How to program in Handel

(based on the Handel hardware compiler, version H161)

Mike Spivey and Ian Page and Wayne Luk

Last revised by Ian Page, August 1995

Contents

1 Introduction

This is a practical guide to writing programs in Ian Page's Handel language and compiling them into hardware netlists.

Handel is an imperative programming language similar to a subset of occam. It provides variables with assignment and the conventional control structures of sequencing, conditionals and loops, as well as parallel composition and CSP-like commands for communication between processes and with the outside world.

A Handel program is not usually compiled into machine code, but into a collection of hardware gates and flip-flops. Each variable in the program corresponds to a hardware register, and each expression to a combinational circuit that computes its value from the register contents. The control structure of the program becomes a network of control logic that activates the registers at the proper time, and synchronises communication between the parallel processes. The result is an implementation of the Handel program which uses genuine parallelism for maximum performance. Typical applications include parallelisation of the inner loops of compute-intensive programs, construction of special-purpose processors, and replacements for random logic in embedded systems.

A convenient way of realising the circuit that is output by the compiler is to load it into a dynamically reconfigurable Field-Programmable Gate Array chip, such as the ones manufactured by Xilinx and installed on the HARP board we have constructed. To this end, the compiler produces its output in the correct format for input to the *place and route* stages of the Xilinx design software.

2 How to use the compiler

Unlike most compilers, this one does not take input in the form of a source program text. Instead, the input to the compiler is an SML data object that represents the source program as an abstract syntax tree. To use the compiler, you write an SML expression that evaluates to a syntax tree which represents a Handel program. You can then apply the compiler to the syntax tree to produce the corresponding hardware netlist. There are a number of good references that can be consulted for details of the SML language [1, 2, 3, 4]

To help you write down the syntax tree for your program, we provide a collection of SML functions and operators that build syntax trees for constructs in the Handel language. The abstract syntax approach has several advantages:

- It is often convenient to write a fragment of SML that generates the syntax tree for your Handel program, instead of writing the Handel program directly. SML is used here in its intended role, as a *meta-language*. This can be especially useful when the program has some regular, even if complex, structure. Also, the compiler can be much smaller when meta-language features are supported instead of a larger number of language features.

  It is of course possible to use program generators with text-based programming languages. However, the SML meta-programming approach results in a framework which is fully type-checked. Additionally there is no tedious stage of writing the generated program as a text file and reading it into the compiler.

- It is easy for to experiment with modifications to the language and compiler without having to modify and maintain a parser. It is also easy for the user to build translators from higher-level languages into Handel.

These advantages are important both for the user of Handel and for the implementor; but there a few disadvantages that show up when things go wrong. Errors in a Handel program can show themselves at a number of different stages:

- The SML system may find a syntax error in the expression you write for the syntax tree of the Handel program.
- The SML system may find a type error in the expression.
- The Handel compiler may discover errors in the program.
- The program may contain no detectable errors, but may describe the wrong piece of hardware (ie. you wrote the wrong program correctly).

The error messages you get at each stage are sometimes hard to relate to the source program you intended to write.

3 Program structure

A simple Handel program consists of some SML declarations, followed by an SML expression that builds the abstract syntax tree for the program. The compiler expects to find the program to be compiled in the global variable `prog`. The SML declarations introduce the resources (such as variables and channels) used in the program and give them SML identifiers to facilitate the construction of the program proper.

The abstract syntax tree itself can be constructed using the `Handel` function which takes three parameters:

- the declaration of the program's external interface,
- the declaration of the internal variables and channels,
- the program body.

So a program typically consists of four parts: the list of SML declarations, followed by the three-part syntax tree. Here is a small example that conforms to this pattern:

```
val X   = Int 1 "X"
val Out = Chan 1 "Out" ;

prog := Handel (

  [ CPC_SimOut Out ],

  [ X ],

  Seq [
    X := C 0,
    While (TRUE,
      Seq [
        Out !! X,
        X := X + C 1 ] ) ]

);
```

This program uses a single variable `X` and a single channel `Chan`, each 1 bit wide. The channel is connected to the outside world, and the program outputs the values 0 and 1 alternately, continuing to output values as long as the environment is prepared to accept them.

The first part of this program introduces identifiers `X` and `Out` as SML values so that they can be used in the rest of the program. The compiler provides a set of functions that create variables, channels, etc. Each returns a value that represents the resource in the Handel program. For example, the SML declaration

```
val R = Int 8 "R"
```

calls the compiler function `Int` to create an SML value representing a variable named '`R`' of width 8 bits. The function returns a value of type `EXPR` that will be used later in writing the program body; this value is bound to the SML identifier `R`.

The main reason for having these SML declarations is so that identifiers can be used to construct the Handel program proper. The string passed to `Int` is the *print name* of the resource, and is used to identify the variable in compiler error messages and simulator output. It may also be used to name wires in the final netlist, thus aiding back-tracing. By convention, these print names are spelled the same as the SML identifiers used in writing the program body, but you can make them different if you like. Because the names are simply related to the names of the final (flattened) netlist, there may be constraints from the downstream CAD software to be taken into account. It is usually preferable to choose *unique, alphanumeric* [1] print names for each variable and channel used in a program.

Another function used in this part of the program is `Chan`. Like `Int`, this takes a width and a string, but it creates a channel that can be used with input and output commands rather than expressions and assignments. Both `Int` and `Chan` are overloaded with a variant that takes a list of strings as the second argument and will return the corresponding list of `EXPR` values. Neither `Int` nor `Chan` cause the compiler to generate any hardware to represent the register or channel: this is done later when the compiler works on the abstract syntax tree.

The remaining parts of the program are contained in one assignment to the global variable *prog*:

```
prog := Handel (
  <external protocols>,
  <internal resources>,
  <program body>
);
```

The external interface is declared by a list of *protocols*. Each protocol is obtained by applying a *protocol converter* to one of the internal channels created earlier. The protocol converter describes how the internal channel is made to interact with the outside world; `CPC_SimOut` is a channel protocol converter which the simulator implements as handshaken output to the screen. Section 7 describes some of the protocol converters currently implemented.

The internal resources are declared by a simple list, which will normally contain exactly those integers and channels that have been created in the first part but not used as external resources.

The last part is the command which is the body of the program. The syntax of Handel expressions is described in section 4 and the syntax of commands is described in section 5.

[1] when using the Xilinx macro package, names with a leading "_" are reserved

4 Expression syntax

Table 1 shows the operators that can be used to build Handel expressions. For each of these operators, there is an SML operator that takes arguments of type `EXPR` and delivers a result of the same type. Some of these are overloaded meanings of the usual arithmetic operators; others are specific to Handel programs.

As an example, if `X` and `Y` are variables that have been declared as decribed in Section 3, then we can write the Handel expression

```
X + Y
```

This is uses the `+` operator between Handel expressions: from the SML point of view, both the variables `X` and `Y` and the whole expression have type `EXPR`, and the `+` operator simply constructs a little piece of syntax tree. When this piece of tree is submitted to the Handel compiler, it builds an adder into the output circuit.

Each expression in Handel has a *width*, and the compiler demands that widths agree in assignment statements, that addition takes two arguments of the same width and produces a results of the same width again, and so on. The widths of arguments and results are shown in the table. A boolean expression is simply any expression of width 1, and the results of comparison operators have this width.

The primitive items in expressions are variables and integer constants. The integer constant 37 can be written either as `C 37` (with an unspecified width) or as `Const(37,8)` (with an explicit width of 8 bits). For the most part, the compiler can deduce from the program what widths should be given to constants: for example, in the expression `X + C 1`, the constant 1 must have the same width as `X`. But sometimes the compiler fails to deduce the width, and you need to specify it yourself.

Some of the operators need more explanation.

- The expression `Cond (b,t,f)` evaluates the Boolean expression `b`, and delivers either the expression `t`, or `f`, depending on whether `b` = 1 or 0, respectively. The expressions `t`, and `f` should have the same width.

- The expression `E <- k` (pronounced "`E` take `k`") returns the least significant `k` bits of `E`, and `E \\ k` (pronounced "`E` drop `k`") gives all but the least significant `k` bits. In both cases, `k` must be a compile-time constant, and in fact it is an SML integer, rather than a Handel expression. The width of this expression depends on the value of k and the width of `E`..

- The expression `E1 ^ E2` concatenates the bit-strings that are the values of `E1` and `E2`, so its width is the sum of the widths of `E1` and `E2`. The bits from `E1` form the *least* significant part of the result and those from `E2` form the *most* significant part.

Handel operators (such as `<-`) that treat expressions as lists of bits identify bit 0 with the first element of the list. So the `LSbit` operator could be defined at the SML level by:

```
fun LSbit expr  =  expr <- 1
```

(In fact, this is a definition that is built into the Handel compiler). The user is encouraged to define other useful SML functions to aid construction of programs: they have the effect of open subroutines, expanded at the point of each call.

Unlike some programming languages, Handel does not have implicit 'coercions' that widen and narrow the values of expressions; instead, the conversion of (for example) an 8-bit unsigned value `X` to 16 bits must be written explicitly as `X ^ ZEROS 8`. Here `ZEROS n` is just an abbreviation for `Const(0,n)`. Truncation of a 16-bit value to 8 bits can be achieved by writing `X <- 8`, taking the least significant 8 bits of `X`.

		Widths of args	result
Unary prefix operators			
`DECODE`	2^x	n	2^n
`~`	bitwise NOT	n	n
`ABS`	absolute value (2's comp)	n	n
`LSbit`	leftmost bit	n	1
`MSbit`	rightmost bit	n	1
Arithmetic operators			
`+`	addition	n, n	n
`-`	subtraction	n, n	n
`*`	multiplication	m, n	$m+n$
Comparison operators			
`==`	equality	n, n	1
`!=`	inequality	n, n	1
`|>|`	signed greater than (also `|>=|`, etc.)	n, n	1
`$>$`	unsigned greater than (also `$>=$`, etc.)	n, n	1
Bit operators			
`^`	concatenation of bit-strings	n, m	$n+m$
`<-`	take	n, const k	k
`\\`	drop	n, const k	$n-k$
`EXOR`	bitwise exclusive-OR	n, n	n
`/\`	bitwise AND	n, n	n
`\/`	bitwise OR	n, n	n
`BIT`	bit selection	n, const	1
Conditional Operator			
`Cond (b,t,f)`		$1, n, n$	n

Table 1: Handel operators

5 Command syntax

We now describe the syntax of Handel commands. Each construct is illustrated by an example and an SML declaration of the functions used to build a tree for the construct. The functions we describe here are not necessarily the actual constructors for the syntax tree, so there is no guarantee that they can be used for pattern-matching in functions that process Handel programs: in fact, the set of constructors may change from time to time, but the functions described here will continue to exist.

In what follows, we are quite precise about the number of clock cycles taken by an execution of each kind of command. There are two reasons for this: first, to help you to estimate the cost of each construct in execution time and calculate how fast your program will run; and second, because it is sometimes expedient to exploit the synchronous nature of the implementation. This involves writing programs that disobey the rules of Occam about simultaneous access by parallel programs to shared variables or channels. The rules are there to make asynchronous or other implementations possible, and knowing that our implementations use synchronous concurrency lets us predict the behaviour of programs that do not obey them. However, this is not a very good idea except in special circumstances, and it requires careful reasoning.

5.1 Skip, Delay, and Stop

Format:

```
Skip
```

or

```
Delay
```

or

```
Stop
```

SML function:

```
Skip, Delay, Stop : STAT
```

The `Skip` command does nothing and takes no time. The `Delay` command also does nothing, but takes one clock cycle to do it. The `Stop` command has only one purpose, which is to refuse to terminate.

5.2 Assignment

Assignment comes in two flavours: the familiar single assignment that gives a new value to a single variable, and a simultaneous assignment that changes the values of several variables at the same instant. Both are expressed using the overloaded `:=` operator of SML.

Format:

```
var := exp
```

or

```
[var1, var2, ..., varn] := [exp1, exp2, ..., expn]
```

SML functions:

```
op := : EXPR * EXPR -> STAT
op := : EXPR list * EXPR list -> STAT
```

The ordinary SML assignment operator is another overloaded meaning of the `:=` operator. It can be distinguished from the tree-constructing forms described here by the fact that its left argument has a `ref` type. Although the left argument of the tree-constructing `:=` operator is a Handel variable, it is represented by a constant fragment of tree.

There is also the primitive constructor `Assign`, which takes a list of (var, expression) pairs; this may be more convenient for use in a program generator:

```
Assign: (EXPR * EXPR) list -> STAT
```

Each kind of assignment command takes a single cycle; all the LHS variables are updated simultaneously with the values of the RHS expressions. The results are unpredictable if two assignments to the same variable are executed simultaneously: this can happen if two assignments to the same variable are executed simultaneously by two parallel processes. Unless you are certain of the correctness of programs that share variables, it is best to follow the rule of Occam that forbids such assignments.

5.3 Communication

Format:

```
chan ?? var
chan !! exp
```

SML functions:

```
op ?? : EXPR * EXPR -> STAT
op !! : EXPR * EXPR -> STAT
```

These commands perform input and output on a channel. The other end of the channel may be another parallel process in the Handel program, or it may be connected to the outside world through a protocol converter. SML does not allow the meaning of certain symbols (such as `!`) to be redefined or overloaded: that is why we have not used the more conventional occam/CSP `?` and `!` operators to denote input and output. This is also the reason behind the choice of names for certain other Handel functions.

The compiler generates hand-shaking hardware that delays input and output commands on a channel until one of each kind is ready. After this, the value is transferred in a single cycle. There is no time overhead for synchronisation, so an output command to a channel will complete in a single cycle, provided another process is already waiting to input on that channel or arrives at an input command at the same instant.

It is illegal to have two input commands or two output commands for the same channel active at once; the rules of Occam forbid sharing of channels by parallel processes in this way, and it is best to obey them except in special circumstances.

Channels of zero width are possible when synchronisation only is required; 'Any' is special EXPR which can be used with any input or output command when the value to be transmitted is irrelevant.

5.4 Parallel composition

Format:

```
Par [ cmd1, cmd2, ..., cmdn ]
```

SML function:

```
Par: STAT list -> STAT
```

When a parallel composition is executed, all the individual commands start at the same instant; they execute with genuine parallelism, and the whole parallel composistion terminates when all the individual commands have terminated.

No clock cycles are taken by the `Par` construct itself, so the time taken by the whole parallel combination is the maximum of the times taken by the individual commands.

There is also an infix version of `Par` with this format:

```
cmd1 || cmd2
```

SML function:

```
op || : STAT * STAT -> STAT
```

The compiler arranges things so that repeated binary composition produces the same hardware as a single use of `Par`, so

```
cmd1 || cmd2 || ... || cmdn
```

is just as efficient as the `Par` form shown above.

5.5 Sequential composition

Format:

```
Seq [ cmd1, cmd2, ..., cmdn ]
```

SML function:

```
Seq: STAT list -> STAT
```

The sequential composition of two or more commands is executed by starting the first command immediately, and starting each successive command when the preceding one finishes. The whole composistion finishes when the last command has finished. The time taken to execute the combination is the sum of the times taken by the individual commands.

There is also an infix form of `Seq`:

```
cmd1 $$ cmd2
```

SML function:

```
op $$ : STAT * STAT -> STAT
```

No special measures need be taken in the compiler to make

```
cmd1 $$ cmd2 $$ ... $$ cmdn
```

equivalent in efficiency to the `Seq` form above, because the two forms naturally compile into the same hardware.

5.6 Conditional

Format:

```
If (cond, then-part, else-part)
```

SML function:

```
If: EXPR * STAT * STAT -> STAT
```

The `If` command uses the value of the Boolean (i.e. one-bit) condition to decide whether to execute the then-part command or the else-part command. It takes the same time as whichever command is selected; no additional time is taken for the decision. The else part of the conditional is not optional, but can be a simple `Skip` command.

5.7 Loops

Format:

```
While(cond, body)
Until(cond, body)
```

SML function:

```
While: EXPR * STAT -> STAT
Until: EXPR * STAT -> STAT
```

The `While` command executes its body repeatedly until the condition is false. The time taken is the sum of the times for each execution of the body – again, there is no overhead [2] for the test. If the condition is false initially, the `While` command takes zero time. The `Until` command is similar except that the test is done *after* each execution of the loop body, and repetition continues until the test is true, so that the loop body is always executed at least once.

5.8 Selection

Format:

```
Case (switch,
      [([label], body), ([label1, label2, ...], body), ... ],
      optional_default)
```

where the optional default is

```
Default statement
```

or

```
NoDefault
```

ML function:

```
Case: EXPR * (int list * STAT) list * OPT_STAT -> STAT

OPT_STAT:  Default   of  STAT
         |  NoDefault
```

A `Case` statement is executed by evaluating the switch expression, and executing whichever arm of the statement has a matching label. If no label matches, the default part is executed. If `NoDefault` is specified and the label matches are not exhaustive, then `Stop` is assumed. Each arm has a *list* of labels, so that an arm can handle more than one value of the switch expression.

Because the implementation of the `Case` statement involves a decoder, it is a good idea to arrange the width of the switch expression to be as small as possible; otherwise, large amounts of almost useless hardware are generated. If the value of an 8-bit variable X is known to be at most 5, say, it is better to say

```
Case (X <- 3, [...], ...)
```

than simply use X itself as the switch expression. Only the low-order 3 bits of X are needed to discriminate the cases, and the compiler will not have to build a 256-output decoder.

Another way of achieving similar behaviour is to use nested `If` commands where the user has explicit control over the selection mechanism. Nested conditional expressions might also be a suitable alternative to the case statement in some circumstances.

[2] advanced users may wish to know about an implementation issue concerning loops with zero-time bodies which is covered in Appendix B

5.9 Alternation

Format:

```
PriAlt [(cond, guard, body), ...]
```

ML function:

```
PriAlt: (EXPR * STAT * STAT) list -> STAT
```

In a `PriAlt` command, all the guard commands should be input commands, of the form `chan ?? var`. All the conditions should be booleans (one bit wide). The command is executed as follows: it waits until there is at least one arm in which both the condition is true and the guard is ready for communication. The textually first of these active arms is chosen for execution: the communication takes place in one clock cycle, then the body of the arm is executed. Execution of the whole construct finishes when the chosen body finishes. The time taken for the whole command is one clock cycle more than the time taken to execute the chosen body.

6 On-chip RAM and ROM

The compiler can generate small RAM and ROM structures by building them from gates and flip-flops. They are expensive structures when implemented on today's FPGAs and only make sense when kept very small.

On-chip RAMs and ROMs are created by the SML functions

```
Ram: string * int * int -> (EXPR list -> EXPR)
Rom: string * int * int list -> (EXPR list -> EXPR)
```

The expression `Ram (name, D, A)` creates a RAM with data width D and address width A: it has 2^A locations each D bits wide. The expression `Rom (name, D, data)` creates a ROM containing the list of integers `data`. Its data width is D, and its address width is just large enough to address all the elements of `data`, that is $\lceil \log_2 N \rceil$ where N is the length of `data`.

Both functions return a result of type `EXPR list -> EXPR`. The idea is that if A is a RAM or ROM, you should write '`A[i]`' to access the i'th element of A. SML parses this expression as the function A applied to the singleton list `[i]`, hence the type. Despite the fact that a list of expressions appears as the subscript, we do not provide multi-dimensional arrays. This notation is a bit of a hack, but it was done to make Handel array accesses look a little more familiar since SML doesn't support new bracketing operators.

The following program is an example of the use of the `Rom` structure. It defines a ROM of width 7 with 12 elements and sequentially reads the elements into `X`. The example also shows the use of SML functions to derive by calculation some parameters for the program. In particular, the programmer specifies the set of values to go into the Rom structure (`vals`) and the desired data width of the values (`DW`); SML then calculates (at compile time) the minimum width of the index register for the Rom and the final value for the loop test.

```
val vals   = [2,4,6,8,10,12,111,3,5,7,9,127]
val DW     = 7

val n      = length vals
val w_indx = ilog2 n
val X      = Int DW "X"
val table  = Rom ("table", DW, vals)
val indx   = Int w_indx "indx" ;

prog := Handel (
```

```
  [],
  [ indx, X, table[] ],
  Until (indx == C (n mod (2**w_indx)),
      [X, indx] := [table [indx], indx + C 1]
      )
  );
```

The following program is a similar example which fills a RAM with values and then reads them all back continuously:

```
val DW    =  4;
val AW    =  3;

val DATA  = Int DW "DATA"
val ADDR  = Int AW "ADDR"
val MEM   = Ram ("MEM", DW, AW);
val Ch1   = Chan DW "Ch1";

prog := Handel (
  [ CPC_SimOut Ch1],
  [ ADDR, DATA, MEM[] ],
  Seq [ Until (ADDR == C 0,
          [ADDR, MEM [ADDR]] :=
             [ADDR + C 1, C ((2**DW)-1) - (ADDR ^ Const (0, DW-AW))]
          ),
        ADDR := C 0,
        While (TRUE,
          [ADDR, DATA] := [ADDR + C 1, MEM [ADDR]]  $$  Ch1 !! DATA )
      ]
  );
```

6.1 Restrictions on using ROM and RAM structures

An important consideration is that a RAM or ROM can only be used once in each clock cycle, because the memories constructed are single-ported; multi-port RAMs and ROMs would far too expensive to implement on current FPGAs. This means that even a statement as apparently innocent as

```
A[i] := A[j]
```

is not allowed, because it involves two accesses to the RAM `A` in the same clock cycle. What you have to write instead is something like

```
Seq [ TMP := A[j], A[i] := TMP ]
```

Things are made even more tricky than this, because expressions in different places can be evaluated simultaneously: for example, the test of a while loop is evaluated simultaneously with expressions in the first command of its body, so the loop

```
While (A[i] > 0,
  Seq [
    SUM := SUM + A[i],
    i := i + C 1])
```

is not allowed either: it needs a `Delay` statement before the assignment to `SUM`. The compiler tries to issue warnings about this kind of thing, but it will sometimes give warnings about perfectly sound programs, and it is the user's responsibility to validate all cases. This situation will be eased when full scope and usage rules for the language are built into the compiler (one day!).

6.2 An alternative to the RAM structure

For many applications that involve systolic computation or replicated parallel processes, RAMs are not the thing to use, because (as we've seen) they allow access to only one location at a time. What is needed in these applications is an array of variables that allows simultaneous access to all of them. To create such an array you can build your own SML functions, or you can use the built-in function

```
Ints: int -> int -> string -> EXPR list
```

like this:

```
val Pipeline = Ints 10 DW "X"
```

This declaration creates a row of 10 variables, each `DW` bits wide, binding `Pipeline` to a list of them. Their print names are `X_0`, `X_1`, etc. . For convenience, you can now define

```
fun X[n] = nth (Pipeline, n)
```

and write `X[3]` in your program for the fourth `X` variable (umbering starts from zero, of course). Note that is only a compile-time indexing operation as no list of registers will exist in the final hardware! SML can now be used very conveniently to build such things as a Handel parallel assignment in which all the elements can be moved simultaneously down the pipeline:

```
Assign (zip (tl Pipeline, Pipeline))
```

7 Channel protocol converters

Some of the channels created in the first part of the program may be connected to the world outside the Handel program. To do this, it is necessary to specify the conversion hardware that mediates between the outside world and the internal conventions of hardware compiled from Handel; this is achieved by choosing a Channel Protocol Converter, or CPC. For example, channel communication between the FPGA and the transputer on the HARP board might be synchronised by setting bits in a flags register and interrupting the transputer, and the appropriate protocol converter hardware would convert between this convention and the handshaking convention used inside the FPGA.

Typically, protocol converters are uni-directional, so you have to decide in advance whether a certain channel will be used for input or for output. A few converters, like those used to access external RAM, provide a bundle of several uni-directional channels, some used for input and some for output.

7.1 Simulator channels: CPC_SimIn, CPC_SimOut.

```
CPC_SimIn  : EXPR -> CPC * IO_SPEC
CPC_SimOut : EXPR -> CPC * IO_SPEC
```

These converters are designed to be used only with the Handel simulator during testing. They do not have any parameters concerned with physical location of pads connecting the channels to the outside world and are thus easier to use in an early stage of development. Each CPC takes a channel as its only parameter and delivers the structure required by the compiler.

The following program show a simple example of a program which uses two simulated channels:

```
val X   = Int  8 "X"
val Ch1 = Chan 8 "Ch1"
val Ch2 = Chan 8 "Ch2";
```

```
prog := Handel (
        [ CPC_SimIn Ch1,  CPC_SimOut Ch2 ], [ X ],
        While(TRUE,  Seq [Ch1 ?? X,  Ch2 !! X + C 1 ]));
```

7.2 Straight-through channels: CPC_NullIn, CPC_NullOut.

These protocol converters are completely empty and simply pass the data bus and the two handshake signals for the internal channel to the outside world via named pads. The data pads are presented least significant first, and the handshake pads appear in the order *receive_ready, transmit_ready*. The external device must conform to the communication protocol that is used between Handel processes.

```
val X  = Int  4 "X"
val Ch = Chan 4 "Ch"
val data_pads  = [ "p10", "p11", "p12", "p13"]
val rx_rdy_pad = "p50"
val tx_ack_pad = "p55";

prog := Handel (
  [ (CPC_NullOut (data_pads, rx_rdy_pad, tx_rdy_pad), Output Ch)],
  [ X ],
     While (TRUE, Seq [ X := X + C 1, Ch !! X ] )
  );
```

This example gives fictitious names to the four data pads and two handshake pads on the target FPGA. In a real situation, the user might care to put such definitions in a file specific to a particular chip or board, so that they can be shared and easily be incorporated into Handel programs.

The full CPC specification consists of a pair of elements. The first is one of the system-provided CPC generator functions applied to the pad description structure. These CPCs have the type:

```
CPC_NullIn  : string list * string * string -> CPC
CPC_NullOut : string list * string * string -> CPC
```

The second component is a data structure indicating which Handel channels are to be connected to the CPC and which data direction they will use.

The Handel channel protocol is such that the externally-produced handshake line (in this case `rx_rdy` since it is an output channel) can be raised at any time to signal that the outside world is ready to recieve data from the channel. The internally-generated handshake signal will be raised whenever an output command to the channel is executed. When both handshake lines are high, synchronisation takes place and the communication is scheduled. All signals are sampled on the rising edge of the global clock. The external handshake signal must be removed before the rising edge of the clock pulse after the communication is scheduled or it will be taken as a request for a further communication.

7.3 Communication via ports: CPC_PortIn, CPC_PortOut.

These CPCs are very similar to the null channels. However, unlike channels, ports are always ready to communicate. Thus, the Handel program is completely responsible for synchronising the communication. To reflect this, there is no returning handshake signal from the environment present in the external interface. Internally, the CPC ties the returning handshake signal to

the originating handshake signal. The handshake signal originating from the Handel program is present in the interface, so that the environment can take note of when an input or output communication with the port takes place.

```
CPC_PortIn  : string list * string -> CPC
CPC_PortOut : string list * string -> CPC
```

7.4 Communication through ports: CPC_NhPortIn, CPC_NhPortOut.

These are similar to `CPC_PortIn` and `CPC_PortOut` except that the remaining outgoing handshake line is also dropped from the interface. Data is communicated whenever the Handel program writes to the port, but the outside world doesn't know when this is happening.

```
CPC_NhPortIn  : string list -> CPC
CPC_NhPortOut : string list -> CPC
```

7.5 Communication with a transputer event line: CPC_EventOut.

This CPC is designed to interact with the event channel of a transputer. The two pads are connected to the transputer's event acknowledge and request pins respectively.

```
CPC_EventOut : string * string -> CPC
```

7.6 Interfaces to external SRAM: CPC_SRam.

This is the channel-based converter for external (off-chip) static ram. See Section 8 for full details of its use.

8 Off-chip RAM

Both the HARP board and our simulator support access to RAM that is not part of the FPGA chip. Only certain types of static RAM are currently supported. This RAM is treated as a process which communicates over three channels. The `Addr` channel carries the address to the ram, the `Write` channel carries the write data to the ram, and the `Read` channel carries the return data from the ram.

To write to a RAM location, a Handel program should *simultaneously* output on both the address and write channels, like this:

```
Par [Addr !! a,  Write !! x]
```

This command never needs to wait for the RAM to be ready, so it executes in a single clock cycle. One can annotate this with the `TakingTicks` tag for the benefit of the optimiser:

```
Par [(Addr !! a) TakingTicks 1,  (Write !! x) TakingTicks 1]
```

To read from the RAM, the Handel program should *simultaneously* output on the address channel and input on the read channel, like this:

```
Par [Addr !! a,  Read ?? x]
```

This command also executes in a single cycle and can be annotated as above. Although this seems to require the RAM to send the data before it has received the address, it is physically possible because the address value is present on the channel before the clock event that synchronises the communication. At the start of their execution, these output and input primitives merely

signal their *readiness* to communicate; it is entirely up to the remote (SRAM) process to decide *when* these communications will take place. This particular protocol converter uses knowledge of the internal channel implementation to achieve a useful doubling of speed. Thus, the protocol converters for these off-chip RAMs are necessarily technology-dependent and their design needs a good understanding of the external device and of the hardware generated by the compiler.

Of course, two parallel processes must not try to use the same RAM at once; but this is just an instance of the rule that two processes must not try to output to the same channel at once. The effects of trying to communicate on the address channel without simultaneously communicating on either the read or the write channel, and so on, are undefined.

The following example shows an example of a simple program using off-chip RAM:

```
val AW       = 3
val ram_size = 2**AW
val hi_addr  = ram_size-1
val DW       = 8
val A        = Int AW "A"
val V        = Int DW "V"
val SUM      = Int DW "SUM"

(* RAM0 defines an AW x DW-bit SRAM with chip-enable (CE),
   write-enable (WE), and output-enable (OE) control signals.
   We invent some chip pad names here for the interface signals as we
     are not bothered about a real implementation.
 *)
val RAM0 = MakeExtRam ("RAM0", AW, DW);
val RAM0_IF =  ( "RAM0",
                 ["A0", "A1", "A2"],
                 ["D0", "D1", "D2", "D3", "D4", "D5", "D6", "D7"],
                 "CE", ["WB"], ["O_EN"]
               );

prog := Handel (
  [ (CPC_SRam RAM0_IF, RamIF RAM0) ],
  [ A, V, SUM ],
  "Add up all RAM contents and overwrite location 0 with the sum" Comments
  Seq [
    Until (A == C ((hi_addr+1) mod ram_size),
      Seq [ RamRead RAM0 (A, V),
            SUM := SUM + V,
            A := A + C 1
          ]),
    RamWrite RAM0 (C 0, SUM)
    ]
  );
```

Not surprisingly, the most complex part of the program is the definition of the protocol converter. The chosen protocol converter is `CPC_SRam` which supports reading and writing to an off-chip SRAM and is also keyed into the built-in Handel simulator. `CPC_SRam` takes a tuple as its parameter which gives the interface a name (`RAM0`), and provides names for all the FPGA pads which are connected to the RAM chip. In this example explanatory names have been given, rather than the names of pads on any currently available FPGA chip.

The parametrised protocol converter, `CPC_SRam RAM_IF`, forms one half of the only external interface in this example. The other half is the specification of the internal channels connected to this protocol converter. For convenience, the compiler knows about some generic styles of interface, and `RamIF` denotes the style of interface with an address channel and two unidirectional data channels.

In this example, the program uses simple SML functions for reading and writing to the off-chip RAM to aid readability of the program. These are Handel/ built-in functions, but users can obviously write their own.

```
fun RamRead  (ram:EXT_RAM) (addr, data) =
    ((#1 ram)!!addr) TakingTicks 1 || ((#2 ram)??data) TakingTicks 1;
fun RamWrite (ram:EXT_RAM) (addr, data) =
    ((#1 ram)!!addr) TakingTicks 1 || ((#3 ram)!!data) TakingTicks 1;
```

8.1 Simulating with external RAMs

The `simRamList` function calls the simulator function `sim` and also allows the user to preset the contents of any simulated external rams.

```
simRamList : (string * int * int * int list) list -> unit
```

The parameter of the `simRamList` function is a list of a 4-tuples: (name of ram, its address width, its data width, list of initial values); one for each ram used. The following shows how to invoke the simulator (having first compiled the above program) with the ram contents predefined by a given list.

```
simRamList [("RAM0", AW, DW, 1 -- hi_addr+1)];
```

To look at ram contents after simulation use:

```
peekRange "RAM0" 0 hi_addr;
```

The following is a similar example of simulation, but with the initial RAM contents predefined by a function:

```
simRamFun : (string * int * int * (int -> int)) list -> unit

fun init_fun i = 10 + i;
simRamFun [("RAM0", AW, DW, init_fun)];
```

9 Using the HARP board

Each HARP board has 128K of static RAM organised as two banks, each of 32K x 16-bit directly connected to a Xilinx 3195 chip (a 3090 on the single prototype board). These banks are completely independent and can be used simultaneously for reads or writes to different addresses. They could also be used together as a single 32k x 32-bit memory. The Xilinx chip also sits on the bus of the T805 transputer and has more or less full access to the bus and to the T805 Event pins.

A configuration file exists for the FPGA which maps this SRAM onto the bus of the transputer. Using this, it is possible to set up data in the SRAM directly from a program running on the transputer, then reload the FPGA with your Handel program, run it to completion (as signalled by programmed use of the Event channel) and then reload the SRAM mapping configuration to examine the data left in the SRAM by the Handel program.

The `CPC_SRam` converter mentioned in Section 8 works on the HARP SRAM, but does not support all possible configuration options at present. The file `/mclab/page/handel/harp_pins.sml`

contains definitions of the Xilinx pin assignments on the HARP board; this will be needed for setting up the parameters for any HARP CPCs.

The following is part of an application program for the HARP board that uses this CPC and the Event CPC.

```
use "/mclab/page/handel/harp_pins.sml";
val AW   = 15   (* For an address width for 32K *)
val DW   = 16   (*     of 16-bit words          *)

val Addr = Chan AW "Addr"
val Din  = Chan DW "Din"
val Dout = Chan DW "Dout"
val RAM_IF =  ( "RAM0", SRL_A, SRL_D,
                SRL_CE, [SR_WB0, SR_WB1], [SRH_IO_EN]
              );

val Ram_CPC = (CPC_SRam RAM_IF, RamIF (Addr, Din, Dout))

val Event     = Chan 0 "Event"
val Event_CPC = (CPC_EventOut EventPads, Output Event);

prog := Handel (
  [ Ram_CPC, Event_CPC ],
  ....
```

10 Using the compiler

The Handel compiler is an SML program that is used from the top-level interactive read-eval-print loop of SML. Integrated with it is a gate-level simulator that can simulate the hardware generated by the compiler, with input and output to the screen. After putting your program in a file, say `sample.sml`, here are the steps to follow in compiling and simulating it:

1. Start the compiler by typing "`handel`" (assuming that you have `/mclab/page/handel` or another suitable directory on your search path). The SML top-level will start, and a minus sign will appear as the prompt.

2. Type "`use "sample.sml";`" to load your program into the SML system. The program is written as an assignment to the global variable `prog`, and the SML system will build a data structure that represents the program and assign it to the `prog` variable.

   As it loads your program, the SML system will display large amounts of uninteresting and largely meaningless stuff. You should watch out, however, for error messages from the SML system that mean your program is ill-formed when considered as an SML expression, or that it contains SML type errors.

   After this, the `prog` variable contains an SML data structure that is the abstract syntax tree of your Handel program. Subsequent commands to the compiler system usually need no parameters as they all look in the `prog` variable for the program. One important command available at this stage is "`pp();`" which pretty-prints the Handel program on the screen in a form similar to an occam program. This can be used to verify that your SML commands actually created the Handel program that you intended.

3. To compile the program into hardware and produce netlist output files type any of the following:

```
c();    (* simple compilation             *)
co();   (* same with hardware optimisation *)
cof();  (* same with full optimisation     *)
```

   "`c();`", "`co();`" or "`cof();`". at the minus sign prompt. As it processes your program, the compiler may print error messages, and it will probably print messages like these:

```
After compilation       : 15 LATCHES, 52 GATES, 3 INVERTERS; SIZE 75
After fast optimisations:  9 LATCHES, 15 GATES, 3 INVERTERS; SIZE 26
After fan-in adjustment :  9 LATCHES, 15 GATES, 3 INVERTERS; SIZE 26
```

   to indicate how much hardware it has generated. The `SIZE` parameter is the difference between (a) the sum of the total number of inputs to all the components in the circuit and (b) the total number of components; a circuit with fewer but more complex gates may have a larger `SIZE` than one with a larger number of simpler gates.

   After this step, the compiler has generated a representation of the hardware in memory, and written the netlist to one or more files on disk. It is the in-memory representation that is used by the simulator. The disk-file representation is only used by later stages of the FPGA configuration process. Thus, it is only possible to use the simulator on a program immediately after it has been compiled.

   You may want to `pp();` the program again at this point, as the compiler makes source level transformations to the program, for example when inferring the widths of constants, or reporting certain errors or warnings about the program.

4. Run the simulator by typing "`sim();`". The ensuing output shows the values of all program variables at each clock cycle. When a simulated external channel is ready for communication, you will be asked whether the environment of the program is ready to communicate before being shown the output value or asked for the input value. By answering "no" (by typing "`n`") several times, you can watch the behaviour of your program when it is waiting.

By default, the compiler writes the output net-list to disk in two forms: a `.hwp` file in the compiler's own format, and another file in a proprietary format. For Xilinx FPGAS, the second file is in `.xnf` format; the netlist may be slightly different from the `.hwp` file, because the compiler adds extra gates to reduce the maximum fan-in of gates to 4 or less as required by Xilinx netlists for the 3000 series FPGAs for example. Both output files contain a pretty-printed version of the input program.

The `.xnf` or other proprietary-format file can be the input to vendor-provided software that will place & route the net-list on an FPGA chip. The `.hwp` file is of use to investigate the effects of optimisation, or to see what parts of a program contribute most hardware. The `.hwp` file is both more terse and more readable than the `.xnf` file. Normally however the user will not look at either of these files.

The name of the output files can be set by assigning to the SML string variable `file`. The default value is `"default"`, so the two output files are called `"default.hwp"` and `"default.xnf"`. The compiler can also generate BLIF files and netlists suitable for Concurrent Logic FPGAs. However, as we do not normally use these facilities, these implementations may not be complete.

11 Statement timing

The timing of Handel programs is, very deliberately, kept simple. Essentially, *all* expressions in Handel programs evaluate within a single clock cycle. The major corollary is that assignment and `Delay` statements each take one clock cycle to execute, as does communication which is ready to run. *All* other statement constructors (except `Stop` of course!) take no additional clock cycles for any housekeeping operations, and thus their timing is determined solely by the statements in their bodies which are actually executed.

The following table summarises the timing behaviour of Handel statements in terms of clock cycles. The actual length of a clock cycle can not in principle be determined at compile-time, as it depends on the results of the (NP-hard) netlist-to-FPGA mapping performed by the vendor's FPGA software.

Statement	Clock cycles for execution
Skip	0
Delay	1
Stop	∞
Assign	1
ChanIn (when ready)	1
ChanOut (when ready)	1
PriAlt (when ready)	1 (time of guard statement) + time of statement executed
While	Σ (times of guarded statements executed)
Until	Σ (times of guarded statements executed)
If	time of guarded statement executed
Case	time of guarded statement executed
Seq	Σ times of nested statements
Par	maximum (times of nested statements)
Let	time of nested statement
Declare	time of nested statement
Tag	time of nested statement

12 Advanced features

Here we describe some features of the Handel language and compiler which are not needed by beginning users, but which may well be useful for later projects. They allow the user to write programs which can be compiled into more efficient implementations by the explicit use of sharing, for example.

12.1 Sub-expressions and sub-statements

Sometimes it is necessary to avoid building the same hardware twice, when only one copy is actually needed. Handel has a sub-expression capability to support this. Sub-expressions can be defined once and used multiple times, either in a statement or an expression. The hardware is generated from the sub-expression declaration, not from its use. The relevant parts of Handel abstract syntax are:

```
datatype EXPR    = Where   of  LETENV  * EXPR
                 | Subexp  of  string
                 .....
datatype STAT    = Let     of  LETENV * LETENV2 * STAT
```

```
                  .....
withtype LETENV  = (string * int * EXPR) list
and      LETENV2 = (string *       STAT) list
```

The `Subexp` constructor takes a string identifier for the sub-expression and references the value of the subexpression, which is in turn defined by either the `Where` or `Let` constructor. The `int` parameter in the environment list is the width in bits of the sub-expression. For convenience, we allow the name of a sub-expression to be any string: it is not used to construct signal names in the `.xnf` output.

In the following example, a single multiplier is implemented once and used twice. The statistics show the size of this program with and without sharing:

```
val DW  = 16
val X   = Int  DW "X"
val Y   = Int  DW "Y"
val Ch1 = Chan DW "Ch1"
;

prog := Handel (
  [ CPC_SimOut Ch1 ],
  [ X, Y ],
  While (TRUE,
      Let ([("X*X", DW, (X * X) <- DW)], [],
            Y := Subexp "X*X" + Subexp "X*X"  $$
            (Ch1 !! Y  || X := X + C 1)
            )
        )
  );

After fast optimisation : 35 LATCHES,  605 GATES, 3 INVERTERS; SIZE  822
After fast optimisation : 36 LATCHES, 1372 GATES, 3 INVERTERS; SIZE 1859
```

These subexpressions are actually parameterless functions. They are *not* evaluated to an integer value at the point of declaration and given the same value throughout the scope of the declaration; they are thus very different from the `VAL` constructor in occam for instance.

The expression hardware generated is purely combinational hardware which depends on the current values of any variables used in its definition. If these values change, then so does the value of the sub-expression. Handel sub-expressions are thus in effect *parameterless functions.*

Sub-expressions are defined in the context of sub-expressions placed earlier in the list, so that they can be nested. Recursive definitions are of course unimplementable in finite hardware, and not supported.

12.2 Local declarations

Declarations of variables and channels can be placed close to their use, rather at the top level of a Handel program using the `Declare` constructor:

```
datatype STAT   = Declare  of   EXPR list * STAT
                  ....
```

The `EXPR` list is the list of resources being declared, and their scope is the statement. Because of the naming restrictions imposed by the way we use the Xilinx software, these local declarations don't operate quite as you might like. Firstly, all variable and channel names used throughout a

Handel program really should be *distinct*. Secondly, no scope checking is currently done by the compiler, so it is the responsibility of the user to ensure appropriate hygiene rules.

In essence, the situation is very much as if all resources *were* defined at the top level. However, there are two benefits from this rather inadequate treatment of local declarations. Firstly, it can aid documentation, by keeping resource declaration close to the point of use. Secondly, it makes it possible to generate fragments of Handel with local variables by SML functions, which therefore do not need to affect the top-level declaration list. Such SML functions can of course use the `Int` and `Chan` keywords locally, so that they can be completely responsible for generating code fragments together with the associated declarations.

12.3 Tagging of statements and expressions

It is possible to add a string 'tag' to any Handel expression or statement. There are several types of these tags as shown in the following extracts from the compiler abstract syntax:

```
datatype TAG
   = Comment      of  string (* User comments                        *)
   | AutoComment  of  string (* Compiler-introduced comments         *)
   | Warning      of  string (* Compiler-introduced warnings         *)
   | Error        of  string (* Compiler-introduced error warnings   *)
   | Pos          of  POS    (* Position in source file             *)
   | Ticks        of  INT    (* For attaching timing assertions     *)
   | Width        of  int    (* To tag widths after inferencing     *)

datatype EXPR   = TagExp  of  TAG * EXPR
                  ....
datatype STAT   = Tag     of  TAG * STAT
                  ....
```

The `Comment` constructor simply allows the user to attach a comment to some fragment of program for documentation purposes. The `AutoComment` constructor is for comments which are added to the user program by the compiler, such as when the user requests statement timing data to be added. The compiler may likewise add tags of type `Warning` or `Error` to the program if some violation, or possible violation, of its 'health checks' is detected. The `Pos` and `Width` tags are for the compiler's internal use to aid error reporting and width inferencing respectively. None of these tags have any semantics.

Currently the only one with any semantics is a `Ticks` tag which, when attached to a channel input or output statement, is interpreted as an assertion that the communication will complete in exactly the number of clock cycles specified, *no matter when it is scheduled*. Its usual purpose is to tell the compiler that some communication with the outside world will always complete in one cycle and this may allow the compiler to further optimise the hardware produced.

12.4 Signal names and the bus constructor

The `Bus` constructor takes a list of strings which are signal names and delivers them as an `EXPR` which can be used in any Handel expression. This can be useful for joining circuits compiled by the Handel compiler to those generated by other means, such as the Ruby compiler.

Joining such circuits together is also the major reason that certain wire names in a Handel netlist are simply formed from the names of the resources they are connected to. It allows the user to predict Handel wire names and use them in the construction of non-Handel circuits. The currently supported names that are likely to be of use are shown in the following example by reference to Handel declarations:

```
val Reg = Int  16 "Reg"
val Ch  = Chan 16 "Ch"
```

Resource	Connections	Signal names
Register	input wires	`I_Reg_IN_0 ... I_Reg_IN_15`
Register	output wires	`I_Reg_0 ... I_Reg_15`
Register	clock enable	`I_Reg_CE`
Channel	data wires	`C_Ch_0 ... C_Ch_15`
Channel	receive-ready wire	`C_Ch_RXRDY`
Channel	transmit-ready wire	`C_Ch_TXRDY`

12.5 Explicit channel synchronisation

The Handel abstract syntax has two constructors, `RxRdy`, and `TxRdy` which support forms of synchronisation not possible with the occam-style commands on their own. They each take a channel as parameter and deliver a Boolean EXPR which is `TRUE` if the corresponding channel is ready to communicate in the appropriate direction. `RxRdy` is `TRUE` if there is some receiving process (`??`) ready to run on the channel; `TxRdy` is `TRUE` if there is some transmitting process (`!!`) ready. These expressions allow programs to 'peek' at channels to see if a communication issued in this cycle would succeed. They can support optimisations of time-critical programs and also allow the user to build their own `Alt`-style commands.

For example, the following program computes Fibonacci numbers correctly and obeys occam scope and usage rules:

```
val DW  = 16
val A   = Int  DW "A"
val B   = Int  DW "B"
val Ch1 = Chan DW "Ch1"
;

prog := Handel (
  [ CPC_SimOut Ch1],
  [ A, B ],
  [A, B] := [C 1, C 1] $$
  While (TRUE,
            [A, B] := [B, A + B]  $$  Ch1 !! A
        )
  );
```

This program obviously produces a new result every two clock cycles. If we want to produce a result on every clock cycle we might re-write the iterated command as:

```
[A, B] := [B, A + B]   ||  Ch1 !! A
```

This program violates the occam usage rule concerning variables in a `PAR` statement. However, the synchronous semantics of Handel mean that this is a perfectly well-behaved program. In fact, in this context, the only badly-behaved Handel program is one which tries to update a shared resource (variable, channel, RAM etc.) more than once in the same cycle.

Even though the semantics of this program are perfectly well-defined, they may not correspond with the programmer's intent. If the output channel is *always* ready to receive a value, then the timing semantics of Handel guarantee that the parallel assignment and the communication will both be scheduled at the same instant and will both complete in a single cycle. This means that the correct (i.e., current) value of `A` will be communicated. If however, the output channel is

blocked for even one cycle, `A` will be updated before it is communicated and one of the desired output values will have been lost forever.

The situation can be recovered and made perfectly hygienic if the programmer provides a guarantee that the output channel will always be ready to run. This can be done by a proof of the timing behaviour of the receiving process, or by using a Port-style CPC on the output channel, rather than a handshaken CPC. An output statement which is guaranteed to complete in a single cycle can be annotated by the keyword `TakingTicks`, such as

```
(ch1 !! A) TakingTicks 1
```

which will allow the hardware optimiser to perform a better optimisation.

If this guarantee cannot be made, it is still possible to write the program in such a way that it produces a new value on every clock cycle that the channel is ready to receive. One way of doing this is by using `RxRdy` to peek at the channel to see if it is ready, and delaying the assignment and communication until it is:

```
While (TRUE,
  Par [ While (~(RxRdy Ch1), Delay)  $$  [A, B] := [B, A + B],
        Ch1 !! A
      ]
  )
```

These facilities can sometimes be very useful in contexts like this where saving a single clock cycle in an inner loop is important. Here, it can double the speed of the program by reducing the inner loop from two cycles to one. However, they are not straightforward to use, nor should you expect them to be. It is in fact quite easy to write a program with these primitives which is unimplementable within Handel synchronous semantics. For instance, if both ends of a channel try to peek at each other and use the information to decide whether they will try to communicate with the other, there is an unresolvable decision to be made. Compiling such a program will usually result in a loop in the combinational hardware (which the simulator will detect). Such programs are hard to analyse automatically, so it is the user's responsibility to analyse carefully any programs which use them.

12.6 Compiler control variables

There are a number of global variables which control various aspects of the compiler's operation. Variables with related functions are placed together into structures. Not all variables are listed here, as some of them are not meant for general use.

12.6.1 Simulator control

After compiling a program and before simulating it, you may want to alter the list of variables that the simulator prints out. By default it prints all variables on each clock cycle. This is the signature of the module:

```
structure Sim_control = struct
                      (* ident.   base   variable  *)
    type MONITOR       = (string * MON_FUNC) * EXPR
    val monitor_list   = ref [] : MONITOR list ref
    val radix          = ref 10   (* Base for displaying numbers.*)
    val display_step   = ref 1;   (* How often to print outputs. *)
end;
```

By explicitly assigning to the `monitor_list` variable, it is possible to force the simulator to print out any variable or channel any number of times with different bases, for example.

```
val X  = Int 16 "X";

prog := Handel ( [ ],  [ X ], Until (FALSE,  X := X + C 1)  );
c();
Sim_control.monitor_list  :=
   [(("X", Sim_res.radix 10), X), (("X", Sim_res.radix 16), X)];
sim();
```

Note that if you compiled *and optimised* this program it would be optimised away to no hardware at all, since it performs no output to the environment!

By examining the compiler source code, advanced users can write their own monitor functions to do more interesting things with the values from simulations. You may want to log them into a file, or display them as a picture in an 'xterm' for example.

12.6.2 FPGA control

The following structure contains variables that deal with particular target technologies. They allow the netlists to be generated in Xilinx (default), VHDL, Concurrent, or Blif formats. The latter three are not well supported since we don't use them on a daily basis; however they might be usable. The relevant output files are suffixed `.xnf, .vhdl, .conc, .blif` respectively. The three pad strings define which FPGA pins the Clock, Stop, and Finish signals are wired to, if any.

```
structure fpga_control =
  struct
    (* Extracts from harp_pins.sml are used for default values. *)

    val Harp1a_fpga = "3090PQ160-125"
    val M2          = "P44"             (* Harp1 LED (output) pad            *)
    val MasterClk   = "P160"            (* Frequency Synthesiser input pad   *)
    val not_ErrorX  = "P55"             (* Tram Error line - NOT T805 VISIBLE*)

    (* Miscellaneous definitions for the netlist output.
       Defaults values are set for HARP1 boards.
       Many of these options only make sense for Xilinx chips.
     *)
    datatype FPGA  = Xilinx2000 | Xilinx3000 | VHDL | VHDL2 | Concurrent | Blif

    val fpga_type        = ref Xilinx3000 (* FPGA family                     *)
    val fpga_chip        = ref Harp1a_fpga(* Destination chip type           *)
    val clock_pad        = ref MasterClk  (* Global clock input pad
                                             Xtal Osc+AClk used if set to "" *)
    val clock_divider    = ref 1          (* Between InputClock & ClockDrive *)
    val notError_pad     = ref not_ErrorX (* Active if Stop is executed      *)
    val finish_pad       = ref M2         (* Asserted on termination         *)
    val carry_weight     = ref 50         (* Timing weight for carry lines   *)
    val critical_weight  = ref 100        (* Timing weight for critical lines*)
  end;
```

12.6.3 Print control

The following variables affect the way that various things are rendered by the pretty-printer. The `full_sig_names` variable controls whether the names of nets in the final netlist are in a

symbolic form somewhat related to the program variables and control struictures, or whether they are essentially numbers. Xilinx software has an annoying habit of truncating net names in summary output, which means that it can't be processed automatically by user software; hence the provision of this mode.

The remaining control variables should be self-explanatory:

```
structure print_control =
  struct
    val verbosity      = ref 1     (* How noisy to be during compilation *)
    val debug_level    = ref 0     (* How noisy to be during simulation  *)
    val const_widths   = ref false (* suffix constants with their widths *)
    val int_width      = ref 32    (* width of an occam INT              *)
    val var_widths     = ref true  (* suffix vbl decls with their widths *)
    val merge_decls    = ref true  (* merge declarations of same type    *)
    val user_comments  = ref true  (* Whether to pp these things ....    *)
    val auto_comments  = ref true
    val warnings       = ref true
    val assertions     = ref true
    val full_sig_names = ref true  (* Print signals as names (or numbers)*)
  end;
```

12.6.4 Transform control

The transform control variables each define whether the compiler should make the associated source level transformation, or apply the associated check. If `add_times` is true, the compiler prepends a comment to every statement giving its minimum and maximum execution time, in clock cycles. Similarly, if `add_space` is true, statements are commented with an *estimate* of the number of gates and flip-flops that will be generated by the compiler. This does not include the effect of any optimisations that the compiler may apply.

The width inference system can be disabled with `width_infer`. The `force_delay` transformation will add `Delay` statements to avoid any combinational loops that might result from `While` and `Until` loops with guarded statements that might execute in 0 cycles. The remaining two variables control the application of checks on the soundness of the Handel program.

```
structure transform_control =
  struct
    val add_times     = ref false  (* Add statement timings to source?     *)
    val add_space     = ref false  (* Add statement h/w space to source?   *)
    val width_infer   = ref true   (* Infer widths of constants?           *)
    val force_delay   = ref true   (* Add minimal delays to While/Until    *)
    val balance_delay = ref false  (* Add delays to balance If and Case?   *)
    val check_decls   = ref true   (* Check declarations in user program   *)
    val health_checks = ref true   (* Apply health checks to user program  *)
  end;
```

12.6.5 Optimiser control

These variables control the netlist optimiser. `par_minimax` will remove the synchronisers from any arms of a Par which have a (statically determinable) execution time always guaranteed to be less than some other arm.

The bottom-up, Common Sub-Expression extractor removes (some) gates with an identical function, and is controlled by `cse_iterations`. Set to a natural number it will run for that

number of iterations, or until fixed point whichever comes first; set negative it will run until fixed point. It will only run if the netlist optimiser is also run. The remaining functions are obvious from the comments:

```
structure optimiser_control =
  struct
    val io_single_tick  = ref true  (* Remove synchronisers from fast i/o   *)
    val par_minimax     = ref true  (* Minimax the Par finish signals       *)
    val strip_comments  = ref false (* Remove comments from block list      *)

    (*  These enable specific gate-level optimisation passes, when full     *)
    (*  optimisation (`cof ()' or `optimise 9') is selected.                 *)
    val graph           = ref true  (* Fast pattern-matching and others.    *)
    val cse             = ref true  (* Common subexpression elimination.    *)

    (* Enabling both tci and tci_inv at once is counterproductive as        *)
    (* tci can actually reduce the amount of information available to       *)
    (* tci_inv, and hence it can make fewer optimisations.                  *)
    val tci             = ref false (* Transitive closure of implication.   *)
    val tci_inv         = ref true  (* Implication through inverters too.   *)

    (* During full gate-level optimisation, all enabled optimisations are   *)
    (* repeated until there is no change in the circuit.  This variable     *)
    (* puts a limit on the total number of repetitions.  (~1 == no limit).  *)
    val max_iterations  = ref ~1    (* Iterate until fixpoint or this many. *)
  end;
```

12.7 Other compiler functions

This section lists the most commonly-used compiler commands and functions:

`c : unit -> unit` applies the compiler to the 'prog' variable

`co : unit -> unit` applies the compiler to the 'prog' variable and then performs a partial hardware optimisation

`cof : unit -> unit` applies the compiler to the 'prog' variable and then performs a full hardware optimisation

`sim : unit -> unit` runs the simulator on the most recently compiled program

`f : string -> unit` sets variable *file*; used for subsequent input/output operations. No suffix should be given.

`u : unit -> unit` issues an SML 'use' for *file.sml*

`## : EXPR -> int` returns the width (in bits) of an expression.

`pp : unit -> unit` pretty-prints the 'prog' variable

`ps : STAT -> unit` pretty-prints a statement

`pe : EXPR -> unit` pretty-prints an expression

`pd : EXPR -> unit` pretty-prints a declaration

12.8 Signal timing

The hardware circuits produced by the Handel compiler are simple synchronous finite state machines of the Mealy form. All state is represented by explicit flip-flops and there are no combinational logic cycles. All flip-flops are edge-triggered from the same global clock. Conditional updating of a flip-flop is achieved by controlling its *clock-enable* input. Any hardware implementation must ensure that all flip-flops are set to 0 before the clock is started, and that clock-skew is low enough that no flip-flop to flip-flop combinational circuit is faster than the maximum clock-skew. These conditional are fulfilled by implementations on the Xilinx 3000 series devices that we use. The clock frequency is determined solely by the maximum flip-flop to flip-flop combinational delay, including any path via external hardware, plus any setup time for the flip-flops used.

In normal operation, there are only two 'system' signals emitted by the circuits. One is the `FINISH` signal which indicates that the computation has terminated. The other is the `STOP` signal, which is asserted when any Handel process executes the `Stop` command. These two signals may be high for only a single cycle, are therefore also made available in latched form as `FINISH_OUT` and `STOP_OUT`. Once asserted, these signals will be high until the circuit is reset. In normal use, these signals are not used. Instead, the user is encouraged to program explicit channel communications to signal termination or failure to the environment.

There is only one input system signal (apart from the clock of course). The `START` signal must be high at the rising edge of one clock cycle, and must thereafter be low, at least until the circuit asserts the `FINISH` signal. Currently, the circuit generates its own `START` signal from the clock, so the user does not normally have to be concerned with this signal either.

All other inputs and outputs of a Handel circuit correspond to channels declared as part of the external interface specification of the program. Each channel is possibly mediated by Channel Protocol Converter hardware. The hardware produced by CPCs is under the control of the user and is thus arbitrary. CPC circuitry may not conform to the design principles of Handel circuits (and indeed cannot when asynchronous interfaces are constructed).

Acknowledgement

This report relates in part to work carried out by Oxford University Computing Laboratory in the Esprit OMI/HORN (P7249) Research Programme.

A Appendix: Documentation files

The following files can be found in `/mclab/page/handel`:

README A file describing the current state of the compiler and the available files.

handel Executable code of the compiler for Sparc processors.

sample.sml A sample Handel program.

examples.sml A set of simple example programs in Handel, illustrating some of its features.

Handelast.sml The sml code from the compiler defining its abstract syntax tree structures. Useful for its definitions of the EXPR, STAT and PROG datatypes.

Sugar.sml The sml code of the standard 'syntactic sugar' functions already loaded into the compiler. Their use can ease the writing of Handel programs in SML.

harp_pins.sml Definitions of Xilinx 3195 pin assignments for the Harp1 board.

Much information on our work has been mounted on the world-wide web:

http://www.comlab.ox.ac.uk/oucl/hwcomp.html

In addition, there is a selection of documentation files, papers, and project reports on our hardware compilation work available by anonymous ftp. Connect to `ftp.comlab.ox.ac.uk` and find the documents in `Documents/techpapers/Ian.Page`. This area is also directly accessible from the web.

We also have a local news group, `prg.hwcomp`, which contains various announcements about local activities in hardware compilation.

B A tricky issue with loop implementation

The Handelimplementation of loops is chosen so that a loop with a body which executes in a single cycle will also perform an entire loop in a single clock cycle. This has the major advantage that it is possible to construct loops which execute at the actual clock frequency of the hardware.

This means that the loop guard must be evaluated by the hardware without taking any additional clock cycles. Another possibility exists which is to take one clock cycle to evaluate the loop guard. But as this would slow down these very important, single-cycle loops by as much as 50%, this was felt to be unacceptable.

The problem with using the faster implementation strategy for loops is that it will not work properly if the body of the `While` loop is a command that *may* execute in zero time, because that ultimately results in a cyclic path in combinational hardware.

The precise rule is that the circuit compiled for the loop body must not have a combinational path from its *start* signal to its *finish* signal, even if it is impossible for the body actually to execute in zero time. Loops that violate this rule are not common, because any initialisation code for an inner loop will take time and introduce flip-flops into the control path, breaking the combinational path.

However, to solve any remaining problems, the compiler incorporates an automatic transformation that will introduce one or more `Delay` commands into programs to force all loop bodies to always take at least one clock cycle to execute. If the user is not satisfied with where the compiler decides to place the necessary `Delay` commands, the program can be re-written to put them in a more appropriate place.

Fortunately, the user need not normally be concerned with this issue. Very occasionally, one additional clock cycle delay will be added to a user program in rectifying this situation. The user is informed if this transformation is made.

C Mapping Statements into Hardware.

The transformation of an arbitrary user program into normal form can be accomplished by a series of syntax-directed applications of the laws of programming to the program. The theoretical basis for this and the transformation steps themselves can be found in the references [5, 6, 7].

The current Handel compiler does not in fact follow these steps. Instead, the program is transformed directly into a netlist graph, which intentionally achieves the same result, but much more quickly. In fact it is much easier to comprehend the basis of the normal form transformation by looking at the fragments of netlist graph which are generated by each of the language constructs, rather than by looking at the normal form transformation steps themselves. The transformation laws are considerably obscured by the necessity to deal formally with shared use of the datapaths, which are just about the simplest parts of the hardware!

All variables in the user program are mapped to hardware registers which are constructed from sets of J-K flip-flops. The registers have input multiplexors if they have multiple sources. This happens when they are the target of more than one assignment or communication in the program. The control circuits for the statements in the program generate the multiplexor control signals and the clock enable signals for destination registers. As previously explained, all expressions are implemented as combinational logic.

In fact, this is about all there is to say about the datapath generation strategy, which is very straightforward. The datapath generated by this strategy thus exactly matches the dataflow graph of the original user program. If the programmer specifically wants some other form of datapath, it is his responsibility to transform the program into a form which exhibits the datapath architecture required, although we have implemented some automatic transformations of this nature for particular forms of datapath. The conversion of a user program into the combination of a machine code program and an application-specific microprocessor is a good example of such a transformation [8].

Each control construct in the program maps onto a control circuit in the hardware. We use a pair of control handshake signals (`start` and `finish`) for each circuit. A handshake signal which is active, is simply a signal which is high at the rising edge of the global clock. We also make an *environment assumption* that a start signal will never be given to a control circuit if there has not been a corresponding finish signal from any previous start signal. The individual circuits are designed to maintain this environment guarantee to any nested control circuits. Since the entire program is only a single statement at the top-level, this translates into an assumption that the environment will start the hardware program running just once and will not attempt to start it again before the program has completed.

To improve readability in the control circuit diagrams which follow, we use a box with a triangle in the upper-left corner to represent a single instance of a control circuit. Each control circuit has a single input and a single output signal which implements the control handshake protocol. The connections between these circuits and the datapath is fairly obvious and is thus not shown in any detail here.

C.0.1 Assignment.

Because of the method of handling expressions, the assignment statement is particularly simple to implement and its control circuit is shown in Fig. 1. The `start` signal forms the clock-enable signal for the destination register(s) of the assignment. At the end of the cycle in which the assignment is scheduled, the expression hardware has calculated the new value (by assumption), and it is thus loaded into the destination register. The `start` signal is delayed by a single cycle

to provide the `finish` signal, since an assignment is always scheduled immediately, and it always completes in exactly one clock cycle (again by assumption).

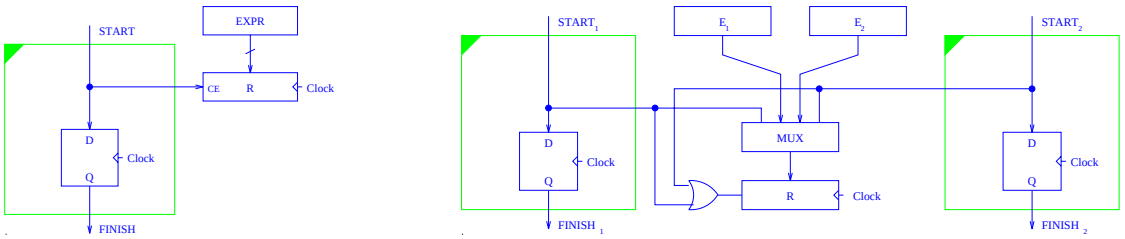

Figure 1: Assignment: Control and Data Multiplexing.

The left-hand part of Fig. 1 shows the circuitry generated by a single assignment of the form `R := EXPR`. The right-hand part of Fig. 1 shows the portion of datapath generated by two assignment statements which target the same register. An active `start` signal steers the appropriate expression value through the multiplexor to the input of the destination register. Whichever `start` signal is active also enables the destination register via the `OR` gate.

If the rules of occam programming are enforced, then any two assignments to the same variable cannot be in separate arms of a `Par` statement, so the issue of what happens when the two assignments are scheduled at the same time simply does not arise. If the user wants to use the additional transformational rules of Handel programs to allow such assignments into two parallel processes, then there is a proof obligation which must show that the two assignments cannot be scheduled in the same clock cycle, and the user must also shoulder the additional burden of arguing what the semantics of his particular use of such shared store is.

C.0.2 Sequential Composition.

The control circuit for sequential composition is trivially simple. The start and finish signals of the component processes are connected together together in a 'daisy-chain' as shown in the left-hand part of Fig. 2. It is particularly clear from this diagram that the control strategy is basically that of 'one-hot' control state encoding. This particular scheme appears to be well-suited to DPGA implementations since it requires little in the way of wiring resources when compared to encoded representations of control state.

C.0.3 Parallel Composition.

The right-hand side of Fig. 2 shows the control circuitry for parallel composition. This circuit passes control simultaneously to all the parallel statements to initiate parallel execution. The `PAR` statement of occam is defined to be synchronised, so that the whole construct terminates only when all of the constituent components have terminated. Thus, the `PAR` control circuit collects together the separate finish signals in a set of flip-flops and produces its own finish signal as soon as the last finish signal is generated. In addition, this signal resets the synchroniser flip-flops ready for the next time that this circuit is used.

Optimisations are routinely performed to remove such synchronisers, wherever a textual analysis of the program can demonstrate a partial ordering on the termination times of the individual processes.

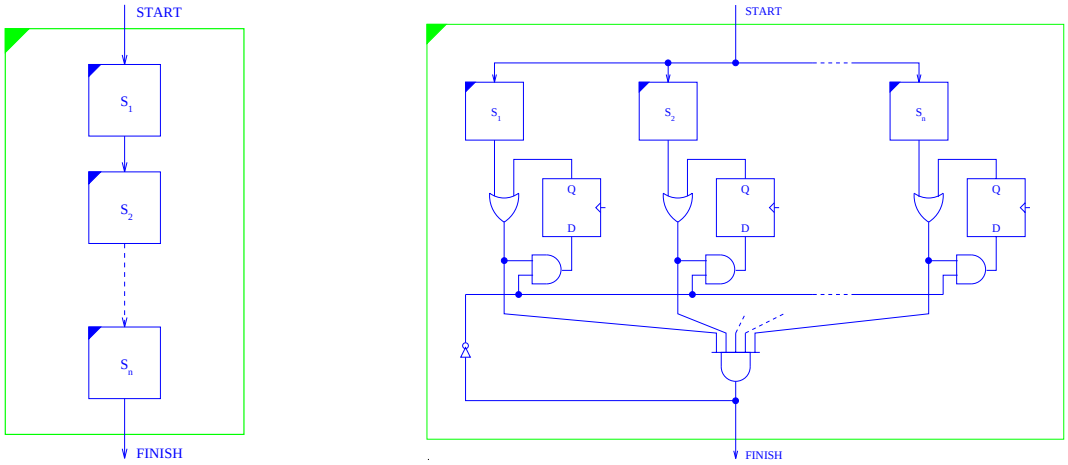

Figure 2: Sequential and Parallel Composition.

C.0.4 Miscellaneous Constructs.

Fig. 3 shows the control circuitry for some minor control constructs. The `Skip` and `Delay` constructs have no effect on the state of the computation. They take exactly 0 and 1 clock cycles respectively to complete.

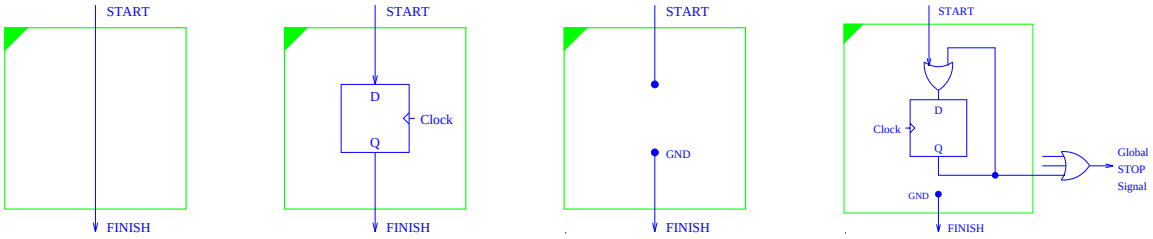

Figure 3: Skip, Delay, Stop1, Stop2.

The `Stop1` and `Stop2` circuits show two different refinements of the `Stop` construct. `Stop` represents a broken computation ('bottom' in the semantics). The `Stop1` circuit simply refuses to pass on the handshake signal (though of course it could do anything at all!) The `Stop2` circuit uses a local latch to remember that a particular instance of the `Stop` command was activated and the state of this latch can be monitored by external hardware for debugging purposes, for example.

C.0.5 Channel Input and Output.

Fig. 4 shows the control circuitry for synchronised channel input and output. The left-hand circuit is the same for either and input or an output command. Synchronisation is achieved by looping back the `start` signal through a flip-flop and a multiplexor which is controlled by the `transfer` signal. The control token is trapped in this feedback loop until activation of the `transfer` signal.

The `ready` signal goes to the arbitration circuit and returns as the `transfer` signal to indicate when the partner to this communication is also ready to run. When both circuits are ready to communicate, the lower-left circuit is activated, which is simply the assignment circuit seen earlier.

Figure 4: Communication: Synchronisation and Arbitration

It can be clearly seen from these diagrams that communication is just distributed, synchronised assignment.

As with assignment, the scope and usage rules of occam guarantee that there can be no more than one input and one output command scheduled for the same channel at the same time, hence the very simple arbitration circuit on the right-hand side of the figure.

C.0.6 Binary Choice.

The left-hand side of Fig. 5 shows the control circuitry for the binary choice, or `If`, statement. The `start` signal is steered to trigger just one of the guarded commands under the control of the guard expression. Since only one command can be active, the `finish` signals of the two arms can be simply `OR`ed together to derive the completion handshake signal for the entire construct.

Figure 5: If and While Constructs.

C.0.7 Guarded Iteration.

The right-hand side of Fig. 5 shows the control circuitry for the `While` loop. It is somewhat similar to the `If` circuit except that here the start `signal` is steered either to the feedback loop or to form the `finish` signal, under the control of the circuit which implements the Boolean guard expression. The control token is trapped in the feedback loop until the controlling Boolean expression, `BE`, evaluates to `false`.

A very similar circuit exists for the `Until` loop which always executes at least once and tests the guard expression at the end of the loop. This form of the loop doesn't exist in occam but it is sometimes useful in hardware compilation since it avoids the need for the duplicated hardware (or procedure call) which would be necessary to implement the program `Until x Do y` as the program `y; While Not x Do y`.

There is a tricky design issue involved with these iteration circuits. The particular implementation shown here is not at all the obvious one, and without further steps being taken it is actually capable of failing. This implementation was pursued despite these problems because it leads to a simpler and more reasonable timing calculus than the alternatives.

The control circuit fails if the controlled statement `S` has any path through it which executes in zero time. This is because there is then a combinational path through the box `S`, which then creates a combinational loop in the hardware because of the feedback loop in the `While` control circuit. For instance, either of the programs `While x Do Skip`, or `While x Do (If a Then b Else Skip)`, would create a loop in the combinational circuitry because of the decision that a false-guarded loop should not take a clock cycle to execute.

The simplest way of implementing the desired semantics without introducing any undesirable combinational loops would be to insist that evaluating the guard expression should take one clock cycle itself, thus breaking the combinational loop. However with hardware compilation we will often be interested in writing loops which execute a single iteration in a single clock cycle. Here it would be unacceptable to introduce even one extra cycle into the loop execution as it would halve system performance.

The scheme adopted in Handel is to use the time-efficient circuit and to make the compiler perform a pre-transformation on the program to replace certain instances of `Skip` with `Delay` so that no zero-time loop bodies exist in the program.

References

[1] Laurence Paulson. *ML for the Working Programmer.* CUP, 1991.

[2] Chris Reade. *Elements of Functional Programming.* Addison-Wesley, 1989.

[3] Åke Wikström. *Functional Programming Using Standard ML.* Prentice Hall, 1987.

[4] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice-Hall, 1993.

[5] J.P. Bowen and He Jifeng. Programs to hardware. In P.G. Larsen, editor, *Tutorial Material, Formal Methods Europe '93, Industrial-Strength Formal Methods*, pages 437–450, 1993. In A.P. Ravn (ed.), *Provably Correct Systems (ProCoS)* tutorial.

[6] Jifeng He, Ian Page, and Jonathan Bowen. Towards a provably correct hardware implementation of Occam. In G.J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods, Proc. IFIP WG10.2 Advanced Research Working Conference, CHARME'93*, volume 683 of *Lecture notes in Computer Science*, pages 214–225. Springer-Verlag, 1993.

[7] Jonathan Bowen, Jifeng He, and Ian Page. Hardware compilation. In J.P. Bowen, editor, *Towards Verified Systems*, Real-time Safety-Critical Systems, chapter 10, pages 193–207. Elsevier, 1994.

[8] Ian Page. Automatic design and implementation of microprocessors. In *Proceedings of WoTUG-17*, pages 190–204, Amsterdam, April 1994. IOS Press. ISBN 90-5199-1630.