Technical Report

UCAM-CL-TR-41
ISSN 1476-2986

Number 41

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

LCF_LSM, A system for specifying and verifying hardware

Mike Gordon

September 1983

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

LCF_LSM

A system for specifying and verifying hardware

Mike Gordon
Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG

Abstract

The LCF_LSM system is designed to show that it is practical to prove the correctness of real hardware. The system consists of a programming environment (LCF) and a specification language (LSM). The environment contains tools for manipulating and reasoning about the specifications. Verification consists in proving that a low-level (usually structural) description is behaviourally equivalent to a high-level functional description. Specifications can be fully hierarchical, and at any level devices can be specified either functionally or structurally.

As a first case study a simple microcoded computer has been verified. This proof is described in a companion report. In this we also illustrate the use of the system for other kinds of manipulation besides verification. For example, we show how to derive an implementation of a hard-wired controller from a microprogram and its decoding and sequencing logic. The derivation is done using machine checked inference; this ensures that the hard-wired controller is equivalent to the microcoded one. We also show how to code a microassembler. These examples illustrate our belief that LCF is a good environment for implementing a wide range of tools for manipulating hardware specifications.

This report has two aims: first, to give an overview of the ideas embodied in LCF_LSM, and second, to be a user manual for the system. No prior knowledge of LCF is assumed.

N.B. This is the second printing of Tech. Report No. 41. Various corrections and additions have been made.

Contents

Introduction

The LCF_LSM system is designed to show that it is practical to prove the correctness of real hardware systems. As a first case study a simple microcoded computer has been successfully verified. This proof is described in a separate report [Gordon5]. Current research is aimed at applying LCF_LSM to examples supplied by industrial collaborators.

Although LCF_LSM was originally implemented to test ideas in verification, it can also be used for other activities requiring the manipulation of specifications, and for general programming. For example, in the companion report [Gordon5], we show how to derive from the microcode and associated control logic,. an implementation of a hard-wired controller which is functionally equivalent to the microprogrammed controller. We also show how to use the system to code a microassembler. These examples illustrate our belief that LCF (which is the basis of LCF_LSM) is a good environment for implementing a wide range of tools for manipulating hardware specifications.

The name of the system is a concatenation of two acronyms:

LCF: Logic of Computable Functions. A computer system, designed and implemented by Robin Milner and his colleagues, for generating formal proofs interactively [Gordon et. al.].

LSM: Logic of Sequential Machines. A formal system which extends the logical calculus embedded in LCF with terms based on the behaviour expressions of Robin Milner's Calculus of Communicating Systems (CCS) [Milner].

LCF_LSM is implemented as an extension to Cambridge LCF, a descendant of Edinburgh LCF. The version of LCF_LSM described below is still experimental and so has a number of rough edges and inelegancies. It is, however, intended to be sufficiently robust and efficient for use on realistic problems.

Instead of the rather theoretical approach taken earlier (e.g. in [Gordon2]), I have recently been concentrating on practical applications and have left a number of mathematical questions pending. This does not mean that I regard theoretical issues as unimportant - indeed the current system would never have been conceived without the pioneering work of Milne, Milner, Plotkin and others. I hope that by studying the specification of relatively large real systems a new set of interesting theoretical questions will be generated, and so the next phase of work may well have to be mostly mathematical.

This report has two aims: first, to give an overview of the ideas embodied in LCF_LSM, and second, to be a user manual for the system. I have included sufficient description of LCF to enable readers unfamiliar with it to understand the main ideas described here. Serious users of LCF_LSM will, however, need to become familiar with more of LCF than is covered. The book [Gordon et. al.], paper [Paulson1] and reports [Paulson2, Paulson3] provide the necessary documentation. An introductory paper is [Gordon4]. An early, non-mechanized, version of LSM is described in [Gordon2], many of the examples described here originated in these reports.

The kind of behaviour expressible in LSM can also be expressed in several other languages; for example, MIDDLE [Dembinski & Budkowski], temporal logic [Moszkowski] and the "synchronous logic" described in [Ayres] as a source language for silicon compilation. Further comparison of the various notations would be fruitful; perhaps LCF_LSM can be combined with Ayres's techniques to yield a uniform framework for VLSI verification and implementation? It would be nice to be able to prove designs correct before compiling them to silicon.

Another system, with similar goals to LCF_LSM, is described in [Barrow]. This work is based on Prolog rather than LCF.

The approach described in this report seems most convincing when registers, gates etc. are taken as primitive, rather than being defined in terms of lower level technology dependent devices like transistors. The low level behaviour realized by particular technologies (NMOS, CMOS etc.) can be expressed in LSM, but the models studied so far are rather crude [Gordon3]. More detailed behavioural representation at this level is possible in Milne's calculus CIRCAL [Milne] and Hanna's predicate logic based framework VERITAS [Hanna].

Introduction to LCF

The LCF system interfaces to the user via an interactive programming language called ML (for Meta Language). When LCF is first run it will output a prompt character #, the user can then input either an expression (which will be immediately evaluated) or a declaration (which will result in a variable being bound to a value, or a function being defined). Below is an example session; lines typed by the user start with #, all other lined are output by ML.

```
#2+3;;
5 : int

#let x = 2+3;;
x = 5 : int

#let f x = x+1;;
f = - : int -> int
```

```
#f x;;
6 : int
```

First the user inputs the expression *2+3* followed by the terminator *;;* and a carriage return. ML responds by printing the value of the expression and its type. Next the user inputs the declaration *let x = 2+3*, this causes the value of expression *2+3* to be bound to the variable *x*, ML indicates this effect by printing out the new value of *x* and its type. The user then inputs a function declaration *let f x = x+1* which defines f to be the function that adds one to its argument. ML prints function values as -, the type *int -> int* indicates that *f* requires an argument of type *int* (i.e. an integer), and that it also returns a result of type *int*. Finally the user inputs the expression *f x* which applies function *f* to argument *x* resulting in the integer *6*.

let *e*, *e1*, *e2*, ... stand for arbitrary ML expressions, and *x*, *x1*, *x2* for arbitrary ML identifiers (variables), then ML includes the following kinds of expressions:

()

This constant expression (pronounced "empty") denotes the only value of type *void*. This value is typically returned by ML functions whose main effect is a side-effect (e.g. *new_theory* - see below).

0, *1*, *2*, etc.

Evaluates to values of type *int*.

true, *false*

Evaluates to values of type *bool*.

`<list of characters>`

Evaluates to a value of type *tok* (for token). Tokens are ML's version of strings; they are often used to name things.

x

Evaluates to the value currently bound to *x*. ML identifiers can be any sequence of letters, digits, primes (') or underlines (_) starting with a letter. Each ML identifier has an ML type (e.g. in the example session above *x* has type *int*).

e1 e2

Function application: *e1* must evaluate to a function, i.e. a value with a type of the form *ty1->ty2* (see the section below on types for the meaning of ->); and *e2* to a value of type *ty1*. The application *e1 e2* then evaluates to the result of applying the function denoted by *e1* to the value of *e2*.

e1 ix e2

Here *ix* must be one of ML's predefined binary operators. These include +, -, *, /, <, >, =, *or*, *&*.

(e1,e2)

Evaluates to a pair whose first component is *e1*'s value, and whose second component is *e2*'s value. If *e1* has type *ty1* and *e2* type *ty2* then *(e1,e2)* has the product type *ty1#ty2*. The components of a pair can be extracted with the built-in ML functions *fst* and *snd*. For example, *fst(e1,e2) = e1* and *snd(e1,e2) = e2*.

[*e1;...;en*]

Evaluates to a list of the values of *e1*, ... , *en*. Each *ei* must have the same type, ty say, and then the list [*e1;...;en*] has the type *ty list*. The standard list processing functions *hd*, *tl*, *cons* (which can be infixed as .), *null* and the empty list *nil* are built-in. They satisfy:

```
hd [e1;e2;...;en]   = e1
tl [e1;e2;...;en]   = [e2;...;en]
null nil            = true
null [e1;...;en]    = false
e1.[e2;...;en]      = [e1;e2;...;en]
```

if e then e1 else e2

The usual conditional: evaluates to the value of *e1* if *e* evaluates to *true*, and to the value of *e2* otherwise. *e1* must have type *bool*, and *e2* and *e3* the same types.

let d in e

This is a block with local declarations *d* (see below). The expression *let d in e* evaluates to the result of evaluating *e* in a local environment with bindings determined by *d*. For example, *let x=3 in x*x* evaluates to *9*.

In addition to the kinds of expressions just described, ML also has expressions which evaluate to terms, types, formulae and theorems of a logical calculus - the object language (OL). In the LCF system, the object language is called PPLAMBDA (which is an acronym derived from "Polymorphic Predicate Lambda-Calculus"). In the LCF_LSM system there is a different object language called LSM (Logic for Sequential

Machines) which has PPLAMBDA as a subset, but also contains some extra terms. In the descriptions that follow I will not be completely precise about which things are in PPLAMBDA and which are only in LSM - I will usually just refer to the logic as OL.

Constructs of OL have a special syntax, which must be surrounded by quotes when inputting to ML. For example "$x+y$" is an ML expression of type *term* (an OL term), and "$x+y == y+x$" is an ML expression of type *form* (an OL formula). Note that whereas *2+3* is an ML expression of type *int*, "*2+3*" is an ML expression of type *term*. The quotes separate the object language OL from the meta language ML. We shall describe OL shortly, but first we must say a little about ML declarations.

The following are the main kinds of declarations:

let x = e

> This binds identifier *x* to the value of the expression *e*.

let x1,...,xn = e

> Here *e* must evaluate to a value of the from *(v1,...,vn)*, the declaration then simultaneously binds each *xi* to *vi*.

let f x = e

> This defines *f* (which must be an ML identifier) to be the function with formal parameter *x* and body *e*.

let f x1 ... xn = e

> This defines *f* to be a curried function of type *ty1->...->tyn->ty* where each *xi* has type *tyi* and *e* has type *ty*. If *f* is defined by this declaration and *e1* has type *ty1* then *f e1* is an ML expression (called the partial application of *f* to *e1*) of type *ty2->...->tyn->ty*.

let (x1,...,xn) = e

> This defines *f* to be a function of type *ty1#...#tyn->ty*, i.e it takes a vector (tuple) as argument.

let b1=e1 and b2=e2 ... and bm=em

> This simultaneously defines *bi* to be *e1*; *bi* can either be an identifier or something of the form *f x1 ... xn*. For example, *let x = 1 and f y = x+y*; in this the *x* in the body of *f* has whatever value *x* has before the declaration was executed (i.e. it is not necessarily *1*).

In the three kinds of function definitions described above the keyword *let* must be replaced by *letrec* if the function is recursive.

Let *ty*, *ty1* and *ty2* range over arbitrary ML types, then the types of ML include:

bool, *int*, *tok*, *void*, *term*, *form*, *type*, *thm*

These are predefined primitive types. The types *term*, *form*, *type* and *thm* are discussed in detail below.

ty1#ty2

The type of ML pairs whose first component has type *ty1* and second component has type *ty2*. (For example *(true,3)* has type *int#bool*).

ty list

The type of ML lists of values of type *ty*. (For example [*1;2;3*] has type *int list*).

ty1->ty2

The type of functions taking arguments of type *ty1* and returning results of type *ty2*.

The Syntax of LSM

LSM differs from PPLAMBDA in two main ways: first, it has some extra kinds of terms loosely based on the behaviour expressions of CCS [Milner], and second, it does not contain the "undefined values" needed for fixed-point induction (Scott induction). Since I did not need induction for my first case study I simplified things by removing the associated paraphernalia (future studies might result in their reinstatement).

OL is interfaced to ML via four types:

term ML values of type *term* denote OL terms. Each such term has an OL type and denotes a value of that type. These values can be numbers, truthvalues, words, pairs, lists, functions, sequential machines etc. It is important to distinguish ML types and values from OL types and values. For example, *3* is an ML value with ML type *int*, whereas "*3*" is an ML value of ML type *term*, and this term denotes an OL value with OL type *num*.

type ML values of type *type* denote OL types. For example ":*num*" is an ML expression of type *type* denoting the OL type *num*. The syntax of quoted OL types is ":<type expression>".

form ML values of type *form* denote OL formulae. Such formulae are predicate calculus sentences. For example, "!t.t==T \/ t==F" is an ML expression of type *form* which denotes a OL formula that expresses the proposition that for every *t* either *t* is *T* or *t* is *F*.

thm ML values of type *thm* are certain pairs *(fml,fm)* where *fml* is a list of formulae, and *fm* is a formula. Such a pair is interpreted as asserting that *fm* holds if all the formulae in *fml* (called assumptions) hold. For example, for any *fm* the assertion corresponding to *([fm], fm)* always holds. The only way to construct values of type *thm* is to use certain predefined ML procedures called inference rules. For example, the ML function *ASSUME : form -> thm*, when applied to any formula *fm* generates the ML theorem represented by *([fm],fm)*. If *([fm1;...;fmn], fm)* is a pair representing a value of ML type *thm* we write *fm1,...,fmn* |- *fm*. The ML system normally prints such a theorem as ... |- "*fm*" - each assumption is printed as a dot.

Note that terms, types and formulae can be directly input using the quotation syntax (details below), but theorems can only be created by the predefined inference rules. A tutorial introduction to ML and the concepts underlying the representation of OL in ML is [Gordon4]. I strongly suggest reading this if the brief remarks above are confusing.

Here is a session to illustrate the manipulation of OL from ML:

```
#let fm = "!t. t==T \/ t==F";;
 fm = "!t. t==T \/ t==F" : form

#let th1 = ASSUME fm;;
th1 = . |- "!t. t==T \/ t==F" : thm

# let th2 = SPEC "x:bool" th1;;
th2 = . |- "x==T \/ x==F" :thm
```

First we bind *fm* to a formula; then we apply the inference rule *ASSUME* to *fm* resulting in a theorem which we bind to th ML identifier *th1*. Then we use the inference rule *SPEC* to specialize the quantified variable *t* in *th1* to *x*.

We now describe the parts of LSM which are similar (but not identical) to PPLAMBDA; for more details see the Manual [Paulson2]. In the next section we introduce the special terms which enable the behaviour and structure of hardware devices to be expressed.

When describing OL constructs we omit the surrounding quotes needed when inputting to ML (note that in the case of types these quotes are ": followed by ").

We use *t*, *t1*, *t2* etc. to range over terms, and *x*, *x1*, *x2* etc. over variables. The terms of OL are:

0, *1*, *2*, etc

These have ML type *term* and OL type *num*.

T, *F*

These have ML type *term* and OL type *bool*.

#b1...bn

Here each *bi* is either *0* or *1*. The term *#b1...bn* has OL type *word*n; for example, *#01101* has OL type *word5*. OL has an infinite family of distinct types: *word1*, *word2*, etc. to represent bitstrings of different sizes. The type *bool list* can be use to represent variable sized bitstrings. There is also an infinite family of types: *tri_word1*, *tri_word2*, etc. to represent tri-state values. These are needed for representing busses, and are explained in [Gordon5] in the context of the computer example described there.

x

OL variables - any sequence of letters, numbers, primes (') or underlines (_) starting with a letter. Each variable has a OL type which determines the range of values it can take. To explicitly indicate this type one can write *x:ty* instead of just *x*; in the absence of explicit type information the OL type-checker (like the ML one) tries to infer types from context.

t1 t2

Function application: *t1* must have a OL type of the form *ty1 -> ty2*, and *t2* type *ty1*, then the application has type *ty2*; it denotes the result of applying the function denoted by *t1* to argument *t2*.

t1 ix t2

Here *ix* must be one of the standard OL binary operators (e.g. =); see Appendix 1 for details.

(t1,t2)

If *t1* and *t2* have OL types *ty1* and *ty2* respectively, then *(t1,t2)* has OL type *ty1#ty2*.

[*t1;...;tn*]

Evaluates to an OL list of the values of *t1, ... , tn*. Each *ti* must have the same type, *ty* say, and then the list [*t1;...;tn*] has the type *ty list*.

(t -> t1 | t2)

This is the conditional "if *t* then *t1* else *t2*'. *t* must have OL type *bool*, and *t1*, *t2* the same OL type, *ty* say, the conditional term then has type *ty* also. (In PPLAMBDA the conditional has syntax *(t => t1 | t2)* and *t* must have type *tr*).

let x=t1 in t2

This is equivalent to *t2* with all free occurrences of *x* replaced by *t1*. For example, one can prove: *let x=2 in x*x == 2*2.*

Besides numbers,truthvalues and bits there are various built-in constant functions (a constant function is just a constant with a functional type, i.e. a type built using ->). For example, for each n there is a constant *WORD*n of OL type *num -> word*n for converting a number to a bitstring (thus |- WORD5 13 == #01101). See Appendix 1 for a list of the built-in constants.

The user can introduce his own constants by creating a theory (see examples below for details).

OL types include:

bool

The type of OL truth-values *T* and *F*.

num

The type of OL numbers *0, 1* etc.

*word*n

An infinite family of types (one for each n) of bitstrings of various lengths. For example, *word1, word2, word3* etc. are all types of LSM.

tri_wordn

An infinite family of types (one for each n) of tri-state bitstrings of various lengths. For example, *tri_word1*, *tri_word2*, *tri_word3* etc. are all types of LSM. These tri-state types are used to represent the values on busses; there are special constants to represent floating (or high impedance) states (see Appendix 1).

memm_n

An infinite family of types (one for each m and n) of memories with m-bit addresses and n-bit contents. For the computer example described in [Gordon5] the main memory has type *mem13_16* and the control store has type *mem5_30*.

ty1#ty2

The type of pairs *(t1,t2)* where *t1* has OL type *ty1* and *t2* OL type *ty2* respectively.

ty list

The type of lists of values of type *ty*. (For example [*1;2;3*] has OL type *num list*).

ty1->ty2

The type of functions taking arguments of type *ty1* and returning results of type *ty2*.

The user can introduce his own types by creating a theory.

Note the potential for confusing ML and OL - beware!

The formulae of OL include the following (*fm*, *fm1*, *fm2* ... range over arbitrary formulae):

t1 == t2

States that *t1* and *t2* denote the same value. (Note that there is also a term *t1=t2* of OL type *bool*; *(t1=t2)==T* is equivalent to *t1 == t2*)

P(t1,...,tn)

Here *P* is a predicate constant (== is a built-in infixed predicate constant).

!x.fm

Read *!* as "for all" (universal quantification).

?x.fm

Read ? as "there exists" (existential quantification).

~fm

Not *fm* (negation)

fm1 /\ *fm2*

fm1 and *fm2* (conjunction).

fm1 \/ *fm2*

fm1 or *fm2* (disjunction).

fm1 ==> *fm2*

fm1 implies *fm2*

fm1 <=> *fm2*

fm1 if and only if *fm2*

The theorems of OL are determined by the axioms and rules of inference. The axioms are a predefined set of values of type *thm*; see Appendix 2 for a list of the current axioms. The rules of inference are predefined ML functions which return theorems as results. For example there is an axiom called *BOOL_CASES* which is |- *!t:bool. t==T \/ t==F* (*BOOL_CASES* is a predefined ML identifier bound to this theorem); the function *ASSUME* described above is an example of a rule of inference. All the axioms and rules of inference currently implemented in the LSM version of OL are described in Appendix 3.

In order to introduce application dependent constants, types and axioms, one can set up a hierarchy of theories. Each such theory has a name, which is a token, together with a set of constants, types and axioms. Each theory may have zero or more parent theories; the constants, types and axioms of a theory's parents (and parent's parents etc.) are available in the theory. The LCF_LSM system is initialised with a number of built-in constants, types and axioms. I shall refer to these as constituting the theory '*lsm*', this should be thought of as acting like a parent to any theories created by the user; it contains the types *bool*, *num* and *word*n (for each n). For a complete description see Appendix 1.

LSM Terms for Representing Sequential Machines

The terms described above are essentially those of PPLAMBDA (except that PPLAMBDA has three truthvalues *TT*, *FF* and *UU*, and it doesn't have numbers, words or lists built-in). LSM also contains some special terms (based on the behaviour expressions of CCS [Milner]) which we describe in detail below. It is these CCS-like terms which constitute the main difference between LCF and LCF_LSM.

Consider a device with input lines *i1* ,..., *im* and output lines *o1*, ... , *on*:

```
      i1  i2 ...  im
      |   |  ...  |
      |   |       |
   |--------------|
   |              |-------clock
   |              |
   |--------------|
      |   |  ...  |
      |   |       |
      o1  o2 ...  on
```

Suppose this device also has some internal registers *x1*, ..., *xp*, and that it behaves like a sequential machine as follows:

> At each moment in time the values on the output lines are a function of the values in the registers (the state) and the values on the input lines.
>
> The values in the registers stay constant until a clock pulse is received on the clock line. (Exactly how a clock pulse is realized physically is left unspecified - it could, for example, actually be two voltage level changes (two phases: phi1 and phi2), or a single pulse).

LSM contains special terms to represent such sequential machines. These terms have OL type *dev* (for "device").

To specify the behaviour of a machine one must:

> Specify the value on each output line in terms of the values of the state registers and the values on the input lines.
>
> Specify how the state changes when the device is clocked.

As an example consider a counter:

```
switch    in
  |        |
|-----------|
|           |
| COUNT(n)  |-------clock
|           |
|-----------|
     |
     |
    out
```

Here the input lines are *switch* and *in*, the only output line is *out* and the only state variable is n. The name of the device is *COUNT*; we write *COUNT(n)* to show that the behaviour (to be described) depends on n (actually *COUNT* would be an OL constant of type *num->dev*). Suppose the behaviour of *COUNT* is informally specified by:

> The value on the output line *out* is always the value of the state variable n. We can express this with the output equation:
>
> ```
> out = n
> ```
>
> When the counter is clocked, the new value of the state variable n becomes $n+1$ (i.e. the old value plus one) if *false* is being input on line *switch*, otherwise it becomes the value input on line *in*. We can express this by:
>
> ```
> CLOCK(n) --> CLOCK(switch -> in | n+1)
> ```

In LSM the behaviour of the counter would be specified by the formula

```
COUNT(n) == dev{switch,in,out}. {out=n}; COUNT(switch -> in | n+1)
```

This formula has the form *t1 == t2* where *t1* is the term *COUNT(n)* and *t2* is a new kind of term of type *dev* described below. Notice that the clocking is implicit in our notation (i.e. we don't explicitly mention the clock line). From now on we will not draw clock lines in diagrams, though they will still be needed in actual hardware implementations. Our model of behaviour abstracts away from the physical details of how state-changes are effected, and treats devices as abstract sequential machines. I hope the examples below show that this abstraction is justified - i.e. that significant aspects of correctness can still be expressed. The new terms for expressing sequential behaviour are:

dev{x1,...,xm}.{l1=t1,...,ln=tn};t

> A term of this form is called a behaviour term. It denotes a sequential machine whose input and output lines are *x1*, ... ,*xm*. If *li* is not listed among *x1*, ... ,*xm* then it is an internal (or local) line (these will be explained later). The term *ti* gives the value output on line *li*. The new state after clocking is specified by the term *t*. Normally *t* will have the form *D(u1,...,ur)* where *D* is a device name

(e.g. *COUNT*) and *u1*, ... ,*ur* are terms giving the new values of the state variables of *D*. If *li* occurs in *t* or one of the *ti* then its value there is determined by the equations {*l1*=*t1*,...,*ln*=*tn*}. The free variables of the whole behaviour term are the free variables in *t1*, ..., *tn* and *t* minus *x1*,..., *xm*, *l1*, ..., *ln*.

An example of a behaviour term is:

```
dev{i,o}.{o=n};REG(i)
```

This specifies a device that outputs on line *o* the value of variable *n* (which is a free variable of the term) and then, when clocked, becomes a device with behaviour *REG(i)* - i.e. becomes the device *REG* in a state holding the value input in line *i*. Suppose we specify *REG* to satisfy:

```
REG(n) == dev{i,o}.{o=n};REG(i)
```

then this defines *REG* to be a device which always outputs its state, and stores the current value input - i.e. it delays by one clock cycle. Formulae of this form - i.e. of the form:

```
D(a1,...,ap) == dev{x1,...,xm}.{l1=t1,...,ln=tn};D(u1,...,up)
```

are called behaviour equations. They are used to directly specify sequential machines. We will shortly show how to give a structural specification of a machine in LSM also.

Here is another example of a behaviour term:

```
dev{switch,in,out}.{out=n};COUNT(switch->in|n+1)
```

This term also has no internal lines, and again *n* is the only free variable.

The counter informally described above can be specified by the behaviour equation:

```
COUNT(n) == dev{switch,in,out}.{out=n};COUNT(switch->in|n+1)
```

An example of a behaviour term with internal lines is:

```
dev{switch,in,out}.{l1=(switch->in|l2), out=n, l2=out+1};COUNT_IMP(l1)
```

Here *l1*, and *l2* are internal lines; again the only free variable is *n*.

If the device *COUNT_IMP* has a behaviour satisfying:

```
COUNT_IMP (n) == dev{switch,in,out}.
                    {l1=(switch->in|l2),out=n,l2=out+1};
                    COUNT_IMP(l1)
```

Then the rule of inference *UNFOLD_IMP* to be described in Appendix 3 will enable us to "solve" the equations for the lines, and hence derive:

```
COUNT_IMP(n) ==  dev{switch,in,out}.
                    {l1=(switch->in|n+1),out=n,l2=n+1};
                    COUNT_IMP(switch->in|n+1)
```

Note that in this formula *l1* and *l2* are no longer used anywhere. The *PRUNE_EQUATIONS* rule (described in Appendix 3) will enable the equations for these variables to be removed to get:

```
COUNT_IMP(n) == dev{switch,in,out}.{out=n};COUNT_IMP(switch->in|n+1)
```

Note that this equation for *COUNT_IMP* is similar to the equation specifying *COUNT*. A rule called *UNIQUENESS* will enable us to derive from this that:

```
COUNT(n) == COUNT_IMP(n)
```

If the state of a device remains constant over time it is called combinational. Here are two examples:

```
switch i1   i2                        i
  |    |    |                         |
|-------------|                    |-------|
|     MUX     |                    |  INC  |
|-------------|                    |-------|
       |                              |
       |                              |
       o                              o
```

The value on the output line *o* of the multiplexor *MUX* equals the value on line *i1* if the value on line *switch* is *T*, otherwise it is the value on line *i2*. Thus the value on the output line is given by the output equation: *o=(switch->i1|i2)*. The behaviour of *MUX* is thus specified by the behaviour equation:

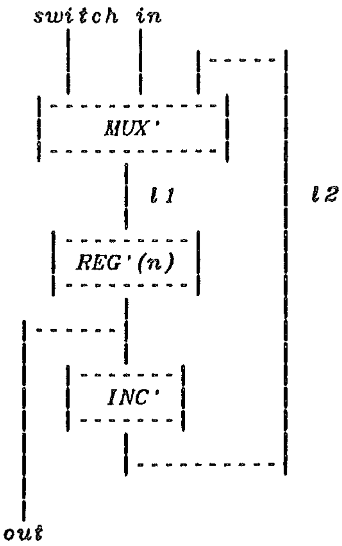
```
MUX == dev{switch,i1,i2}.{o=(switch->i1|i2)};MUX
```

Note the lack of state variables. The behaviour of *INC* is specified by the behaviour equation:

```
INC == dev{i,o}.{o=(i+1)};INC
```

Thus *INC* is a combinational device that always outputs (on line *o*) one plus the value input (on line *i*).

Behaviour terms are used to directly specify what devices are supposed to do. LSM can also be used to describe the structure of digital systems as the interconnection of separate devices. Consider the system below:



This device is built from components similar to *MUX*, *REG(n)* and *INC* as described above, except that the line names have been changed. The multiplexer *MUX'* is like *MUX* except that it has lines *in*, *l2*, *l1* instead of *i1*, *i2*, *o* respectively. The need to rename lines motivates the following kind of term:

t rn[*l1*=*l1'*;...;*ln*=*ln'*]

> Here *t* should be a term of OL type *dev* and *l1*, ..., *ln*, *l1'*, ... ,*ln'* are line names (which must be OL variables). The term denotes a behaviour similar to that denoted by *t* except that each line *li* is systematically renamed to *li'*.

Suppose that *MUX* is as specified above, i.e. it satisfies:

```
MUX == dev{switch,i1,i2,o}.{o=(switch->i1|i2)};MUX
```

Then if we specify *MUX'* by the formula:

```
MUX' == MUX rn[i1=in;i2=l2;o=l1]
```

then it will follow (using the rule *EXPAND_DEF* described in Appendix 3) that:

```
MUX' == dev{switch,in,l2,l1}.{l1=(switch->in|l2)};MUX'
```

Note that line *switch* has not been renamed. The register *REG'* in the diagram above is defined by renaming the lines of the generic register *REG* by:

```
REG'(n) == REG(n) rn[i=l1;o=out]
```

Then we will be able to prove using *EXPAND_DEF* that if *REG* is defined as above, i.e.:

```
REG(n) == dev{i,o}.{o=n};REG(i)
```

then:

```
REG'(n) == dev{l1,out}.{out=n};REG'(l1}
```

Similarly we can define:

```
INC' == INC rn[i=out;o=l2]
```

and then prove:

```
INC' == dev{out,l2}.{l2=(out+1)};INC'
```

Note that instead of defining *MUX'*, *REG'* and *INC'* by renaming lines of *MUX*, *REG* and *INC*, we could have defined them directly by the behaviour equations:

```
MUX'    == dev{switch,in,l2,l1}.{l1=(switch->in|l2)};MUX'
REG'(n) == dev{l1,out}.{out=n};REG'(l1}
INC'    == dev{out,l2}.{l2=(out+1)};INC'
```

If we did things this way, then the above equations would be axioms rather than derived theorems. LCF_LSM gives one the option of either defining each component directly (using a behaviour equation), or as a renaming of a previously defined primitive.

To represent a schematic diagram, the first step is to define its components (either directly, or by renaming) so that lines which are to be connected have the same name. For example, in the diagram above we have arranged that output line of *MUX'* is the same as the input line to *REG'* (viz. *l1*). The next step is to write down a term which denotes the result of connecting together the component devices. LSM has a special kind of term for this purpose:

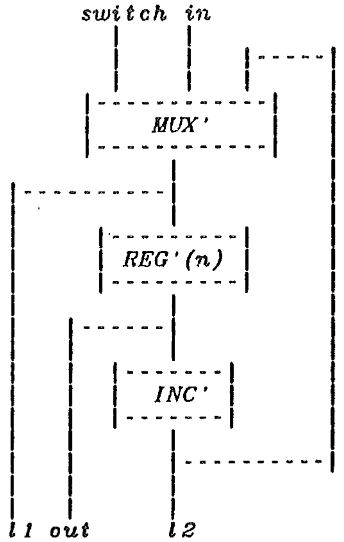
[| *t1* | *t2* | ... | *tn* |]

> Here each *ti* must be a term of OL type ***dev***. The term [|*t1*|...|*tn*|] then denotes the result of connecting together the *ti*'s by joining lines with the same name. The lines of the resulting device are the union of the lines of each of the the component devices *ti*.

For example, if *MUX'*, *REG'* and *INC'* are as above, then:

```
[| MUX' | REG'(n) | INC' |]
```

denotes the device with structure:

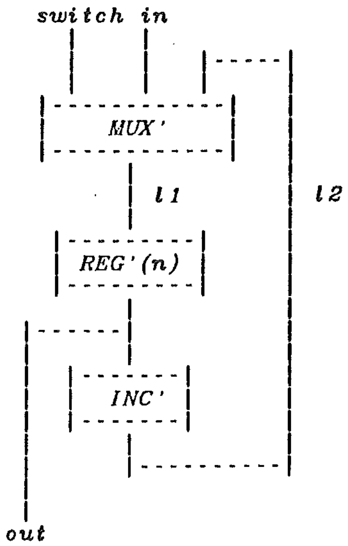


In this diagram the lines *l1* and *l2* are output lines. To represent the diagram in which these lines are internal we need another kind of term.

t hide{*l1*,...,*ln*}

> Here *t* must be a term of OL type ***dev***. Suppose it represents a system specified by a diagram with lines ***l1***,...,***ln***, then *t hide*{*l1*,...,*ln*} represents the system specified by the same diagram except that lines *l1*,...,*ln* are internalized.

For example, the diagram:



Can be represented by the term:

```
[| MUX' | REG'(n) | INC' |] hide{l1,l2}
```

One can also explain the effect of hiding in terms of behaviour equations. Suppose *COUNT_IMP1* is like *COUNT_IMP* (as described above) except that lines *l1* and *l2* are no longer internal, i.e.:

```
COUNT_IMP1 (n) ==  dev{switch,in,out,l1,l2}.
                      {l1=(switch->in|l2),out=n,l2=out+1};
                      COUNT_IMP1(l1)
```

Then if we define:

```
COUNT_IMP2(n) == COUNT_IMP1(n) hide{l1,l2}
```

One can show (using inference rules described in Appendix 3) that:

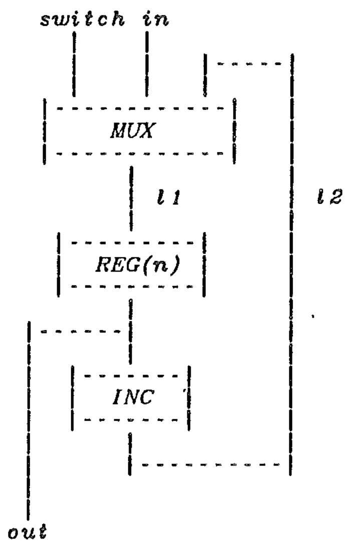
```
COUNT_IMP2 (n) ==  dev{switch,in,out}.
                      {l1=(switch->in|l2), out=n, l2=out+1};
                      COUNT_IMP2(l1)
```

Notice that in this behaviour equation lines *l1* and *l2* are internal, whereas in the equation for *COUNT_IMP1* they are output lines.

It is not necessary to introduce the constants *MUX'*, *REG'* and *INC'*. One can simply write:

```
[| MUX     rn[i1=in;i2=l2;o=l1]
 | REG(n)  rn[i=l1;o=out]
 | INC     rn[i=out;o=l2]  |] hide{l1,l2}
```

When diagraming such terms we will draw in explicit names to indicate how the lines of the devices have been renamed. For example, the term above will be drawn as:



Note that *MUX*, *REG* and *INC* (as defined by behaviour equations above) have different line names to those in this diagram. For example, *MUX* was defined to have lines *i1*, *i2* and *o* instead of *in*, *l2* and *l1*. This illustrates our convention of using the names in diagrams to indicate renaming.

The final kind of term in LSM is used when several "microcycles" of an implementation are used to achieve one "macrocycle" of a specification. For example, the specification of a processor might define a "virtual device" which takes one cycle to execute a machine instruction. An implementation might use several microinstructions (and hence several microcycles) to fetch, decode and execute a single machine instruction. To express the correctness of such an implementation one must somehow say that sequences of cycles in the implementation correspond to a single cycle in the specification. We describe an example like this in [Gordon5], but here we consider something a bit simpler:

```
          i1      i2
          |       |
 |-----------------|
 |                 |
 | MULT_IMP(m,n,t) |
 |                 |
 |-----------------|
          |       |
          |       |
        done      o
```

We want to specify this device to have the following behaviour:

If

1. state variable *t* is initially *T* and
2. numbers *x* and *y* are held on lines *i1* and *i2* and
3. the device is clocked until a state is reached in which line *done* has value *T*, then in this "done-state" the value on line *o* will be the product $x*y$

To express this in LSM we first define a "virtual device" *MULT* which takes just one step to compute the product, and has no *done* line:

```
MULT(m) == dev{i1,i2,o}.{o=m};MULT(i1*i2)
```

We then use a new kind of term to define a behaviour in which sequences of steps of *MULT_IMP* for which the *done* line is *F* are merged to a single step. We then assert this derived behaviour should equal the behaviour of *MULT*. This assertion is:

```
MULT(m) == until done do MULT_IMP(m,n,T)
```

This has the form *t1==t2* where *t2* is a new kind of term defined below:

until l do t

Here *l* must be an OL variable of OL type *bool*, and *t* a term of type *dev* denoting a device with *l* as an output line. The term *until l do t* then denotes a device with the same lines as *t* except *l*. One cycle of this device consists of a sequence of cycles of *t*. To get a single state transition of *until l do t* one repeatedly clocks *t* (holding the inputs constant) until one reaches the first state in which line *l* carries *T*.

The exact meaning of *until l do t* is best grasped from the rule *UNTIL* (described below) and the examples of its use. For example, later we will define *MULT_IMP* so that it has a behaviour satisfying:

```
MULT_IMP(m,n,t) ==
dev{done,i1,i2,o}.
    {o=m,done=t};
    MULT_IMP((t -> (i1=0->0|i2) | (i1=0->0|i2)+m),
             (t -> i1 | n-1),
             ((t -> i1-1 | (n-1)-1)=0 OR (i2=0))
```

Under the assumption that *0-1* equals *0* we will use LCF_LSM to show that:

```
MULT(m) == until done do MULT_IMP(m,n,T)
```

Summary of LSM's Terms for Defining Devices

Sequential Machine:	*dev{x1,...,xm}.{l1=t1,...,ln=tn};t*
Renaming:	*t rn[l1=l1';...;ln=ln']*
Joining:	*[| t1 | t2 | ... | tn |]*
Hiding:	*t hide{l1,...,ln}*
Merging Microcycles:	*until l do t*

In Appendix 1 we list various ML functions for manipulating (e.g. constructing and destructing) these terms.

The Axioms and Inference Rules of LSM

LSM contains all of PPLAMBDA (see [Paulson2]) except those parts pertaining to the theory of complete partial orders. In particular LSM does not have the Scott induction rule, the constants *UU*, *TT*,*FF*, *FIX* and *COND*, or the axioms for partial ordering (i.e. <<), monotonicity, minimality and fixed points. PPLAMBDA's three truth values *TT*, *FF* and *UU* of type ***tr*** are replaced in LSM by *T* and *F* of type ***bool***, and the conditional *COND* is replaced by *SCOND* of type ***bool->*->*->**** (where * can be any type). Appendix 1 contains the various built-in axioms, together with their ML names.

The selection of rules currently included in LSM is rather ad hoc - I have just implemented what seemed needed for the examples I have done. It is hoped that future versions of the system will have a more complete and rationally chosen set. Some theoretical work is needed to isolate a minimal and independent collection of axioms characterizing the class of diagrams representable by terms (see [Milner] for an analysis of a different class of diagrams). Further experimental work is needed to develop a convenient suite of derived rules.

We will introduce various rules using the *COUNT* example discussed above. We give below a session in which we verify that the specification:

```
COUNT(n) == dev{switch,in,out}.{out=n};COUNT(switch->in|n+1)
```

is correctly implemented by the following device:

```
  switch in
    |     |
    |     |    |-----|
  |-------------|    |
  |    MUX      |    |
  |-------------|    |
         |           |
         | l1        | l2
         |           |
    |----------|     |
    |  REG(n)  |     |
    |----------|     |
         |           |
  |------|           |
  |      |           |
  |  |--------|      |
  |  |  INC   |      |
  |  |--------|      |
  |      |           |
  |      |-----------|
  |
 out
```

This can be represented in LSM by introducing a constant *COUNT_IMP* defined by:

```
COUNT_IMP(n) ==   [| MUX      rn[i1=in;i2=l2;o=l1]
                   | REG(n)   rn[i=l1;o=out]
                   | INC      rn[i=out;o=l2]  |]
                  hide{l1,l2}
```

Where the primitives used in this implementation are specified by:

```
MUX    == dev{switch,i1,i2,o}.{o=(switch->i1|i2)};MUX
REG(n) == dev{i,o}.{o=n};REG(i)
INC    == dev{i,o}.{o=(i+1)};INC
```

Formally we wish to prove that:

```
!n. COUNT(n) == COUNT_IMP(n)
```

The sessions that follow are intended to occur in sequence, so that one can assume the effects of earlier sessions persist into later ones. To start with, here is a session in which we create theory called *COUNT* containing the specification of our counter; recall that everything preceded by the prompt symbol # is typed by the user, and everything else by the system.

```
#new_theory'COUNT';;
() : void

#new_constant('COUNT', ":num->dev");;
() : void

#new_axiom
# ('COUNT',
#  "COUNT(n) == dev{switch,in,out}.
#                {out=n}; COUNT(switch->in|n+1)");;
|-"!n.
   COUNT n == dev{switch,in,out}. {out=n}; COUNT(switch -> in | n + 1)"
: thm

#close theory();;
() : void
```

To fully understand this and subsequent sessions, one needs to be familiar with Cambridge LCF. We will try and provide enough commentary so that if you don't know LCF you can still get the gist of what is going on, but inevitably some things will seem unmotived and mysterious. The ML function ***new_theory*** has type ***tok->void***; it creates a new theory whose name is the token given as argument. The function ***new_constant*** has type ***tok#type->void***; it declares an OL constant with the name and OL type passed to it. The function ***new_axiom*** has type ***tok#form->thm***; it takes a name and a formula and makes the formula into an axiom with the given name, the resulting theorem is returned; any free variables in the formula (e.g. n above) are automatically universally quantified. Finally, the function ***close theory*** of type ***void->void*** freezes the current theory; no new constants or axioms can subsequently be added, all one can do is prove theorems and save them.

In the next session we create a theory named *COUNT_IMP* containing the implementation of the counter, including the definition of the primitives *MUX*, *REG* and *INC*. We use the ML function *map* which applies its first argument (which must be a function) to each element of its second argument (which must be a list) and then returns the resulting list. Evaluating the apparently useless expression "*switch:bool,n:num,...*" has the side effect of telling the OL typechecker that *switch* has default OL type *bool*, *n* has default OL type *num*, etc. It must be admitted that a cleaner form of type declaration would be preferable. Exactly when explicit typing is needed in OL is fairly subtle. Beginners are advised to explicitly declare the type of each variable before it is used (after a while one learns when it is safe to allow OL types to be inferred from context; once a variable has a type it will keep it unless explicitly overwritten).

```
#new_theory`COUNT_IMP`;;
() : void

#map new_constant [(`MUX`,":dev");(`REG`,":num->dev");(`INC`,":dev")];;
 [(); (); ()] : void list

#"switch:bool,n:num,i:num,i1:num,i2:num,o:num";;
"switch,n,i,i1,i2,o" : term

#map
# new_axiom
# [(`MUX`, "MUX    == dev{switch,i1,i2,o}.{o=(switch->i1|i2)};MUX");
#   (`REG`, "REG(n) == dev{i,o}.{o=n};REG(i)");
#   (`INC`, "INC    == dev{i,o}.{o=(i+1)};INC")];;
[|-"MUX == dev{switch,i1,i2,o}. {o=(switch -> i1 | i2)}; MUX";
 |-"!n. REG n == dev{i,o}. {o=n}; REG i";
 |-"INC == dev{i,o}. {o=(i + 1)}; INC"]
: thm list

#"l1:num,l2:num";;
"l1,l2" : term

#new_axiom
# (`COUNT_IMP`,
#  "COUNT_IMP(n) ==  [| MUX    rn[i1=in;i2=l2;o=l1]
#                     | REG(n) rn[i=l1;o=out]
#                     | INC    rn[i=out;o=l2] |]
#                    hide{l1:num,l2:num}");;
|-"!n.
    [| MUX rn[i1=in;i2=l2;o=l1]
     | (REG n) rn[i=l1;o=out]
     | INC rn[i=out;o=l2] |]
    hide{l1,l2}"
: thm

#close theory();;

() : void
```

We now begin a session in which we will verify *COUNT_IMP*. First we create a new theory called *COUNT_VER* in which we will prove the desired theorem. This theory must have access to the theories *COUNT* and *COUNT_IMP*; this is achieved by declaring these theories to be parents of *COUNT_VER* using the ML function *new_parent*. The ML function *axiom* fetches an axiom from a theory; it takes the theory name and axiom name as parameters (in that order). After starting the new theory we retrieve the axioms *COUNT* and *COUNT_IMP* from their respective theories and bind the resulting theorems to the ML names *COUNT* and *COUNT_IMP*. Perhaps confusingly we are using these names for three separate purposes: (1) as names of theories, (2) as names for certain axioms on these theories and (3) as ML names for the axioms. Rather than give each primitive a separate ML name it will be more convenient to gather their defining axioms into a list, called *prims*. To generate this list we use the function obtained by partially applying *axiom* to the token '*COUNT_IMP*'; this yields a function of type *tok->thm* which when applied to a token, *tn* say, fetches the axiom named *tn* from the theory *COUNT_IMP*. This function is then *map*ed down a list of axiom names to get the corresponding list of axioms.

```
#new_theory'COUNT_VER';;
() : void

#map new_parent ['COUNT';'COUNT_IMP'];;
Theory COUNT loaded
Theory COUNT_IMP loaded
[(); ()] : void list

#close theory();;
() : void

#let COUNT     = axiom 'COUNT'      'COUNT'
#and COUNT_IMP = axiom 'COUNT_IMP'  'COUNT_IMP';;
COUNT =
|-"!n.
    COUNT n == dev{switch,in,out}. {out=n}; COUNT(switch -> in | n + 1)"
: thm
COUNT_IMP =
|-"!n.
    COUNT_IMP n ==
    [| MUX rn[i1=in;i2=l2;o=l1]
     | (REG n) rn[i=l1;o=out]
     | INC rn[i=out;o=l2] |]
    hide{l1,l2}"
: thm

#let prims = map (axiom 'COUNT_IMP') ['MUX';'REG';'INC'];;
prims =
[|-"MUX == dev{switch,i1,i2,o}. {o=(switch -> i1 | i2)}; MUX";
 |-"!n. REG n == dev{i,o}. {o=n}; REG i";
 |-"INC == dev{i,o}. {o=(i + 1)}; INC"]
: thm list
```

We can now start to prove theorems. First we expand the definitions of the primitives in *COUNT_IMP* using the inference rule *UNFOLD_IMP*. This is an ML function of type *thm list -> thm -> thm*. It takes a list of device definitions and a theorem and replaces each instance of a left hand side of a definition occurring in the theorem by the corresponding right hand side. We bind the theorem resulting from the application of *UNFOLD_IMP* to the ML name *th1*. For a complete description of *UNFOLD_IMP* and all the other inference rules of LSM (except those inherited from PPLAMBDA) see Appendix 3.

```
#let th1 = UNFOLD_IMP prims COUNT_IMP;;
th1 =
|-"COUNT_IMP n ==
   [| (dev{switch,i1,i2,o}. {o=(switch -> i1 | i2)}; MUX)
        rn[i1=in;i2=l2;o=l1]
```

```
     | (dev{i,o}. {o=n}; REG i) rn[i=l1;o=out]
     | (dev{i,o}. {o=(i + 1)}; INC) rn[i=out;o=l2] |]
   hide{l1,l2}"
 : thm
```

The next step is to perform line renaming. For example, we replace:

```
(dev{switch,i1,i2,o}.{o=(switch->i1|i2)};MUX) rn[i1=in;i2=l2;o=l1]
```

by:

```
dev{switch,in,l2,l1}.{l1=(switch->in|l2)};(MUX rn[i1=in;i2=l2;o=l1])
```

This renaming is done using the inference rule *RENAME_LINES*. We bind the resulting theorem to the ML name *th2*.

```
#let th2 = RENAME_LINES th1;;
th2 =
|-"COUNT_IMP n ==
   [| dev
        {switch,in,l2,l1}.
        {l1=(switch -> in | l2)};
        MUX rn[i1=in;i2=l2;o=l1]
    | dev{l1,out}. {out=n}; (REG l1) rn[i=l1;o=out]
    | dev{out,l2}. {l2=(out + 1)}; INC rn[i=out;o=l2] |]
   hide{l1,l2}"
 : thm
```

The next step is the main one. We take the union of the output equations from each of the components to get a single set of equations for all the lines in the composite device. We also gather together the "next-state" terms of the components to get a single term for the whole. The rule *COMBINE_EQUATIONS* does this; for a general description of it see Appendix 3, but the general idea can be got by comparing *th2* above with *th3* below.

```
#let th3 = COMBINE_EQUATIONS th2;;
th3 =
|-"COUNT_IMP n ==
   dev
    {switch,in,out}.
    {l1=(switch -> in | l2),out=n,l2=(out + 1)};
    [| MUX rn[i1=in;i2=l2;o=l1]
     | (REG l1) rn[i=l1;o=out]
     | INC rn[i=out;o=l2] |]
    hide{l1,l2}"
 : thm
```

The "next-state" part of the right hand side of *th3* can we simplified by folding in the definition of *COUNT_IMP*. This is done using the inference rule *FOLD*.

```
#let th4 = FOLD COUNT_IMP th3;;
th4 =
|-"COUNT_IMP n ==
   dev
    {switch,in,out}.
    {l1=(switch -> in | l2),out=n,l2=(out + 1)};
    COUNT_IMP l1"
 : thm
```

Having got a single set of equations for the values on the lines, we can now solve them. We replace:

```
{l1=(switch->in|l2),out=n, l2=(out+1)}
```

by:

```
{l1=(switch->in|n+1),out=n, l2=(n+1)}
```

The inference rule *UNWIND_EQUATIONS* is used to do this. This rule has type *tok list -> thm -> thm*; the token list is a list of line names that one doesn't want unwound, an illustration of when this is useful is the line *bus* in the computer example in [Gordon5].

```
#let th5 = UNWIND_EQUATIONS [] th4;;
th5 =
|-"COUNT_IMP n ==
   dev
    {switch,in,out}.
    {l1=(switch -> in | n + 1),out=n, l2=(n + 1)};
    COUNT_IMP(switch -> in | n + 1)"
: thm
```

Next we notice that the equations for lines *l1* and *l2* are not used anywhere. Since these lines are internal we can delete the equations for them. The inference rule *PRUNE_EQUATIONS* is used to do this.

```
#let th6 = PRUNE_EQUATIONS th5;;
th6 =
|-"COUNT_IMP n ==
   dev{switch,in,out}. {out=n}; COUNT_IMP(switch -> in | n + 1)"
: thm
```

We are now almost done. Notice that the theorem *th6* shows that *COUNT_IMP* satisfies the same equation that was used to define the specification *COUNT*. The inference rule *UNIQUENESS* enables us to deduce that two devices have equal behaviour if they satisfy the same equation. This is valid because behaviour equations have unique solutions [Gordon1]. Using *UNIQUENESS* we prove the theorem expressing the correctness of *COUNT_IMP* with respect to the specification *COUNT*. We then save the resulting theorem on the theory *COUNT_VER* with name *CORRECTNESS*.

```
#let th7 = UNIQUENESS COUNT th6;;
th7 = |-"COUNT n == COUNT_IMP n" : thm

#save_thm(`CORRECTNESS`, th7);;
|-"!n. COUNT n == COUNT_IMP n" : thm
```

Instead of laboriously doing each of the above steps by hand, as above, we can define derived rules in ML which apply each inference rule in turn automatically. The rule *EXPAND_IMP* below derives a behaviour equation from a structural description of an implementation, together with the definitions of the primitives used.

```
#let EXPAND_IMP L prims imp =
# let th1 = UNFOLD_IMP prims imp
# in
# let th2 = RENAME_LINES th1
# in
# let th3 = COMBINE_EQUATIONS th2
# in
# let th4 = FOLD imp th3
# in
# let th5 = UNWIND_EQUATIONS L th4
# in
# PRUNE_EQUATIONS th5;;
EXPAND_IMP = - : (tok list -> thm list -> thm -> thm)

#EXPAND_IMP [] prims COUNT_IMP;;
|-"COUNT_IMP n ==
   dev{switch,in,out}. {out=n}; COUNT_IMP(switch -> in | n + 1)"
: thm
```

Using *EXPAND_IMP* (which is, in fact, built-in to LCF_LSM) we can define another derived rule *VERIFY* which does the whole correctness proof in one step.

```
#let VERIFY prims spec imp = UNIQUENESS spec (EXPAND_IMP [] prims imp);;
VERIFY = - : (thm list -> thm -> thm -> thm)

#VERIFY prims COUNT COUNT_IMP;;
|-"COUNT n == COUNT_IMP n" : thm
```

Although the example just done is trivial, it does illustrate many of the things needed in bigger proofs. The computer example in [Gordon5] shows this.

Two things not illustrated by the *COUNT* example are the use of LCF theorem proving techniques to prove ancillary lemmas (verification conditions), and the inference rule needed to handle cycle merging when one has terms of the form ***until l do t***. To illustrate these we now describe another example - a multiplier. First we will construct a system *MULT_IMP* with behaviour:

```
MULT_IMP(m,n,t) ==
dev{done,i1,i2,o}.
   {o=m,done=t};
   MULT_IMP((t -> (i1=0->0|i2) | (i1=0->0|i2)+m),
            (t -> i1 | n-1),
            ((t -> i1-1 | (n-1)-1)=0 OR (i2=0))
```

The we will show that:

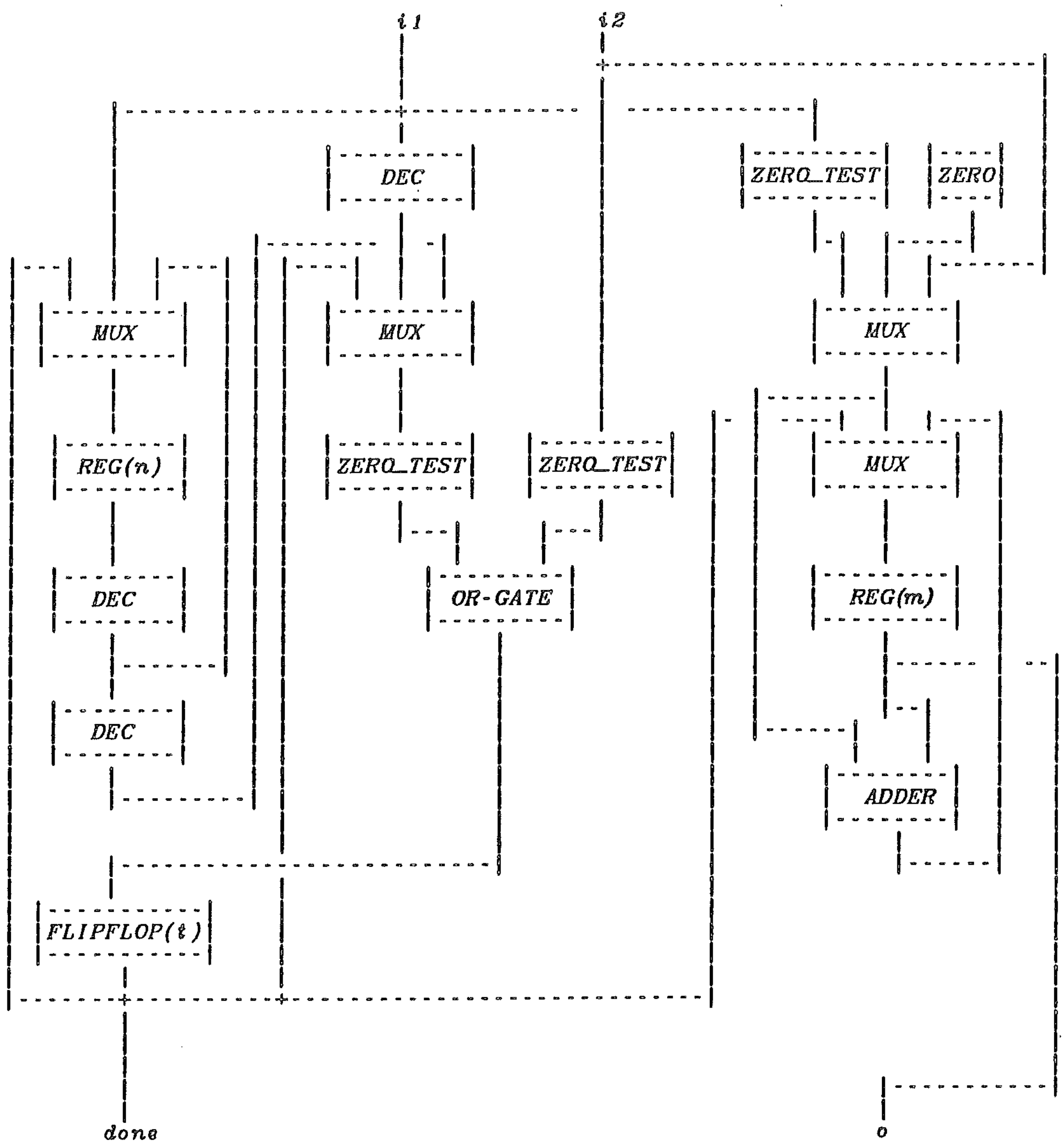
```
!m n. MULT(m) == until done do MULTIMP(m,n,T)
```

where *MULT* is defined by:

```
MULT(m) == dev{i1,i2,o}.{o=m};MULT(i1*i2)
```

Thus one "macrocycle" of *MULT* is implemented by a variable number of microcycles of *MULT_IMP*. (Note that we assume *0-1* is *0*; we could easily modify the implementation to avoid the need for this assumption).

The definition of *MULT_IMP* we use is given by the diagram below (we leave the reader the exercise of working out the line names from the structural description in LSM which we give shortly).



Although this is a simple system, it is not instantly obvious that it correctly does multiplication. As with the previous example we create theories for the specification and implementation (named *MULT* and *MULT_IMP* respectively).

```
#new_theory'MULT';;
() : void

#new_constant('MULT', ":num -> dev");;
() : void
```

```
#"m:num","i1:num","i2:num","o:num";;
"m","i1","i2","o" : (term # term # term # term)

#new_axiom
# (`MULT`,
#   "MULT m == dev{i1,i2,o}.{o=m}; MULT(i1*i2)");;
|-"!m. MULT m == dev{i1,i2,o}. {o=m}; MULT(i1 * i2)" : thm

#close_theory();;
() : void
```

Before defining the implementation of the multiplier in LSM we make a theory containing the definitions of the various primitives used.

```
#new_theory `prims`;;
() : void

#map new_constant [`MUX`          , ":dev";
#                  `REG`          , ":num->dev";
#                  `FLIPFLOP`     , ":bool->dev";
#                  `DEC`          , ":dev";
#                  `ADDER`        , ":dev";
#                  `ZERO_TEST`    , ":dev";
#                  `OR_GATE`      , ":dev";
#                  `ZERO`         , ":dev"];;
[(); (); (); (); (); (); (); ()] : void list

#"i1:num","i2:num";;
"i1","i2" : (term # term)

# map
#  new_axiom
#  [(`MUX`        , "MUX == dev{switch,i1,i2,o}.{o=(switch->i1|i2)};MUX");
#   (`REG`        , "REG(n) == dev{i,o}.{o=n};REG(i)");
#   (`FLIPFLOP`   , "FLIPFLOP(t) == dev{i,o}.{o=t};FLIPFLOP(i)");
#   (`DEC`        , "DEC == dev{i,o}.{o=(i-1)};DEC");
#   (`ADDER`      , "ADDER == dev{i1,i2,o}.{o=(i1+i2)};ADDER");
#   (`ZERO_TEST`  , "ZERO_TEST == dev{i,o}.{o=(i=0)};ZERO_TEST");
#   (`OR_GATE`    , "OR_GATE == dev{i1,i2,o}.{o=(i1 OR i2)};OR_GATE");
#   (`ZERO`       , "ZERO == dev{o}.{o=0}; ZERO")];;
[|-"MUX == dev{switch,i1,i2,o}. {o=(switch -> i1 | i2)}; MUX";
 |-"!n. REG n == dev{i,o}. {o=n}; REG i";
 |-"!t. FLIPFLOP t == dev{i,o}. {o=t}; FLIPFLOP i";
 |-"DEC == dev{i,o}. {o=(i - 1)}; DEC";
 |-"ADDER == dev{i1,i2,o}. {o=(i1 + i2)}; ADDER"
 |-"ZERO_TEST == dev{i,o}. {o=(i = 0)}; ZERO_TEST";
 |-"OR_GATE == dev{i1,i2,o}. {o=(i1 OR i2)}; OR_GATE";
 |-"ZERO == dev{o}. {o=0}; ZERO"]
: thm list

#close_theory();;
() : void
```

Next we define a theory *MULT_IMP* with *prims* as a parent. Note that we use a slightly more compact way of assigning default OL types to variables.

```
#new_theory`MULT_IMP`;;
() : void

#new_parent`prims`;;
Theory prims loaded
() : void
```

```
#new_constant(`MULT_IMP` , ":num # num # bool -> dev");;
() : void

#"[done;b1;b2;b3;b4]:bool list",
#"[l1;l2;l3;l4;l5;l6;l7;l8;l9;l10]:num list";;
 "[done;b1;b2;b3;b4]","[l1;l2;l3;l4;l5;l6;l7;l8;l9;l10]" : (term # term)

#new_axiom
# (`MULT_IMP`,
#  "MULT_IMP(m,n,t) ==
#    [| MUX           rn[switch=done;i1=l9;i2=l8;o=l7]
#     | REG(m)        rn[i=l7]
#     | ADDER         rn[i1=l9;i2=o;o=l8]
#     | DEC           rn[i=i1;o=l6]
#     | MUX           rn[switch=done;i1=l6;i2=l4;o=l5]
#     | MUX           rn[switch=done;i2=l3;o=l1]
#     | REG(n)        rn[i=l1;o=l2]
#     | DEC           rn[i=l2;o=l3]
#     | DEC           rn[i=l3;o=l4]
#     | ZERO          rn[o=l10]
#     | MUX           rn[switch=b4;i1=l10;o=l9]
#     | ZERO_TEST     rn[i=i1;o=b4]
#     | ZERO_TEST     rn[i=l5;o=b1]
#     | ZERO_TEST     rn[i=i2;o=b2]
#     | OR_GATE       rn[i1=b1;i2=b2;o=b3]
#     | FLIPFLOP(t)   rn[i=b3;o=done] |]
#   hide {b1,b2,b3,b4,l1,l2,l3,l4,l5,l6,l7,l8,l9,l10}");;
|-"!m n t.
    MULT_IMP(m,n,t) ==
    [| MUX rn[switch=done;i1=l9;i2=l8;o=l7]
     | (REG m) rn[i=l7]
     | ADDER rn[i1=l9;i2=o;o=l8]
     | DEC rn[i=i1;o=l6]
     | MUX rn[switch=done;i1=l6;i2=l4;o=l5]
     | MUX rn[switch=done;i2=l3;o=l1]
     | (REG n) rn[i=l1;o=l2]
     | DEC rn[i=l2;o=l3]
     | DEC rn[i=l3;o=l4]
     | ZERO rn[o=l10]
     | MUX rn[switch=b4;i1=l10;o=l9]
     | ZERO_TEST rn[i=i1;o=b4]
     | ZERO_TEST rn[i=l5;o=b1]
     | ZERO_TEST rn[i=i2;o=b2]
     | OR_GATE rn[i1=b1;i2=b2;o=b3]
     | (FLIPFLOP t) rn[i=b3;o=done] |]
    hide{b1,b2,b3,b4,l1,l2,l3,l4,l5,l6,l7,l8,l9,l10}"
: thm

#close theory();;
() : void
```

Now we can use the derived rule *EXPAND_IMP* defined above as the first stage of the verification of *MULT_IMP*.

```
#new_theory`MULT_VER`;;
() : void

#map new_parent [`MULT`;`MULT_IMP`];;
Theory MULT loaded
Theory MULT_IMP loaded
[(); ()] : void list

#save_thm
# (`MULT_IMP_EXPAND`,
#   EXPAND_IMP
#    []
#    (map
#      (axiom `prims`)
#      [`MUX`;`REG`;`FLIPFLOP`;`DEC`;`ADDER`;
#       `ZERO_TEST`;`ZERO`;`OR_GATE`])
```

```
#     (axiom 'MULT_IMP' 'MULT_IMP'));;
|-"!m n t.
    MULT_IMP(m,n,t) ==
    dev
     {done,o,i1,i2}.
     {o=m,done=t};
    MULT_IMP
     ((t -> (i1 = 0 -> 0 | i2) | (i1 = 0 -> 0 | i2) + m),
      (t -> i1 | n - 1),
      ((t -> i1 - 1 | (n - 1) - 1) = 0) OR (i2 = 0))"
: thm
```

Notice that we have not yet closed the theory *MULT_VER*; we will want to add another axiom later. First must use the rule of inference called *UNTIL*. This represents the meaning of terms ***until l do t*** which we described above. The general idea is this: suppose we have a behaviour equation:

```
D(a1,...,ap) == dev{l,x1,...,xm}.
                   {l=t,l1=t1,...,ln=tn};
                   D(u1,...,up)
```

If we regard each step of *D* as a microcycle of ***until l do D(a1,...,ap)***, then the rule *UNTIL* allows us to derive a behaviour equation for cycles of the *until*-term. Ignoring some details (which are discussed below), the equation yielded by *UNTIL* is:

```
until l do D(a1,...,ap) == dev{x1,...,xm}.
                              {l1=t1,...,ln=tn};
                              until l do D(f(u1,...,up))
```

where intuitively ***f(u1,...,up)*** is the first state following ***(a1,...,ap)*** in which line ***l*** has value *T*. Notice that the done-line ***l*** is not a line of the *until*-device.

The next-state function ***f*** must satisfy the following equation:

```
f(a1,...,ap) == (t -> (a1,...,ap) | f(u1,...,up))
```

Here ***a1***, ... , ***ap*** are the state variables of *D* and ***u1***, ... , ***up*** are the expressions from the behaviour equation for *D* that specify the next state. The equation above uniquely defines ***f***. The rule *UNTIL* generates this equation from the behaviour equation for *D*, and so constructs the definition of the next-state function for ***until l do D(a1,...,ap)***.

Note how the equation for ***f*** simply iterates the state transformation:

(a1,...,ap) –> (u1,...,up)

until a state is reached in which the value on the output line ***l*** (which is given by ***t***) is *T*. In other words ***f(a1,...,ap)*** is the first state after *(a1,...,ap)* in which the value of ***t*** is *T*.

The discussion just given is slightly oversimplified in that it ignores the requirement that during sequences of microcycles which make up a macrocycle, the inputs are assumed constant. To reflect this we must make the inputs parameters to the next-state function *f*.

We define an input line of a behaviour term:

```
dev{l,x1,...,xm}.{l=t,l1=t1,...,ln=tn};D(u1,...,up)
```

to be a line *xi* that occurs free in *t*, or in one of the *ti*, or in one of the *ui*. For example, the input lines of:

```
dev{done,i1,i2,o}.
   {o=m,done=t};
   MULT_IMP((t -> (i1=0->0|i2) | (i1=0->0|i2)+m),
            (t -> i1 | n-1),
            ((t -> i1-1 | (n-1)-)1=0 OR (i2=0))
```

are *i1* and *i2*.

To express the requirement that inputs are held constant during microcycles, we modify the definition of the next-state function for *until l do D(a1,...,ap)* by changing the definition of *f* to:

```
f(x1,...,xq,a1,...,ap) == (t -> (a1,...,ap) | f(x1,...,xq,u1,...,up))
```

where *x1*, ... , *xq* are the input lines. We call this equation the next-state equation for *until l do D(a1,...,ap)*. Notice how this reflects the constancy of the input lines during microcycles.

We can now describe the LSM rule *UNTIL*. It takes a theorem of the form:

```
D(a1,...,ap) == dev{l,x1,...,xm}.
                {l=t,l1=t1,...,ln=tn};
                D(u1,...,up)
```

and produces a theorem of the form:

```
f(x1,...,xq,a1,...,ap) == (t -> (a1,...,ap) | f(x1,...,xq,u1,...,up))
==>
until l do D(a1,...,ap) == dev{x1,...,xm}.
                           {l1=t1,...,ln=tn};
                           until l do D(f(x1,...,xq,u1,...,up))
```

This says that if *f* satisfies the equation preceding the ==> then *until l do D(a1,...,ap)* satisfies the behaviour equation following it. The *f* here is a variable, though the equation it satisfies uniquely determines it. The user will usually introduce an OL constant for this uniquely defined function, *F* say, so that he can instantiate *f* to *F* and apply modus ponens (the PPLAMBDA rule *MP*) to derive:

```
until l do D(a1,...,ap) == dev{x1,...,xm}.
                           {l1=t1,...,ln=tn};
                           until l do D(F(x1,...,xq,u1,...,up))
```

Let us do this for our multiplier. First we must use the *UNTIL* rule. The user must give as parameters to *UNTIL* the names of the next-state function and the done-line. Thus the ML type of *UNTIL* is *tok->tok->thm->thm*. The first argument token is the name of the next-state function. The second token is the name of the done-line. We will call the next-state function-variable for the multiplier *mult_fn*; the done-line is *done*.

```
#save_thm
# (`MULT_IMP_UNTIL`,
#  UNTIL `mult_fn` `done` (theorem `MULT_VER` `MULT_IMP_EXPAND`));;
|-"!mult_fn.
    (!i1 i2 m n t.
      mult_fn(i1,i2,m,n,t) ==
       (t ->
        (m,n,t) |
        mult_fn
         (i1,
          i2,
          (t -> (i1 = 0 -> 0 | i2) | (i1 = 0 -> 0 | i2) + m),
          (t -> i1 | n - 1),
          ((t -> i1 - 1 | (n - 1) - 1) = 0) OR (i2 = 0)))) ==>
    (!m n t.
      until done do MULT_IMP(m,n,t) ==
      dev
       {o,i1,i2}.
       {o=m};
       until
        done
        do
        MULT_IMP
        (mult_fn
         (i1,
          i2,
          (t -> (i1 = 0 -> 0 | i2) | (i1 = 0 -> 0 | i2) + m),
          (t -> i1 | n - 1),
          ((t -> i1 - 1 | (n - 1) - 1) = 0) OR (i2 = 0))))"
: thm
```

Notice how *UNTIL* produces a theorem of the form *fm1 ==> fm2*, where *fm1* is the recursive definition of the next-state function (called *mult_fn* in the example above), and *fm2* is the behaviour equation of the *until*-term (*until done do MULT_IMP(m,n,t)* in the example above).

Let us now introduce a constant *MULT_FN* defined by the next-state equation just generated. Since we have not yet closed theory MULT_VER we can include it there.

```
#new_constant(`MULT_FN`, ":num#num#num#num#bool -> num#num#bool");;
() : void

#new_axiom
# (`MULT_FN`,
#   "!i1 i2 m n t.
#      MULT_FN(i1,i2,m,n,t) ==
#       (t ->
#        (m,n,t) |
#        MULT_FN
#          (i1,
#           i2,
#           (t -> (i1 = 0 -> 0 | i2) | (i1 = 0 -> 0 | i2) + m),
#           (t -> i1 | n - 1),
#           ((t -> i1 - 1 | (n - 1) - 1) = 0) OR (i2 = 0)))");;
|-"!i1 i2 m n t.
    MULT_FN(i1,i2,m,n,t) ==
```

```
(t ->
 (m,n,t) |
  MULT_FN
   (i1,
    i2,
    (t -> (i1 = 0 -> 0 | i2) | (i1 = 0 -> 0 | i2) + m),
    (t -> i1 | n - 1),
    ((t -> i1 - 1 | (n - 1) - 1) = 0) OR (i2 = 0)))"
: thm

#close theory();;
() : void
```

We now need a lemma relating the recursively defined function *MULT_FN* to multiplication. The required lemma can be proved in LCF using mathematical induction and various arithmetical properties (including *0-1==0*). We will not give the proof here, but just make the needed theorem an assumption as follows:

```
#let lemma =
  ASSUME
   "!i1 i2.
     MULT_FN(i1,i2,(i1=0->0|i2),i1,((i1-1)=0)OR(i2=0)) ==
       (i1*i2, (((i1-1)=0)OR(i2=0)->i1|1), T)";;
lemma =
.|-"!i1 i2.
     MULT_FN(i1,i2,(i1 = 0 -> 0 | i2),i1,((i1 - 1) = 0) OR (i2 = 0)) ==
      i1 * i2,(((i1 - 1) = 0) OR (i2 = 0) -> i1 | 1),T"
: thm
```

The rule *ASSUME* of ML type *form->thm* takes a formula *fm* to a theorem *fm* |- *fm*.

Next we specialize the variable *mult_fn* in *MULT_IMP_UNTIL* to the constant *MULT_FN* we have just introduced. Then finally we do modus ponens with the specialized theorem and the definition of *MULT_FN*.

```
#let th1 =
# MP (SPEC "MULT_FN" MULT_IMP_UNTIL) (axiom `MULT_VER` `MULT_FN`);;
th1 =
|-"!m n t.
    until done do MULT_IMP(m,n,t) ==
    dev
     {o,i1,i2},
     {o=m};
     until
      done
      do
      MULT_IMP
      (MULT_FN
       (i1,
        i2,
        (t -> (i1 = 0 -> 0 | i2) | (i1 = 0 -> 0 | i2) + m),
        (t -> i1 | n - 1),
        ((t -> i1 - 1 | (n - 1) - 1) = 0) OR (i2 = 0)))"
: thm
```

Now we specialize *t* to *T* and simplify the result using the OL axiom *BOOL_COND_CLAUSES* and *lemma*. The axiom *BOOL_COND_CLAUSES* is:

```
!x y. (T -> x | y) == x /\ (F -> x | y) == y
```

The simplifier is a derived inference rule called *REWRITE_RULE*; it takes a list of theorems and uses them to simplify a given theorem (see [Paulson1] for details).

```
#let th2 =
# REWRITE_RULE
#   [BOOL_COND_CLAUSES; lemma]
#   (SPEC "T" (SPEC "n" (SPEC "m" th1)));;
th2 =
.|-"until done do MULT_IMP(m,n,T) ==
    dev
     {o,i1,i2}.
     {o=m};
     until
      done
      do
      MULT_IMP(i1 * i2,(((i1 - 1) = 0) OR (i2 = 0) -> i1 | 1),T)"
```

The last step is to use the inference rule *UNIQUENESS* to prove that the specification *MULT* (which is an axiom on the theory *MULT* set up above) is correctly implemented by *MULT_IMP*.

```
#let th3 = UNIQUENESS (axiom 'MULT' 'MULT') th2;;
th3 = .|-"MULT m == until done do MULT_IMP(m,n,T)" : thm
```

Then we can save *th3* as the theorem *CORRECTNESS*.

```
#save_thm('CORRECTNESS', th3);;
.|-"!m n.
    (!i1 i2.
      MULT_FN(i1,i2,(i1 = 0 -> 0 | i2),i1,((i1 - 1) = 0) OR (i2 = 0)) ==
      i1 * i2,(((i1 - 1) = 0) OR (i2 = 0) -> i1 | 1),T)  ==>
    MULT m == until done do MULT_IMP(m,n,T)"
: thm
```

When a theorem is saved in a theory all its assumptions are automatically discharged. Hence in the example above, the assumed lemma appears as the antecedent of an implication whose consequent is the formula we have just proved. If we subsequently prove this lemma as a theorem, we can remove it as an antecedent using modus ponens. We shall not do this here as it is routine LCF.

This concludes the correctness proof of the multiplier example, and also our introduction to LCF_LSM. Although the examples we have used are trivial, I hope they are suggestive of what might be done. A much bigger example is given in [Gordon5].

Conclusions

Is it practical to prove real hardware correct?

The LCF_LSM verification system has been built to try and answer this question.

Based on the case studies done so far (primarily [Gordon5]), I claim that:

1. Non-trivial hardware can be proved correct *using existing techniques*.

2. Machine assistance is *essential* for such proofs.

Further evidence of this is provided by the work of John Herbert, a research student in the Computer Laboratory. He has proved correct one of the integrated circuits used in the Cambridge Fast Ring (a 100 megabit per second local network). The chip that has been verified translates between a serial data path and a sequence of 8-bit packets; it is implemented in ECL technology.

Future research

It is hoped to apply LCF_LSM (or its descendants) to examples supplied by industrial collaborators. I am actively seeking partners for this enterprise.

An industrially supported student (Inder Dhingra) is developing techniques specialized to the cMOS technology.

I am currently simplifying LCF_LSM using ideas from Temporal Logic. This work is being done in collaboration with my colleague Dr. Ben Moszkowski (who has a research position in the Computer Laboratory). It is also inspired by suggestions made by Edmund Ronald and Professor Tony Hoare. The idea is to dump LSM and just work with LCF. For example, instead of describing the counter specification and implementation with functions *COUNT* and *COUNT_IMP*, where:

```
COUNT(n) == dev{switch,in,out}. {out=n}; COUNT(switch -> in | n+1)

COUNT_IMP(n) ==  [| MUX     rn[i1=in;i2=l2;o=l1]
                  | REG(n)  rn[i=l1;o=out]
                  | INC     rn[i=out;o=l2]  |]
                 hide{l1,l2}
```

we use predicates *COUNT_PRED* and *COUNT_IMP_PRED* defined by:

```
COUNT_PRED(n,i,switch,o) <=> (o == n) /\ DEL(n,(switch->i|n+1))

COUNT_IMP_PRED(n,i,switch,o) <=> ?l1 l2.
                                   MUX_PRED(switch,i,l2,l1) /\
                                   REG_PRED(n,l1,o)         /\
                                   INC_PRED(o,l2)
```

where the primitives are specified by:

```
MUX_PRED(switch,i1,i2,o) <=> o == (switch->i1|i2)

REG_PRED(n,i,o) <=> (o == n) /\ DEL(n,i)

INC_PRED(i,o) <=> o == i+1
```

This approach appears to have the following nice properties:

1. Formal descriptions are in "pure logic"; the extra CCS-like terms are not needed. For example, hiding is represented by existential quantification and parallel composition (i.e. [| ... | ... | ... |]) by conjunction.

2. Verification can be done using standard inference rules of logic. The *ad-hoc* rules in Appendix 3 are not needed.

3. Combinational devices are not forced to have a next-state part. Sequential behaviour is specified with delay operators. (The predicate *DEL* is the unit-delay of temporal logic [Moszkowski].)

4. Bidirectional devices have a more natural specification.

5. Devices can be specified to have "pure delay" instead of state. This turned out to be useful for the ECL chip verification mentioned above.

Various case studies are being done to see if these nice properties really hold. For example, the cMOS work is being conducted in this framework.

Appendix 1: Types, terms and constants of LSM

LSM has the following types of arity 0 built in: *bool* (booleans), *num* (non-negative integers), *word*n (n-bit words), *tri-word*n (n-bit tri-state words), *memm*_n (m by n memories) and *dev* (devices). The type *list* of arity 1 is also built in.

The current implementation of LCF_LSM requires the user to explicitly declare the sizes of words and memories that he will use. Two ML functions are provided for this purpose:

declare_word_widths : int list -> void
declare_memories : (int # int) list -> void

In the computer example described in [Gordon3], words of sizes 2, 3, 5, 13, 16 and 30, and memories of sizes 5 by 30 and 13 by 16 are used. The declarations needed for this are:

declare_word_widths[*2;3;5;13;16;30*]
declare_memories[*(5,30);(13,16)*]

These declarations call *new_type* for the appropriate types of the form *word*n, *tri-word*n and *memm*_n. They should be done in a theory that is a parent to all the theories that use *word* and *mem* types. The theory *values* has this role in the computer example.

In addition to the usual PPLAMBDA terms, LSM also has the following terms of OL type *dev*:

Sequential Machine:	*dev{x1,...,xm}.{l1=t1,...,ln=tn};t*
Renaming:	*t rn[l1=l1';...;ln=ln']*
Joining:	[| *t1* | *t2* | ... | *tn* |]
Hiding:	*t hide{l1,...,ln}*
Merging Microcycles:	*until l do t*

Here *x1*, ... ,*xm*, *l*, *l1*, ... , *ln* are OL variables used to represent lines.

For each of these terms there are ML syntax functions with prefixes *mk_*, *is_* and *dest_* as described in [Paulson2]. The constructors and destructors are inverses; we just describe the former.

```
mk_dev(["x1"; ... ;"xm"], ["l1","t1"; ... ;"ln","tn"], "t")
-->
"dev{x1, ... ,xm}.{l1=t1, ... , ln=tn}; t"
```

```
mk_join["t1"; ... ;"tn"]
-->
"[| t1 | ... | tn |]"

mk_hide("t",["l1"; ... ;"ln"])
-->
"t hide{l1, ... ln}"

mk_rename("t",[l1,l1'; ... ;lm,lm'])
-->
"t rn [l1=l1'; ... ;lm=lm']"

mk_until("l","t")
-->
"until l do t"
```

An ML-like syntax for lists is also available in LSM: If *t1*, ..., *tn* all have OL type *ty*, then the term [*t1;...;tn*] has OL type *ty list*. The empty list of OL type **list* is denoted by [].

LSM also has terms: *let x=t1 in t2*. These are just alternative syntax for: *(\x.t2)t1* (where *\x.t2* is the PPLAMBDA notation for lambda-expressions).

The LSM syntax for conditionals is *(t->t1|t2)* where *t* is a term with OL type *bool* and *t1* and *t2* have the same OL type. Note that this differs from the PPLAMBDA (and ML) syntax of *(t=>t1|t2)* where *t* must have type *tr* (the PPLAMBDA type of three-valued truthvalues).

The built-in (non-functional) constants of LSM are:

T, F	The truth-values of OL type *bool*
0, 1, 2, ...	Numbers of OL type *num*
#b1...bn	Words of OL type *word*n (*bi* is either *0* or *1*).
[]	The empty list of OL type **list*.

The following infixed binary operators are built in:

=	equality. OL type **#*->bool*
+	addition. OL type *num#num->num*
-	subtraction. OL type *num#num->num*
*	multiplication. OL type *num#num->num*
OR	disjunction. OL type *bool#bool->bool*
AND	conjunction. OL type *bool#bool->bool*
NOR	negated disjunction. OL type *bool#bool->bool*
XOR	exclusive OR. OL type *bool#bool->bool*
EQV	logical equivalence. OL type *bool#bool->bool*

In addition there are the following (non-infixed) functions:

SUC	successor function. OL type num->num
PRE	predecessor function. OL type num->num
NOT	negation. OL type bool->bool
CONS	list cons. OL type *->* list->* list
HD	head. OL type * list->*
TL	tail. OL type * list->* list
NULL	null test. OL type * list->bool
EL	nth element of a list. OL type num->* list->*
SEG	sublist of a list. OL type num#num->* list->* list
V	number denoted by a bit list. OL type bool list->num

The ML function *declare_word_widths* creates the following functions for each width declared:

*VAL*n	number denoted by a word. OL type wordn->num
*WORD*n	word representing a number. OL type num->wordn
*BITS*n	list of bits in a word. OL type wordn->bool list
*NOT*n	complement a word. OL type wordn->wordn
*MK_TRI*n	make a tri-state value. OL type wordn->tri_wordn
*DEST_TRI*n	convert a tri-state value to a word. OL type tri_wordn->wordn

For each n there is also a constant *FLOAT*n of OL type tri_wordn to represent the value on a floating bus of width n. The following infixes are also defined for each width:

*OR*n	bit-by-bit OR. OL type wordn#wordn->wordn
*AND*n	bit-by-bit AND. OL type wordn#wordn->wordn
*U*n	combining tri-state values. OL type tri_wordn#tri_wordn->tri_wordn

The ML function *declare_memories* creates the following functions for each memory size (m,n):

*STORE*m	store a value. OL type wordm->wordn->memm_n->memm_n
*FETCH*m	fetch a value. OL type memm_n->wordm->wordn

Note that only one type of memory with a given address size is possible. For example, one can't have both *mem13_8* and *mem13_16*. This restriction is purely to keep the names of the fetch and store functions short; it may be relaxed in the future.

Appendix 2: Axioms and Built-in Theorems

Little attempt has been made to give a well rounded set of axioms and built-in theorems. The ones listed below are motived by the examples that have been done (notably the computer example in [Gordon4]). I expect that future studies will expose the need for more.

The following axioms are included in LSM. We give their ML name followed by the corresponding formula.

```
EQ                      !x:*.!y:*.  x = y == T  <=>  x == y"

BOOL_COND_CLAUSES       !x:*.!y:*.  (T -> x | y) == x /\  (F -> x | y) == y

BOOL_CASES              !b:bool.  b==T \/ b==F

BOOL_EQ_DISTINCT        ~T==F /\ ~F==T

NOT                     NOT F == T /\ NOT T == F

OR                      F OR F == F /\
                        F OR T == T /\
                        T OR F == T /\
                        T OR T == T

AND                     !b1 b2.  b1 AND b2 == NOT((NOT b1) OR (NOT b2))

NOR                     !b1 b2.  b1 NOR b2 == NOT(b1 OR b2)

XOR                     !b1 b2.  b1 XOR b2 == (b1 OR b2) AND (NOT(b1 AND b2))

EQV                     !b1 b2.
                          b1 EQV b2 == (b1 AND b2) OR ((NOT b1) AND (NOT b2))
```

The following consequences of these axioms are available:

```
NEG_F                   !x y.  ~ x == y   ==>   x = y == F

F_NEG                   !x y.  x = y == F   ==>   ~ x == y

NEG_EQ                  !x y.  ~ x == y   <=>   x = y == F

EQ_T                    !b.  b = T == b

NEG_T_F                 !b.  ~ b == T   ==>   b == F

F_T                     !b.  b == F   ==>   ~ b == T

NEG_F_T                 !b.  ~ b == F   ==>   b == T

T_F                     !b.  b == T   ==>   ~ b == F

BOOL_EQ                 !b.  (~ b == T   <=>   b == F)  /\
                             (~ b == F   <=>   b == T)

FN_COND                 !f t x y.  f(t -> x | y) == (t -> f x | f y)

TRIV_COND               !t x.  (t -> x | x) == x

COND_PAIR               !t x1 x2 y1 y2.
                          (t -> (x1,x2) | (y1,y2)) ==
                            (t -> x1 | y1),(t -> x2 | y2)
```

```
AND_TABLE        F AND F == F ∧
                 F AND T == F ∧
                 T AND F == F ∧
                 T AND T == T

NOR_TABLE        F NOR F == T ∧
                 F NOR T == F ∧
                 T NOR F == F ∧
                 T NOR T == F

XOR_TABLE        F XOR F == F ∧
                 F XOR T == T ∧
                 T XOR F == T ∧
                 T XOR T == F

EQV_TABLE        F EQV F == T ∧
                 F EQV T == F ∧
                 T EQV F == F ∧
                 T EQV T == T

OR_CLAUSES       !t. t OR T == T ∧
                     T OR t == T ∧
                     t OR F == t ∧
                     F OR t == t

AND_CLAUSES      !t. t AND T == t ∧
                     T AND t == t ∧
                     t AND F == F ∧
                     F AND t == F

DEMORGAN_OR      !t1 t2. NOT(t1 OR t2) == NOT t1 AND NOT t2

DEMORGAN_AND     !t1 t2. NOT(t1 AND t2) == NOT t1 OR NOT t2

NOT_NOT          !t. NOT(NOT t) == t
```

At present there are no built-in axioms or theorems for numbers, words, memories or lists. In Appendix 3 we describe some inference rules which enable certain constant expressions at these types to be simplified (for example *2+3* can be simplified to *5*).

Appendix 3: Rules of Inference

LSM contains inference rules for reasoning about the various terms of type *dev*. We give the ML type of each rule followed by a schematic description of its effect.

COMPOSE : term -> thm

```
COMPOSE
 "[| dev X1.EQ1;N1 | ... | dev Xn.EQn;Nn |] hide L"
 -->
 "|- [| dev X1.EQ1,N1 | ... | dev Xn.EQn,Nn |] hide L ==
     dev (X1u ... uXn)\L.EQ1u ... uEQn;([| N1 | ... | Nn|] hide L)"
```

Here *X1*u ... u*Xn* and *EQ1*u ... u*EQN* denotes the union of sets of lines *X1*, ... , *XN* and equations *EQ1*, ... , *EQN* respectively; *(X1u ... uXn)\L* denotes the union of the lines minus the lines in *L* (thus \ is set subtraction, or complement).

COMPOSE fails if:

1. *L* contains an *l* which is an input line (i.e. occurs in an *Ni* or the rhs of an equation in an *EQi* and is not the lhs of an equation).
2. There exist distinct *i* and *j* such that *EQi* contains *x=ti* and *EQj* contains *x=tj*.

UNFOLD_IMP : thm list -> thm -> thm

```
UNFOLD_IMP
 ["|-t1==u1"; ... ;"|-tm==um"]
 "|- t == [| t1' rn[R1] | ... | tn' rn[Rn] |] hide L"
 -->
 "|- t == [| u1' rn[R1] | ... | un' rn[Rn] |] hide L"
```

Where if *ti* has an instance *ti'* then *ui'* is the corresponding instance of *ui* (i.e. *ui'* is got from *ui* using the same substitution that yields *ti'* when applied to *ti*).

UNFOLD_DEF : thm list -> thm -> thm

```
UNFOLD_DEF
 ["|-t1==u1"; ... ;"|-tm==um"]
 "|- t == ti' rn[R]"
 -->
 "t == ui' rn[R]"
```

Where, as above, *ui'* is the instance of *ui* obtained by matching *ti'* to *ti* ("|-*ti*==*ui*" must not have any assumptions).

FOLD : thm -> thm -> thm

```
FOLD
  "|- t == u"
  "|- tm == devX,EQS,u'"
  -->
  "|- tm == devX,EQS,t'"
```

Where *t'* is the instance of *t* corresponding to the way *u'* is an instance of *u*.

RENAME : term -> thm

```
RENAME
  "(devX.EQS,NXT)rn[R]"
  -->
  "|- (devX.EQS,NXT)rn[R] == devX'.EQS';(NXT' rn[R])"
```

Where *X'*, *EQS'* and *NXT'* are derived from *X*, *EQS* and *NXT* respectively by renaming according to *R*. RENAME fails if:

1. *R* has the form [...*x*=*a*;...*y*=*a*;...].
2. "*x*:*ty1*" is in *X* and "*x*:*ty2*" is a lhs of an equation in *R* and *ty1*, *ty2* are distinct.

UNWIND : tok list -> term -> thm

```
UNWIND
  L
  "devX.{o1=t1, ... ,on=tn};NXT"
  -->
  "|- devX.{o1=t1, ... ,on=tn};NXT == devX.{o1=t1', ... ,on=tn'};NXT'"
```

Where *t1'*, ... ,*tn'* and *D'* are got from *t1*, ... ,*tn* and *D* by unwinding those equations whose lhs is not in *L*.

PRUNE : term -> thm

```
PRUNE
  "devX.EQS;NXT"
  -->
  "|- devX.EQS;NXT == devX.EQS';NXT"
```

Where *EQS'* is the subset of *EQS* obtained by removing equations whose lhs (i) is not an output line, and (ii) does not occur in *D* or the rhs of an equation used to define variables whose values are observable.

UNIQUENESS : thm -> thm -> thm

```
UNIQUENESS
  "|- D1(a1,...,ap) == devX1.EQ1;D(A1,...,Ap)"
  "|- D2(b1,...,bq) == devX2.EQ2;D(B1,...,Bq)"
  -->
  "|- D1(a1,...,ap) == D2(b1,...,bq)"
```

Where:

1. For each *i*: if *ai* is not a variable then *ai* equals *Ai* and if *bi* is not a variable then *bi* equals *Bi*.
2. *X1* denotes the same set of lines as *X2*, and *EQ1* the same set of equations as *EQ2*.
3. The subset of pairs *(ai,Ai)* where *ai* is a variable occurring free in either the right hand side of an equation in *EQ1* or in an *Ai*, equals the subset of pairs *(bi,Bi)* where *bi* is a variable occurring free in either the right hand side of an equation in *EQ2* or in a *Bi*.

The easiest way to understand these conditions is to look back at the *COUNT* and *MULT* examples and see what they mean there.

UNTIL : tok -> tok -> thm -> thm

```
UNTIL
 `f`
 `l`
 "|- D(x1,...,xm) == devX.
                          {l=t,o1=t1,...,on=tn,l1=u1,...,lp=up};
                          D(E1,...Ep)"
 -->
 "!f.
   (!i1 ... ir x1 ... xm.
     f(i1,...,ir,x1,...,xm) ==
      let l1=u1 in
              .
              .
              .
      let lp=up in
       (t -> (x1,...,xm) | f(i1,...,ir,E1,...,Ep)))
  ==>
  until l do D(x1,...,xm) ==
  dev(X-{l}). {o1=t1,...,on=tn,l1=u1,...,lp=up};
              until l do D(f(i1,...,ir,E1,...,Ep))"
```

Where *i1,...,ir* are the input lines occurring in *t,E1,...,Ep,u1,...,up* and *l1,...,lp* are the internal lines (i.e. *li* is not in *X*). *UNTIL* fails if:

1. *l* is not in *X*, or *l* does not have type **bool**,
2. *x1,...,xm* are not distinct, or some *xi* is not a variable,
3. one of *E1,...,Ep* has a free variable which isn't in *X* or
4. any of *u1,...,up* have free variables which arn't in *X* or *{x1,...,xm}* (thus the local equations *{l1=u1,...,lp=up}* must not be recursive).

The rules that follow are not primitive (i.e. there are definable in terms of the rules above) but are included for convenience.

EXPAND_DEF : thm list -> thm -> thm

```
EXPAND_DEF
 [...;"|- D == devX.EQS;D'";...]
 "|- D1 == D rn[R]"
 -->
 "|- D1 == devX'.EQS';D1'"
```

Where *X'*, *EQS'* are got from *X*, *EQS* by renaming according to *R*. This is useful for creating copies of generic devices with different line names.

RENAME_LINES : thm -> thm

```
RENAME_LINES
 "|- t == [| (devX1.EQS1;NXT1)rn[R1]
             .
             .
             .
           | (devXn.EQSn;NXTn)rn[Rn]  |] hide L"
 -->
 "|- t == [| devX1'.EQS1';(NXT1' rn[R1])
             .
             .
             .
           | devXn'.EQSn',(NXTn' rn[Rn])  |] hide L"
```

Where the primed components are got from the corresponding unprimed ones by renaming.

COMBINE_EQUATIONS : thm -> thm

```
COMBINE_EQUATIONS
 "|- t == [| dev X1.EQ1;N1 | ... | dev Xn.EQn;Nn |] hide L"
 -->
 "|- t == dev (X1u ... uXn)\L.EQ1u ... uEQn;([| N1 | ... | Nn|] hide L)"
```

This generates a behaviour equation for an implementation from the behaviour equations for the components (u denotes set union and \ set subtraction).

UNWIND_EQUATIONS : tok list -> thm -> thm

```
UNWIND_EQUATIONS
 L
 "|- t == devX.EQS;NXT"
 -->
 "|- t == devX.EQS';NXT'"
```

Where *EQS'* and *D'* are got from *EQS* and *D* by unwinding on *L*.

PRUNE_EQUATIONS : thm -> thm

```
PRUNE_EQUATIONS
 "|- t == devX.EQS;NXT"
 -->
 "|- t == devX.EQS';NXT"
```

Where *EQS'* is got form *EQS* by pruning.

The ML functions *EXPAND_IMP* and *VERIFY* are defined by:

```
let EXPAND_IMP L prims imp =
 let th1 = UNFOLD_IMP prims imp
 in
 let th2 = RENAME_LINES th1 ? th1
 in
 let th3 = COMBINE_EQUATIONS th2
 in
 let th4 = FOLD imp th3
 in
 let th5 = UNWIND_EQUATIONS L th4
 in
 PRUNE_EQUATIONS th5;;

let VERIFY prims spec imp =
 UNIQUENESS spec (EXPAND_IMP nil prims imp);;
```

The next collection of rules enable a certain amount of evaluation of constants to be done. For each kind of evaluation we provide a simple rule and a formula conversion (details of conversions are in [Paulson1]). Only the simple rules are used in the examples described in this report The conversions will be useful when we come to define tactics for LSM (see [Gordon et. al.] and [Paulson] for a description of goal-directed proof using tactics).

We list below a name and the corresponding evaluation, for example:

```
FOO   t1 --> t1'
      t2 --> t2'
```

this means that there are two ML functions:

```
FOO_RULE  : thm -> thm
FOO_FCONV : form -> thm
```

The rule *FOO_RULE* will take a theorem *th* to *th'* and *FOO_FCONV* will take a formula *fm* to the theorem *fm* <=> *fm'*, where *th'* and *fm'* are got from *th* and *fm* respectively by replacing all subterms of the form *t1* and *t2* by the corresponding ones of the form *t1'* and *t2'*. We use *t*, *t1*, *t2*, ... to range over OL terms.

```
BITS     BITSw #b1...bw          --> [t1;...;tw]
```

(where *ti* is *T* if *bi* is *1* and *F* otherwise)

```
ADD      m+n                     --> r
```

(where *r* is the numeral denoting the sum of *m* and *n*)

```
DIF      m-n                     --> r
```

(where *r* is the numeral denoting the difference of *m* and *n*)

```
EQ       x=x                     --> T
         x=y                     --> F
```

(where *x* and *y* are distinct constants)

```
EL      EL i [tn; ... . ; t0]       --> ti

WORD    WORDw n                     --> #b1...bw
```

(where #$b1...bw$ is the w-bit binary representation of n)

```
VAL     VALw #b1...bw               --> n
```

(where n is the number denoted by #$b1...bw$)

```
V       V [t1;...;tm]               --> n
```

(where n is the number denoted by $[t1;...;tm]$)

```
SEQ     SEQ(i,j)[tm;...;t0]         --> [tj;...;ti]

AND     #a1...aw AND #b1...bw       --> #c1...cw
```

(where ci is the value of the conjunction of ai and bi)

```
OR      #a1...aw OR #b1...bw        --> #c1...cw
```

(where ci is the value of the disjunction of ai and bi)

```
NOT     NOT #a1...aw                --> #b1...bw
```

(where bi is the negation of ai)

```
COND      (T->t1|t2)                  --> t1
          (F->t1|t2)                  --> t2

U         FLOATw Uw t                 --> t
               t  Uw FLOATw           --> t

TRI       DEST TRIw(MK TRIw t)        --> t

bool       t AND T                    --> t
           t AND F                    --> F
           T AND t                    --> t
           F AND t                    --> F
           t OR  T                    --> T
           t OR  F                    --> t
           T OR  t                    --> T
           F OR  t                    --> t
             NOT T                    --> F
             NOT F                    --> T
```

For example, *ADD_RULE* would reduce |- t == *(2+3)+4* to |- t == *9*, and *bool_FCONV* would map the formula *NOT T == x OR F* to the theorem:

```
NOT T == x OR F <=> F == x
```

References

[Ayres]
R. Ayres. *VLSI: silicon compilation and the art of automatic microchip design*. Prentice-Hall, 1983

[Barrow]
H. Barrow. *Proving the Correctness of Digital Hardware Designs*. Available from: Dr. Harry Barrow, Fairchild Laboratory for Artificial Intelligence Research, 4001 Miranda Ave., Palo Alto, CA 94304.

[Dembinski & Budkowski]
P. Dembinski and S. Budkowski, *Verification, Design and Description Oriented Microprogramming Language*. Proc. 4th EUROMICRO Symp. on Microprocessing and Microprogramming, Munich, Oct. 17-19, 1978.

[Gordon et. al.]
M. Gordon, R. Milner and C. Wadsworth. *Edinburgh LCF: A mechanised logic of computation*. Lecture Notes in Computer Science Number 78, Springer-Verlag, 1979.

[Gordon1]
M. Gordon. *The Denotational Semantics of Sequential Machines*. Information Processing Letters, Volume 10, Number 1, 1980

[Gordon2]
M. Gordon. *A Model of register Transfer Systems with Applications to Microcode and VLSI Correctness*. Internal Report CSR-82-81, Dept. of Computer Science, University of Edinburgh, 1981

[Gordon3]
M. Gordon. *A Very Simple Model of Sequential Behaviour of nMOS*. In *VLSI 81* (ed. J. Gray), Academic Press, 1981.

[Gordon4]
M. Gordon. *Representing a Logic in the LCF Metalanguage*. In *Tools and Notions for Program Construction* (ed. D. Neel), Cambridge University Press, 1982.

[Gordon5]
M. Gordon. *Proving a Computer Correct with the LCF_LSM Hardware Verification System*. University of Cambridge computer laboratory technical report Number 42, 1983.

[Hanna]
K. Hanna. *Overview of the VERITAS Project*. Available from: Dr. F. K. Hanna, Electronics Laboratory, University of Kent at Canterbury, Canterbury, England.

[Milne]
G. Milne. *CIRCAL: A Calculus for Circuit Description*. Internal Report CSR-122-82, Dept. of Computer Science, University of Edinburgh, 1982.

[Milner]
R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science Number 92, Springer-Verlag, 1980.
[Moszkowski]
B. Moszkowski. *A Temporal Logic for Multi-Level Reasoning about Hardware*. In the proceedings of the IFIP Sixth International Conference on Computer Hardware Description Languages and Their Applications, Pittsburgh, U.S.A., 1983.
[Paulson1]
L. Paulson *A Higher Order Implementation of Rewriting*. To be published in Science of Computer Programming, 1983.
[Paulson2]
L. Paulson. *The Revised Logic PPLAMBDA: A Reference Manual*. University of Cambridge computer laboratory technical report Number 36, 1983.
[Paulson3]
L. Paulson. *Tactics and Tacticals in Cambridge LCF*. University of Cambridge computer laboratory technical report Number 39, 1983.

Acknowledgements

I have had useful conversations with Harry Barrow, John Herbert, George Milne, Robert Milne, Ben Moszkowski, Larry Paulson and Edmund Ronald. Tom Melham made valuable comments on a first draft of this paper.