Automatic Specialisation of Standard Designs

W. F. CLOCKSIN

Computer Laboratory, University of Cambridge, Corn Exchange Street, Cambridge CB2 3QG

We introduce a technique, called specialisation, for automatically performing a certain class of optimisation in hierarchically specified designs composed of standard modules. Modules are subject to specialisation when they provide facilities that are not used in a design. This situation is not uncommon in 'gate array' and 'standard cell' designs used in VLSI circuits. New definitions of modules are collected in a library as they are constructed so that detailed processing need not be repeated when the same specialisation is required elsewhere in a design. The technique has been implemented in Prolog. For explanatory purposes the technique has been applied here to specialising 'standard cell' specifications, but the technique is generally applicable for rewriting arbitrary specifications written as collections of Horn clauses.

Received November 1985

1. INTRODUCTION

The use of standard cells[1] has increased the convenience of designing VLSI circuits. However, circuits constructed from standard cells generally contain more components (transistors) than purpose-built circuitry for two reasons.

- The need to present a standard interface. It is necessary to provide internal glue circuitry which is matched on the other side of an interface by more internal glue circuitry.
- The need for a standard cell library to provide a manageably small number of general-purpose cells. Suppliers of standard cell libraries cannot predict in advance the precise functionality required by the designer. A consequence is that some of the functionality of general-purpose cells is not required in a design, causing 'over-designed' circuits to be produced.

The abstraction boundaries provided by the standard cell approach exist for the convenience of the designer. However, once the circuit is out of the designer's control, it is desirable to automatically redesign the circuit to more easily meet constraints imposed by technology. Such redesign would involve removing redundant circuitry that exists because of the two reasons given above, and may result in redefining abstraction boundaries within a circuit. We describe a program, called the Specialiser, that removes redundant components by automatically generating more specialised designs when given a design defined in terms of general-purpose modules. When applied to digital circuit design, this technique can be seen as a compromise between the ultimate goal of silicon compilation and the need to use existing methodologies similar to standard cells. It is important to note that the purpose of the Specialiser is not to perform layout or to optimise layout. Specialisation is carried out at the level of module connectivity and hierarchy, not at the geometric level.

It is understandable that specialisation has not been considered for previous digital technologies such as SSI and MSI chips, because even if specialisations can be identified, it is impossible to implement them owing to robust packaging: it is impossible to break chips apart. Consequently, previous methods for simplifying circuits have involved well-known techniques such as Boolean optimisation and strength reduction in an attempt to reduce chip count. However, the definitions of standard cells can be manipulated by software, and more sophisticated methods such as the Specialiser become relevant. When applied to digital circuits, the class of components removed by the Specialiser is disjoint to that removed by the well-known simplification techniques. The Specialiser will not remove parts along a datapath having an output that is used elsewhere in a circuit. The reason for this more conservative approach is to prevent the violation of designed-in timing constraints owing to propagation delay through critical paths. Instead, the Specialiser deals with over-general design which does not contribute to the final outputs of the circuit.

The Specialiser works by selectivity ignoring abstraction boundaries imposed by the standard module definitions and by the hierarchical specification of the design, automatically designing new module definitions where required. We observe in digital designs that much redundant circuitry can be identified by tracing unused outputs back through the circuit. Although this tracing is a simple operation that produces a specialised circuit from one constructed from general-purpose cells, this operation when performed manually is often tedious and error-prone as the designer can become lost in the complexity of large circuits.

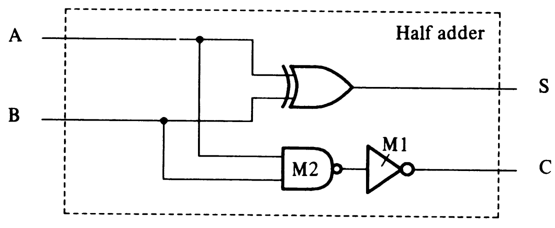
This technique can be computationally complex, but in our implementation efficiency is achieved by exploiting the hierarchical nature of the design specification to suppress unnecessary detail. Moreover, the computations required to specialise a given type of module in a given way are performed only once. The new definition is stored in a library to be used again when required. Thus the work done in specialising a complete design is linear in the number of types of module in its design, rather than linear in the number of primitive components (in the case of digital design, transistors).

2. DESIGN SPECIFICATION

The input to the Specialiser is a design specification. We use Prolog[4] as a specification language in a manner described elsewhere.[3] The specification technique is similar to that used by others.[2, 8] When applied to digital circuits, the specification language is not restricted to combinatorial circuits, and may also be used for directly executing specifications of synchronous sequential circuits and circuits containing components that exhibit bidirectionality, such as pass-transistors. The output of the Specialiser is also a design specification, so the Specialiser may be seen as a source-to-source program transformation.

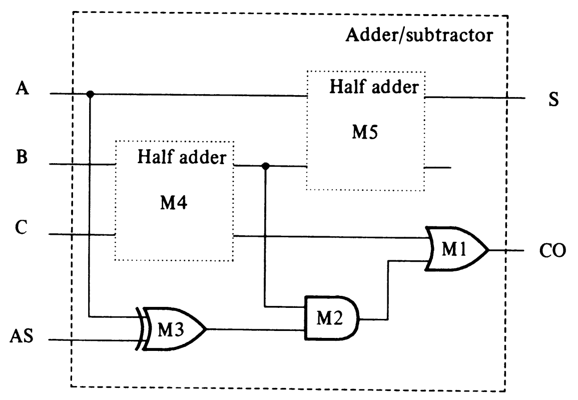
A design is composed of a set of modules and connections between modules. With each module is associated a set of ports between which connections are specified, which may be used for input and output. Modules can be composed hierarchically, by which means modules are specified in terms of other module specifications. At the bottom of the hierarchy are found primitive elements, which depend on the technology being used. In this paper we shall consider cases where digital logic gates are considered primitive, and where *p*- and *n*-transistors are considered primitive.

Refer to Fig. 1. A design is presented as a set of Horn clauses shown here in Edinburgh Prolog syntax. A module having *n* ports is represented as a predicate of arity *n*. A module may be specified as a Horn clause in which the head of the clause represents the module to be defined, and the body of the clause is a composition of submodules defining the module. The submodules comprising a given module are composed with the comma connective. Ports within a given design that share a unique common connection are represented by a unique like-named variable. The ':–' operator is re-interpreted to mean 'is defined by'. The order of the modules in the body of the clause is not important.



(*a*) The half adder is specified as the formula:

```
halfadd(A,B,S,C) :- xor(A,B,S), nand(A,B,T), not(T,C).
```



(*b*) The adder/subtractor is specified as the formula:

```
addsub(A,B,C,AS,S,CO) :-
        halfadd(B,C,T1,T2),
        halfadd(A,T1,S,Z),
        xor(A,AS,T3),
        and(T1,T3,T4),or(T2,T4,CO).
```

Fig. 1

For example, in Fig. 1(*a*), the **halfadd** module is composed of three gates with connections named by variables as shown. The variable **T** defines the hidden connection between the output of the **nand** gate and the input of the inverter. In Fig. 1(*b*), two instances of the half-adder are used to define a full 1-bit adder/subtractor, a circuit which either adds or subtracts depending on the state of the **AS** input. Note the use of void variable **Z** to represent the unused carry-out from the first half-adder.

In this example, we have considered the logic gates to be primitive, and they can be defined (again using Prolog) in terms of their Boolean truth tables. For example, the two-input exclusive-or gate is represented as **xor (A,B,C)** having inputs **A** and **B**, and output **C**,and is defined by the clauses

```
xor(0,0,0).
xor(0,0,1).
xor(1,0,1).
xor(1,1,0).
```

If it is necessary to specify the input and outputs of this module, then the additional unit clause

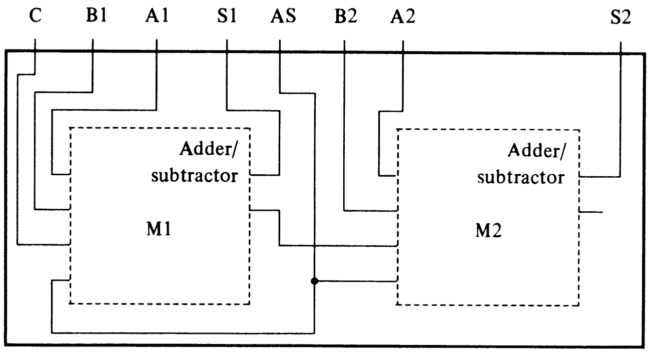
```
direction(xor(A,B,C),[A,B],[C])
```

May be used to project the arguments into a list of inputs and a list of outputs respectively.

3. DESIGN SPECIALISATION

A module can be specialised if it contains at least one unused output. The trivial case of specialisation is when the unused output is the only output of the module. In this case, the entire module can be removed from the design with impunity. Removal of the module may result in the disconnection of outputs from other modules, which can be specialised in turn. Specialisation thus becomes a tree search rooted in each unused output of a module, and propagating back through the inputs to the module. The tree search propagates though the design, inspecting only those parts of the design on a direct path to the unused output.

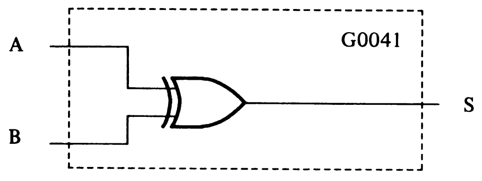
The search through the design must also take account of hierarchical structure. Consider first primitive modules, where specialisation results in no change in the hierarchy. For primitive modules having all outputs unused, the trivial case holds, and the module can be removed. For primitive modules having at least one used output no change can be made, and the search terminates. Now consider non-primitives. If all outputs of a non-primitive module are unused, then the trivial case again holds, and the module is removed. However, given a non-primitive module in which some but not all outputs are unused, it may be necessary to recursively descend the hierarchy into the module definition to determine whether any submodules sourcing the unused outputs can be removed from the definition of the module. Removing submodules may have the effect of rendering redundant some of the inputs to the module, and the search proceeds from there after emerging from the recursion. But, before the search continues, it is necessary to *redefine* the modified module as a specialised version of the original module. In our implemented system, redefined modules are placed in a 'library' from which they can be accessed in case another module of the same specialisation is found elsewhere in



The two-bit adder/subtractor is specified as:

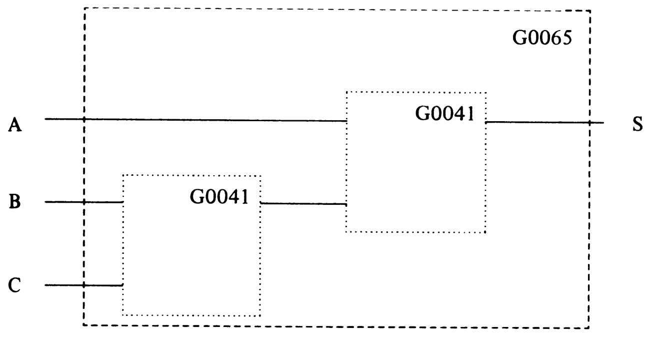
```
twobit(A1,B1,A2,B2,C,AS,S1,S2) :-
        addsub(A1,B1,C,AS,S1,T),
        addsub(A2,B2,T,AS,S2,Z).
```

Fig. 2



(*a*) The generated specification for the resulting specialised module is:

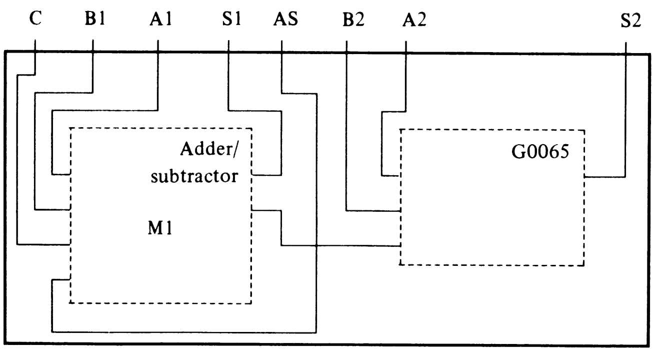
```
g0041(A,B,C) :- xor(A,B,C).
```



(*b*) The generated specification for the resulting specialised module is:

```
g0065(A,B,C,S) :- g0041(B,C,T), g0041(A,T,S).
```

Fig. 3



The generated specification for the resulting specialised two-bit adder/subtractor is:

```
twobit(A1,B1,A2,B2,C,AS,S1,S2) :-
        addsub(A1,B1,C,AS,S1,T),
        g0065(A2,B2,AS,S2).
```

Fig. 4

the circuit. If an attempt is made to specialise a module having a given pattern of unused outputs, the library is first scanned for a definition which, if found, immediately replaces the module under consideration.

Applying this technique to digital circuits can be demonstrated by means of an example shown in Fig. 2. Consider the two-bit adder/subtractor **twobit** which is defined using the modules shown in Fig. 1. The carry output of the last stage of **twobit** is unused. Given the top-level goal of specialising **twobit**, the Specialiser proceeds as follows.

(1) Inspect the modules of **twobit**, finding **addsub** M2 with an unused output. Recursively enter the definition of **addsub** with the goal of specialising it.

(2) The following modules are removed from **addsub**: the or-gate M1, the and-gate M2, and the xor-gate M3. The half-adder M4 now has a disconnected carry output, so recursively enter the definition of **halfadd** with the goal of specialising it.

(3) The following modules are removed from **halfadd** M4: inverter M1 and nand-gate M2. No other module with unused outputs remains in **halfadd**, so now preparations are made to return to the next higher level of the circuit hierarchy. The specialised half-adder is added to the library with a generated unique name, in this case G0041 (see Fig. 3(*a*)). Module **halfadd** M4 is now replaced by module G0041.

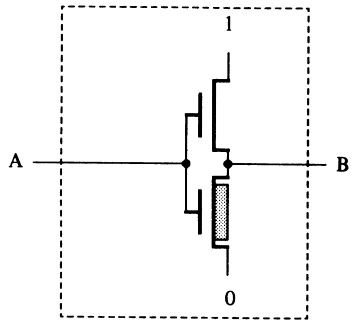
(4) The one remaining module in **addsub** having an unused output is **halfadd** M5. Because a half-adder with an unused carry output exists in the library as module G0041, a copy of G0041 replaces **halfadd** M5. No other module with unused outputs remains in **addsub**, so now preparations are made to return the next higher level of the circuit hierarchy. The specialised adder/subtractor is added to the library with a generated unique name, in this case G0065 (see Fig. 3(*b*)). Module **addsub** M2 in **twobit** is now replaced by module G0065.

(5) Propagation continues along the **AS** input of **addsub** M2, but search terminates because the **AS** line sources a used input (**AS** of **addsub** M1). No other module with unused outputs remains in **twobit**, so now control returns to the top level of the circuit hierarchy.

By convention, the top-level circuit is not added to the library, so the final result is the circuit shown in Fig. 4.

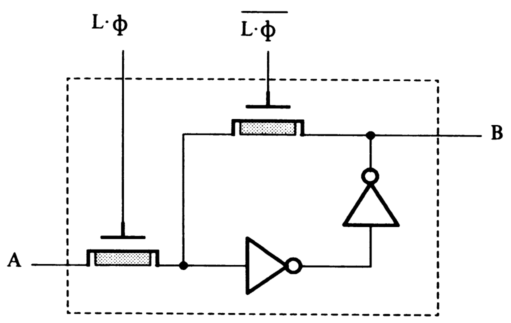
4. EXAMPLES

The Specialiser has been tested on a variety of small modules in the domain of digital design. For each module in a circuit, the Specialiser needs to know the direction (whether input or output) of each port. For testing the Specialiser it is sufficient to manually specify a unit clause for predicate **direction** (see above). However, there are circumstances where is it preferable to decide directions automatically by flow analysis. This is especially useful in CMOS transistors, for which the sources and drain ports can be used as either inputs or outputs interchangeably. A program that decides



(*a*) The CMOS inverter is specified as:

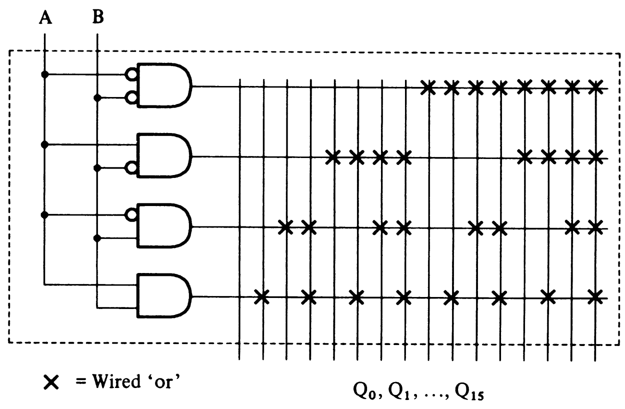
```
inv(A,B) :- ntrans(A,0,B), ptrans(A,1,B)
```



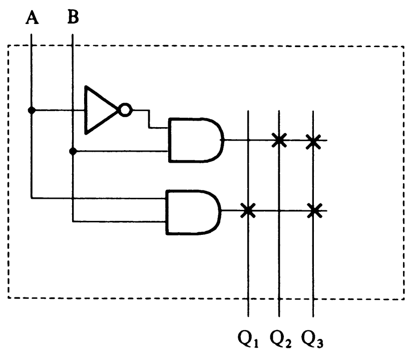
(*b*) The RAM cell is specified as:

```
ram(A,B,L,NL) :-
        ntrans(L,A,T1),
        inv(T1,T2),
        inv(T2,B),
        ntrans(NL,B,T1).
```

Fig. 5



(*a*) The original function generator



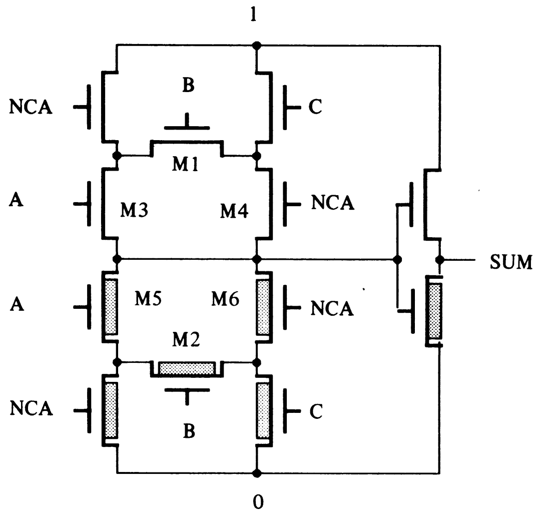
(*b*) The generated specialised function generator

Fig. 6

directions for modules within Horn clause circuit specifications has been written[5] and can be used to pre-process circuits given to the Specialiser.

Fig. 5 shows the specification of a dynamic memory cell[6] together with a specification of the inverter used by the memory cell. An 8-bit register was specified in a circuit, of which only 5 bits were used. The Specialiser correctly generated the specification of a 5-bit register. Fig. 6(*a*) shows a circuit for generating the sixteen functions of two inputs. The function generator was used in a circuit in which only the second to fourth functions were used. The Specialiser correctly generated a specification of the function generator shown in Fig. 6(*b*).

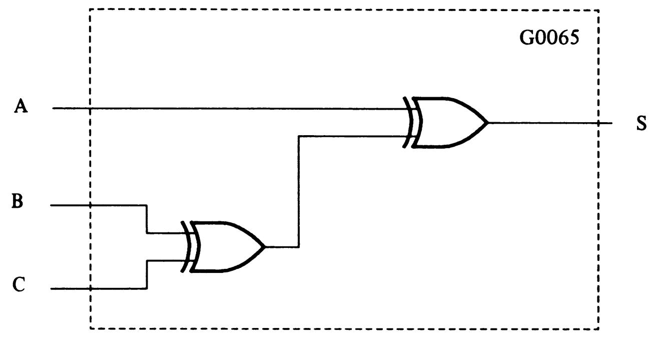
Fig. 7 shows part of a full adder constructed with CMOS transistors. The associated carrygenerator that sources NCA is not shown. The interesting feature of this circuit is the use of two transistors, M1 and M2, which conduct in different directions depending on the inputs to the adder. Therefore the direction of the transistors cannot be determined, and as a consequence the direction of transistors M3–M6 cannot be decided.[5] Nevertheless, the Specialiser is able to make specialisations of this circuit, but transistors M1–M6 are never removed. For example, if the SUM output were unused, then the only action available is to remove the two transistors making up the inverter at the last stage.



The portion of a CMOS full adder that computes the sum is specified as:

```
sumpart(A,B,C,NCA,SUM) :-
    ptrans(NCA,T1,1),
    ptrans(C,1,T5),
    ptrans(B,T1,T5),
    ptrans(A,T1,T2),
    ptrans(NCA,T5,T2),
    ptrans(T2,1,SUM),
    ntrans(A,T2,T3),
    ntrans(NCA,T2,T6),
    ntrans(T2,SUM,0),
    ntrans(B,T3,T6),
    ntrans(NCA,T3,0),
    ntrans(C,T6,0).
```

Fig. 7



The specification resulting from merging two levels of the model hierarchy:

```
g0065(A,B,C,S) :- xor(B,C,T), xor(A,T,S).
```

Fig. 8

5. DISCUSSION

The proprietary design automation aid Chipsmith[7] removes redundant circuitry at the mask and geometric level. Whilst this may achieve a similar effect as Specialisation applied to the particular domain of digital design, the purpose of Specialisation is to *rewrite* the circuit *specification*. Chipsmith does not rewrite the specification. Thus the user will never know how removal of redundant circuitry will change the original specification, and he will not have the advantage of using a modified specification for further processing (the modified netlist may be obtained by using the EXPAND command though). By contrast, the user of the Specialiser will have automatically generated a minimal set of new module specifications which can be examined, placed in a library, and subjected to further processing.

There are further differences between Chipsmith and Specialisation. Chipsmith needs to know input/output specifications. Specialisation does not, because this is provided by another specification-rewriting program.[5] Chipsmith does not exploit the modular decomposition of circuits to aid in removing redundant components. In Chipsmith, the whole circuit has to be flattened before circuitry can be removed. This is not the case with Specialisation, in which the circuit hierarchy is searched, and any effects of removing modules are propagated throughout the structure. This is much less computationally complex than searching a flattened circuit.

Two consequences of using the Specialiser are as follows. First, the technique ignores the timing effects of removing components of a circuit. In some cases, propagation times are calculated carefully by the designer so that the circuit meets a set of temporal constraints. Removing unused components from the circuit may cause the temporal constraints to be violated in subtle ways. Unused components obviously will not be on a datapath for which propagation delay along the path will be relevant. However, unused components may consume power and dissipate heat, and may thus affect propagation delays of physically neighbouring circuitry when removed. Such higher-order effects on design are not taken into account by the specialisation technique reported here.

The second consequence is that the library into which specialised module definitions are stored contains no information about the physical layout of the module. This is due to the specification language for modules, which represents only connectivity and functional constraints, and not geometric constraints. After a circuit has been specialised, therefore, it is necessary to use some other method to generate the layout of each module added to the library. The extent to which layout generation can be performed automatically is an active area of research.

The Specialiser can also be used in a less sophisticated application, choosing modules from a very large cell library. In this application, the designer is familiar with, say, a dozen popular modules, but complete cell layout information for perhaps hundreds of specialised variants is stored in a library. The Specialiser replaces a given module in a circuit only if the desired variant can be found in the library. Otherwise, the module is not changed.

Another area yet to be explored is the automatic modification of the hierarchical structure of the circuit. For example, the specialised circuit G0065 shown in Fig. 3 could well be defined as shown in Fig. 8, merging two levels of the hierarchy. A library entry for the somewhat singular module G0041 would thus not be required. Likewise, in Fig. 4, the definition of G0065 is small enough that the two exclusive-or gates may as well be substituted into the circuit. This situation could be detected by either of several heuristics.

- The contents of very small modules would be substituted directly into a circuit (macro expansion).
- A generated specialised module found to be functionally equivalent to an existing module in the library would not be placed in the library, but the existing library module would be used instead. This is the case with module M0041, which is functionally equivalent to the library entry for an exclusive-or gate.

Acknowledgements

I thank Mike Gordon, Inder Dhingra, Donald Gaubatz and Miriam Leeser for advice.

REFERENCES

1. R. F. Ayres, *VSLI: Silicon Compilation and the Art of Automatic Microchip Design.* Prentice-Hall, Englewood Cliffs, N.J. (1983).
2. J. W. Batten, *Prolog: Its Potential for Hardware Description and Verification.* Department of Computation, University of Manchester Institute for Science and Technology (UMIST) (1983).
3. W. F. Clocksin, *Logic Programming and the Specification of Circuits.* Technical report, Computer Laboratory, University of Cambridge (1985).
4. W. F. Clocksin and C. S. Mellish, *Programming in Prolog.* Springer-Verlag, Heidelberg (1981).
5. W. F. Clocksin and M. E. Leeser, Automatic determination of signal flow through MOS transistor circuits. *Integration* **4**, 53–63 (1986).
6. A. Fusaoka, H. Seki and T. Takahashi, Description and reasoning of VLSI circuit in temporal logic. *New Generation Computing* **2**, 79–90 (1984).
7. Anon., *Chipsmith Brochure.* Lattice Logic, 9 Wemyss Place, Edinburgh (1984).
8. D. Svanæs and E. J. Aas, Test generation through logic programming. *Integration* **2**, 49–67 (1984).