hierfile.pl consulted (0.117 sec 28 bytes)]
Hierarchy levels: 2
Number of macros: 1
Hierarchical order of macros: [latch]
Higher level extraction ...
Trying : latch
Higher level extraction completed
Checking uniqueness ... done
Top level objects:
latch(q,qbar,r,s).
latch(qbar,q,s,r).
*** Verification proved in 4.0 CPU Secs

yes
```

Figure 3.
A trace of a sample VERCON run

4. DISCUSSION

Compared to existing connectivity verification programs, the strong point of VERCON is that it is the only system that always works, irrespective of circuit connectivity. VERCON does not depend on any assistance from human designers, as some other systems do for "difficult" circuits.

VERCON has also a number of advantages over the CV system. CV [4] handles terminal permutable objects through the use of multiple rules for the description of this object: for example, a 4-input NMOS NOR gate would require 4 ! rules, i.e. 24. To avoid this the designer must incorporate terminal permutability information in the appropriate rule. In contrast to CV, in VERCON a single rule is used for each object, irrespective of terminal permutability and without designer specified terminal permutability information.

Another limitation of CV, overcome in VERCON, is the constraint that each object may only be matched once [5]. Although this is true most of the time, it is a constraint that limits the generality of the approach. An example where the constraint could lead to problems, is when considering an NMOS NOR gate and an inverter. If the inverter rule was to be used first, then it would match the load and one of the driver transistors, and therefore delete them from memory. Hence, the NOR rule would not be matched, resulting in failure to verify the circuit. It is anticipated however, that this particular example will not cause CV to malfunction, because this effect is apparent.

VERCON is slower than other systems, but is still useful to verify circuits up to a few thousand transistors. It should be stressed though, that the aim was to achieve full functionality without any speed concerns. No major effort was expended on improving the system's speed. VERCON's slow speed of execution can be primarily attributed to two factors: firstly, to the slow speed with which Prolog executes in conventional hardware and secondly, to the way Prolog conducts its search.

The Prolog system used has a rating of approximately 15 KLIPS, while other combinations of software and conventional hardware have been rated up to 50 KLIPS. However, a rating of 200 KLIPS may be reached in the very near future through the use of dedicated Prolog processors, which in our case will mean an improvement in speed of another order of magnitude.

With regard to Prolog's search mechanism, as shown in Section 2.5, the search mechanism when trying to satisfy a goal is linear search of all clauses whose heads match the goal to be satisfied. Although this is sufficient when looking globally for an object and in fact provides the cornerstone of this research effort, most of the times what is needed is just the objects that are connected to a certain node. Such a data structure could be implemented efficiently in a language such as C, but is infeasible in Prolog where the only types of data structures available are facts and rules.

5. CONCLUSION

A connectivity verification system, based on Prolog, that can handle any type of circuits has been demonstrated. The system uses the designer's hierarchy and a transistor level description of the circuit to form appropriate Prolog rules and facts which describe the circuit. It then uses Prolog's pattern matching capability to extract the various subcircuits used in the design.

VERCON makes no assumptions about the circuit connectivity and thus works under any circumstances, something not accomplished before. The approach followed aimed to give full functionality without being concerned about the speed of execution. The system is relatively slow, but is useful for circuits that comprise up to a few thousand transistors. Moreover, the order of the algorithm has been found to be no worse than that of other approaches. This means that VERCON in its current form can handle circuits up to a few thousand devices with run times less than 10 hours of CPU time.

It has been shown that a global view of the circuit under examination is necessary to overcome problems introduced by symmetric circuits. For efficiency reasons though, this capability has to be coupled with local view of the circuit that minimizes the amount of search. Implementation of such features is not feasible in Prolog, but can be programmed in a language like C. VLSI circuits could then be verified in reasonable CPU times.

Despite Prolog's inability to efficiently implement various structures and operations, it has proved to be very efficient for prototype programming as used here. Its main advantages are high expressivity, pattern matching capability, the backtracking mechanism, and the powerful metalanguage facilities that enable the manipulation of both the program and the data.

REFERENCES

[1] A.C. Papaspyridis, "GRACE: A Schematic Capture Editor", Internal Report, Department of Electrical Engineering, Imperial College of Science and Technology, London, UK, June 1985.

[2] R.L. Spickelmier, and A.R. Newton, "WOMBAT: A New Connectivity Verification", Proceedings of the International Conference on Computer-Aided Design, 1983.

[3] E. Barke, "A Network Comparison Algorithm for Layout Verification of Integrated Circuits", IEEE Transactions on Computer-Aided Design, Vol. CAD-3, No.2, April 1984, pp. 135-141.

[4] C. Lob, R. Spickelmier, and A.R. Newton, "Circuit verification using rule-based expert systems", Proceedings of the 1985 International Symposium on Circuits and Systems, Kyoto, Japan, June 5-7, 1985, pp. 881-884.

[5] R.L. Spickelmier, and A.R. Newton, "Connectivity Verification Using a Rule-Based Approach", Proceedings of the 1985 International Conference on Computer-Aided Design, Santa Clara, USA, November 18-21, 1985, pp. 190 - 192.

[6] C. Ebeling, "Using Gemini to Validate Circuit Layout", Internal Report, Computer Science Department, Carnegie-Mellon University.

[7] Y. Shiran, "$YNCC_{FLT}$: A New Algorithm for Filtering and Comparing Different but Logically Equivalent VLSI Networks", Proceedings of COMPEURO 87, Hamburg, West Germany, May 11-15, 1987, pp. 215 - 218.

[8] W.F. Clocksin, and C.S. Mellish, "Programming in Prolog", Springer-Verlag, Berlin 1987, pp. 1-281.

[9] H.G. Barrow, "Proving the Correctness of Digital Hardware Designs", VLSI Design, July 1984, pp. 64-77.

[10] P.J. Drongowski, "A graphical, rule-based assistant for control graph-datapath design", Proceedings of the ICCD, October 1985, pp. 208- 211.

[11] M. Fujita, S. Kono, H. Tanaka, and T. Moto-Oka, "Aid to hierarchical and structured logic design using temporal logic and Prolog", IEE Proceedings, Vol. 133, Pt. E, No. 5, September 1986, pp. 283-294.

[12] P.W. Horstmann and E.P. Stabler, "Computer Aided Design Using Logic Programming", Proceedings 21st Design Automation Conference, June 1984, pp. 144-151.

[13] K. Mitsumoto, H. Mori, T. Fujita, and S. Goto, "AI approach to VLSI routing problem", Proceedings of the 1984 International Symposium on Circuits and Systems, Montreal, Canada, May 7-10, 1984, pp. 449-452.

[14] N. Suzuki, "Concurrent Prolog as an efficient VLSI design language", IEEE Computer, February 1985, pp. 33-40.

[15] N.S. Woo, "A Prolog Based Verifier for the Functional Correctness of Logic Circuits", Proceedings of the International Conference on Computer Design, Port Chester, USA, October 7-10, 1985, pp. 203 - 207.

[16] W.F. Clocksin, "Logic Programming and Digital Circuit Analysis", Journal of Logic Programming, 4, 1987, pp. 59-82.

[17] A.C. Papaspyridis, "The Use of Prolog for Connectivity Verification", Proceedings of the 1988 IEEE International Symposium on Circuits and Systems, Espoo, Finland, June 6-9, 1988.