TRANSPUTER COMMUNICATIONS, VOL. 1(1), 3–15 (AUGUST 1993)

SUMMARY

The study of computing science is split at an early stage between the branches dealing separately with hardware and software; and there is a corresponding split in later professional specialisation. This paper explores the essential unity and overlap of the two branches. The basic concepts are those of occam, taken as a simple example of a high-level programming language; its notations may be translated by the laws of programming to the machine code of a conventional machine. Almost identical transformations can produce the networks of gates and flip-flops which constitute a hardware design. These insights are being exploited in hybrid systems, implemented partly in hardware and partly in software. They are being extended to earlier phases in the software life cycle, where fully automatic transformations are not available. The results may be applicable in embedded real time systems, where considerations of safety make provable correctness a criterion.

1 Correctness of design

The design of a complex engineering product like a real time process control system is decomposed into a progression of related phases. It starts with an investigation of the properties and behaviour of the process evolving within its environment, and an analysis of requirements for its optimal or satisfactory performance, or at least for its safety. From these is derived a specification of the electronic or program-controlled components of the system. The project then may pass through an appropriate series of design phases, culminating in a program expressed in a high level language. After translation into the machine code of the chosen computer, it is loaded into memory and executed at high speed by electronic circuitry. Additional application-specific hardware may be needed to embed the computer into the system which it controls. Each of these phases presents a conceptual gap, as wide and challenging as that between hardware and software. Reliability of the delivered system requires that all the gaps be closed. It is achieved not just by testing, but by the quality of thought and meticulous care exercised by analysts, designers, programmers and engineers in all phases of the design. The goal of our research is to discover and formalise methods which reduce the risks and simplify the routines of the design task, and give fuller scope for the exercise of human skill and invention in meeting product requirements at low cost and in good time. The goal of this paper is to convey an impression of the methods and intermediate results of the research.

In principle, the transition between one design phase and the next is marked by delivery of a document, expressed in some more or less formal notation. Each phase starts with study and acceptance of the document produced by the previous phase; and ends with the delivery of another document, usually formulated at a lower level of abstraction, closer to the details of the eventual implementation. Each designer seeks high efficiency at low cost; but is constrained by an absolute obligation that the final document must be totally correct with respect to the initial document for this design phase. Thus the requirements must be faithfully reflected in the specification, the specification must be fully achieved by the design, the design must be correctly implemented by the program, the program must be accurately translated to machine code, which must be reliably executed by the hardware. Although we have used different words in English to describe the correctness relation at each different level of design, we shall show that conceptually it is the same relation in all cases, namely logical implication, denoted by $\Leftarrow$.

????–????/93/010003–13$11.50

Received xx November 1993
Revised 4 February 1993

When the system is eventually delivered and put into service, all that really matters is that the actual hardware delivered should meet the overall requirements of the system. This is guaranteed by a simple mathematical property of the implementation relation: it is transitive. If P is implemented by Q and Q is implemented by R then P is implemented by R.

$$\text{If } P \Leftarrow Q \text{ and } Q \Leftarrow R \text{ then } P \Leftarrow R.$$

However long the chain of intermediate documents, if each document correctly implements the previous one, the overall requirements will be correctly implemented by the delivered hardware.

We have given a very simple account of the design process, and the reason why it can validly be split into any number of phases. The account is highly abstract: in concrete reality, complications arise from the fact that each of the design documents is written in a different notation, adapted to a different conceptual framework at a different level of abstraction. For example, a requirements document for a real time system may use timing diagrams or temporal logic, a specification may use set theory (Z or VDM), a design may use flow charts or SSADM, a program may use ADA or C, the machine code may be INTEL 8080 and the hardware may be described in pictures or as a netlist of components and wires. How can we be certain that a document serving as an interface between one design phase and the next has been correctly understood (i.e. with the same meaning) by the specialists who produced it as a design and the different specialists who accepted it as a specification for the next phase? The utmost care and competence in each individual phase of design will be frustrated if bugs are allowed to congregate and breed in the interfaces between one phase and the next.

The solution is to interpret every one of the documents in the chain as a direct or indirect description at an appropriate level of abstraction of the observable properties and behaviour of some system or class of system or component that exists (or could be made to exist) in the real world. These descriptions can be expressed most precisely in the language which science has already shown to be most effective in describing and reasoning about the real world, namely the language of mathematics. Such descriptions use identifiers as free variables to stand for observations or measurements that could in principle be made of the real world system, for example the position and momentum of a physical point, plotter pen or projectile, or the initial and final values of a global variable of a computer program [1].

A simple example of a mathematically expressed requirement is that for a straight line constant speed trajectory of our point of interest. Let x_t be its displacement on the x axis at time t, and let a be the desired velocity. Then within some desired interval the difference between the actual and desired position should, within the relevant period, always be less than some permitted tolerance:

$$| \, x_t - at - x_0 \, | \leq 0.4 \qquad \text{for all } t \in (3 \ldots 5).$$

If we also want steady motion on the y-axis, this is stated separately:

$$| \, y_t - bt - y_0 \, | \leq 0.4, \qquad \text{for all } t \in (3 \ldots 5).$$

The additional requirement is just conjoined by “and” to the original requirement. The use of conjunction to compose complex requirements from simple descriptions is a crucial

advantage of the direct use of logical notations at the earliest stage of a design project.

In this example, the requirements formalise the permitted tolerances on the accuracy of implementation. Of course an implementation is permitted to achieve even greater accuracy. For example, suppose the behaviour of a particular implementation is described by

$$(x_t - at - x_0)^2 + (y_t - bt - y_0)^2 < 0.1, \qquad \text{for all } t \leq 5.$$

This implies both of the requirements displayed above; consequently the implementation correctly fulfils its specification.

This notion of correctness is perfectly general. Suppose a design document P and a specification S use consistent naming conventions for variables to describe observations of the same class of system; and suppose that P logically implies S. This means that every observation of any system described by P is also described by S, and therefore satisfies S as a specification. Certainly, no observation of the system described by P can violate the specification S. That is the basis of our claim that the relationship of correct implementation is nothing other than simple logical implication, the fundamental and familiar transitive relation that governs every single step of all valid scientific and mathematical reasoning. It should therefore be no surprise that it is also fundamental to all stages and phases of sound engineering design.

2 Hardware

The example we have just described might have been part of the requirement on a control system, formulated at the start of a design project. Our next example describes the actual behaviour of the ultimate components available for its implementation, right at the final phase of electronic circuit design and assembly. Let the variables x_t, y_t and w_t stand for voltages observable at time t on three distinct wires connected to an OR-gate (Fig. 1). The voltage takes one of two values, 0 standing for connection to ground and 1 standing for presence of electrical potential. The specification of the OR-gate is that the value of the output wire w is the greater of the values of the input wires x and y. This relationship cannot be guaranteed at all times, but only at regular intervals, at the end of each operational cycle of the circuit. For convenience, the unspecified duration of each cycle is taken to be the unit of time. The behaviour of the OR-gate is described as an equation:

$$w_t = x_t \vee y_t, \qquad \text{for all } t \in (0, 1, 2, \ldots).$$

where the range of t is here and later restricted to the natural numbers. This means that observations can be made only at discrete intervals, understood to be on the rise of the relevant clock signal.

Another example of a hardware circuit is the Delay element. On each cycle of operation, the voltage at its output c is the same as the voltage at its input p on the previous cycle of operation:

$$c_{t+1} = p_t.$$

Here, we are unable to predict the initial value c_0, obtained when the hardware is first switched on. The clock event which advances t is communicated to the Delay element by the global clock input signal (marked `C1` in Fig. 1). The correctness of any circuit using this component must not depend on the initial value. The description reflects a certain physical

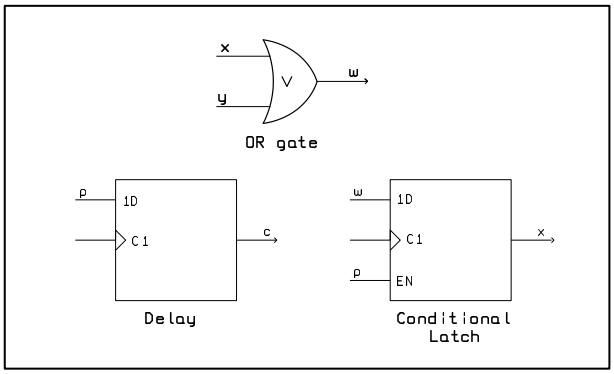

Figure 1.

non-determinism, which cannot be controlled at this level of abstraction. As in the case of engineering tolerance, the specification must also allow for a range of possible outcomes; otherwise correctness will not be provable.

A useful variation of the Delay element is the Conditional Latch element (in engineering terms this is an edge-triggered flip-flop with a clock enable input). Here, the value of the output is changed only when a clock event occurs and the input control wire p is high; and then the new value of x is taken from the other input wire w. Otherwise the value of x remains unchanged. This behaviour is formally described:

$$x_{t+1} = (w_t \lhd p_t \rhd x_t),$$

where $a \lhd b \rhd c$ is read as "a `if` b `else` c".

A pair of hardware components is assembled by connecting output wires of each of them to like-named input wires of the other, making sure that any cycle of connections is cut by a Delay. Electrical conduction ensures that the value observed at the input ends of each wire will be the same as that produced at the output end. As a result, the combined behaviour of an assembly of hardware components is described surprisingly but exactly by a conjunction of the descriptions of their separate behaviours.

For example (Fig. 2), an assembly consisting of the OR-gate, the Delay element and the Conditional Latch is described by

$$w_t = x_t \vee y_t \text{ and } c_{t+1} = p_t \text{ and } x_{t+1} = (w_t \lhd p_t \rhd x_t).$$

The purpose of the internal wire w is solely to carry information from the OR-gate to the Conditional Latch. Its existence can therefore be concealed by existential quantification, with beneficial simplification of the behavioural description:

$$x_{t+1} = ((x_t \vee y_t) \lhd p_t \rhd x_t) \text{ and } c_{t+1} = p_t.$$

The descriptive and deductive power of logical conjunction has been illustrated by two examples, one at the highest level of system requirement capture, and one at the lowest level of hardware implementation. In principle, the conjunction of the descriptions of all the hardware components of the system should imply the conjunction of all the requirements originally placed upon the system as a whole. In a simple system, this implication may be

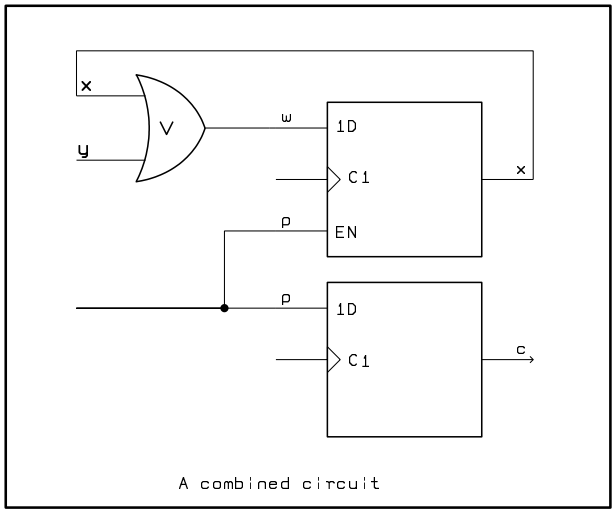

Figure 2.

proved directly; otherwise it is proved through a series of intermediate design documents, perhaps including a program expressed in a high level language. A program also must be interpreted as an indirect description of its own behaviour when executed. We therefore need names to describe its observable features, and for reasons of our own we have chosen to reuse the names of the hardware wires.

Let p_t be an assertion true at just those times t when execution of the program starts, and let $c_{t'}$ be true at just those times t' at which it terminates. Let x_t be the value of a program variable x at time t, so $x_{t'}$ is the final value. With a slight simplification the assignment statement

```
x := x OR y
```

can now be defined as an abbreviation for

$$p_t \Rightarrow \exists t' \geq t.\ (x_{t'} = x_t \vee y_t) \text{ and } c_{t'}.$$

If the program starts at time t, then it stops at some later time t'; and at that time the final value of x is the disjunction of the initial values of x and y. Note that the execution delay $(t'-t)$ has been left deliberately unspecified. This gives design freedom for implementation in a variety of technologies, from instantaneous execution (achieved by compile time optimisation) to the arbitrary finite delays that may be interposed by an operating system in execution of a timeshared program.

Now we confess why we have chosen the same names as the hardware wires. It demonstrates immediately that our example hardware assembly is also a valid implementation of the software assignment: the description of one of them implies the description of the other. The proof is equally simple: take the software termination time as exactly one hardware cycle after the start.

It is the translation of both hardware and software notations into a common framework of timed observations that permits a proof of the correctness of this design step as a simple

logical implication, thereby closing the intellectual and notational gap between the levels of hardware and software. Our example has been artificially simplified by use of exactly the same observation names at both levels. In general, it may be necessary to introduce a coordinate transformation to establish the link between them.

In principle, proof of correctness of a design step can always be achieved, as we have shown, by expanding the abbreviation of the relevant notations. But for a large system this would be impossibly laborious. What we need is a useful collection of proven equations and other theorems expressed wholly in the abbreviated notations; it is these that should be used to calculate, manipulate, and transform the abbreviated formulae, without any temptation or need to expand them. For example, the power of matrix algebra lies in the collection of equational laws which express associative and distributive properties of the operators:

$$A \times (B \times C) = (A \times B) \times C$$
$$(A + B) \times C = (A \times C) + (B \times C).$$

The mathematician who proves these laws has to expand the abbreviations, into a confusing clutter of subscripts and sigmas.

$$\Sigma_j A_{ij} (\Sigma_k B_{jk} C_{kl}) = \Sigma_k (\Sigma_j A_{ij} B_{jk}) C_{kl}.$$

But the engineer who uses the laws just does not want to know.

Fortunately, programming notations have been proved by mathematicians [2,3] to enjoy algebraic properties just as simple and useful as those of matrix algebra. For example, sequential composition of program statements is associative like multiplication; and it distributes leftwards into conditionals. In the occam language these facts are expressed in quite unmathematical notations (Fig. 3).

Even without mathematical proof, an engineer can check quite easily that the two sides of these equations describe exactly the same possible sequences of execution of the commands `P, Q` and `R`.

3 Software

We have already given an example to show the power of observational reasoning on a small scale to prove the correctness of a simple circuit as a hardware implementation of a simple assignment statement of a programming language. To illustrate the power of algebraic reasoning, we will use it to derive a much more elaborate software implementation of the same assignment statement

```
x:= x OR y,
```

translating it step-by-step into the machine code of a simple computer.

A high-level language uses symbolic names (eg. `x` and `y`) for variables ranging over a certain finite range of integers. An actual computer stores these values in a random access memory (`ram`) at distinct locations, say 11 for `x` and 9 for `y`. Since we want to use occam throughout our calculations, we represent the random access memory as an integer array, with (say) 8192 locations :

```
[8192] INT ram:
```

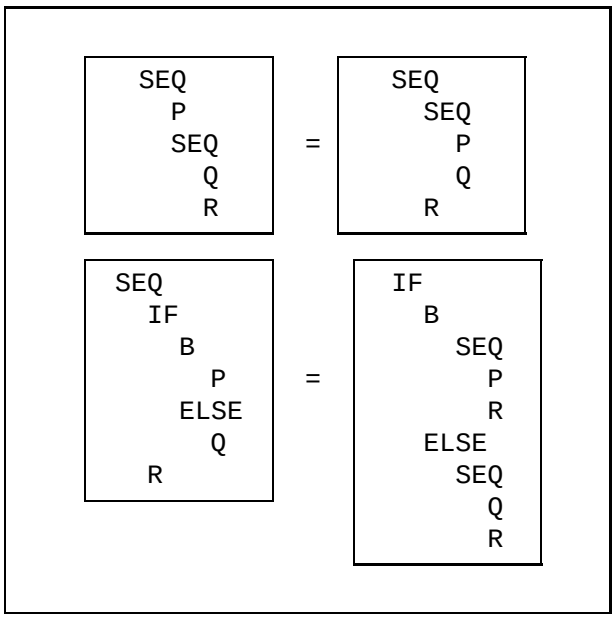

Figure 3.

An actual computer can access and change locations in this memory, for example by an assignment :

```
ram [11] := ram [11] OR ram [9]
```

but it cannot access the high level variables `x` and `y`. We therefore need an abstraction function (data refinement), showing how the abstract variables `x` and `y` are represented concretely in `ram`. Fortunately this too can be expressed in occam as an initial assignment which copies the values of `x` and `y` into the allocated locations before the program starts:

```
ram [11], ram [9] := x,y.
```

Similarly, the values `x` and `y` need to be copied back after the program finishes

```
x,y := ram [11], ram [9].
```

Although expressed in occam, these initial and final assignments are purely conceptual, and will not be executed in practice.

```
    x := x OR y
⇐   [8192] INT ram:
    SEQ
      ram [11], ram [9]:= x, y
      ram [11] := ram [11] OR ram [9]
      x, y := ram [11], ram [9].
```

This is the first of a series of transformations conceptually or actually made by a compiler. It is formulated as an implication whose proof is simultaneously a proof of the correctness of the transformation itself.

The next phase of translation concentrates on the essential assignment statement taken from the middle of the code

```
ram [11] := ram [11] OR ram [9].
```

In a conventional single-address machine code, this has to be split into three instructions, each of which refers to a general purpose register called A. Its sole purpose is to carry information between the instructions. It is therefore declared as a local variable, whose existence can be concealed by existential quantification in the same way as local wire w in the previous hardware example.

```
INT A:
SEQ
  A := ram [11]
  A := A OR ram [9]
  ram [11] := A.
```

Again, an algebraic calculation similar to symbolic execution shows that the mathematical meaning of the assignment is unchanged, so that this translation is also proved to be correct.

The next task is to implement the sequencing construction SEQ, ensuring that the three assignments are executed properly in the right order. In a conventional computer, sequencing is implemented by a program pointer (called p), which in this case steps through four values, say 20, 21, 22 and 23. The value of p selects which of the three instructions is to be executed, and at the same time p itself is incremented by one

```
INT p :
SEQ
  p := 20
  WHILE (20 <= p) AND (p < 23 )
    IF
      p=20
        A, p := ram [11], p+1
      p=21
        A, p := A OR ram [9], p+1
      p=22
        ram [11], p := A, p+1
  -- p=23.
```

The last line is an assertion, ensuring that the code ends decently by falling off the end, rather than by a wild jump. Such assertions do not have to be executed; we will see that they are an essential aid to reasoning about designs and their correctness.

The purpose of the design so far is to ensure that each of the clauses of the IF statement describes exactly the effect of execution of a single machine code instruction available on the target computer. Of course, in practice each instruction is represented as a bit-pattern, which packs together the operation code for the instruction and the address. Fortunately the execution of the instruction can again be accurately and conveniently be described in

occam itself, by a conventional interpreter or simulator for the machine code:

```
PROC do (VAL INT (instruction, address))
  CASE instruction
    0
      A, p := ram [address], p+1
    1
      ram [address], p := A, p+1
    7
      A, p := A OR ram [address], p+1
    ...
    ...
```

An interpreter like this is often decreed as a definition of the architecture of a computer, and is accepted by hardware designers as a formal specification document for their design. We will use it for the same purpose later.

The final transformation implements the basic idea of the stored program computer. The program store is represented by an occam array, say with 64 instructions

```
[64] INT code :
```

The action of the program loader is represented by an assignment of the initial values to this array:

```
code[20], code[21], code[22] := (0,11), (7,9), (1,11)
```

The program starts by initialising the sequence register, which is then used repeatedly as an index to fetch the successive instructions from the code array and execute them. The combined process of loading and running is described by

```
[64] INT code:
SEQ
  code[20],code[21],code[22]:=(0,11),(7,9),(1,11)
  p:= 20
  WHILE (20 <= p) AND (p < 23)
    do (code[p])
  -- p=23.
```

Thus we reach the end of a long and complex series of transformations of a single simple assignment statement of a high level language into a complex structure of statements expressed in the same high level language. The only consolation for the length of the transformation process is its modularity. Each transformation of the series has separately introduced a separate architectural feature

1. random access memory and symbol tables
2. registers and single address instructions
3. sequence control
4. numeric encoding of instructions
5. the stored program concept.

This kind of unravelling and separation of concerns is an essential goal in engineering method. Careful modular structuring of the theory is essential to allow the same theorems to be reused on a variety of source languages and different target architectures, including direct implementation in hardware.

The same principle of abstraction can be maintained in translation of the structural features of the language, for example `SEQ, IF` and `PAR` in occam. These are translated or rather eliminated by the algebraic technique of reduction to normal form. We define a normal form as an iterated conditional, where all the conditions are disjoint; when they are all false, the iteration terminates:

```
SEQ
  p:= k
  WHILE (k <= p) AND (p < l)
    IF i=k FOR l-k
      p=i
        Q_i
  -- p = l.
```

If all the Q_i can be represented as binary machine code instructions, it is fairly obvious that any complete program in normal form can be further translated into code for execution in a stored-program computer, using a loader and interpreter. Let us introduce the abbreviation `[k, Q, l]` to stand for this normal form.

The remaining task is to show how to reduce every program of the source language into normal form. First, all the primitive expressions of the language are reduced (or more usually expanded) into the desired normal form; we have already shown how to do this for the assignments which are the ultimate components of a conventional program.

```
    SEQ
      [k, Q, l]
      [l, R, m]
⇐   SEQ
      [k, (Q R), m]
```

Figure 4.

The next step is to eliminate all the composition operators of the language, one by one in a bottom-up fashion. The operands can therefore be assumed already to be in normal form; what is needed is that the result of elimination should also be in normal form. The algebraic laws used for this kind of elimination usually have the shape

$$NF1 \odot NF2 \Leftarrow NF3$$

where $\odot$ is the operator to be eliminated, and all three operands are in normal form, which does not contain $\odot$. Repeated application of such laws from left to right will eventually eliminate all occurrences of the given operator, leaving just a single normal form, which by transitivity implements the original program.

An example of a reduction law for our compiler is one that eliminates sequential composition (Fig. 4). On the left hand side, the assertion guarantees that the first operand of `SEQ` leaves the sequence register at exactly the value `l` at which the second operand

expects to start. The second initialisation can therefore be omitted, and so can the assertion; the resulting code is just the concatenation of the codes `Q` and `R` contributed by the two operands. The proof of this particular normal form theorem is quite elegant, and revealed some surprising mathematical insights.

A law for the elimination of a conditional is given in (Fig. 5). It assumes availability in the machine code of a conditional jump

```
p := (p+1 ◁ A ▷ l+1)
```

and an unconditional jump

```
p := m.
```

The first of these is planted at location `k` in the code memory, and the second at location `l`.

```
      IF
        A
          [k+1, Q, l]
        ELSE
          [l+1, R, m]
⇐     [k,
       (p=k
          p := (p ◁ A ▷ l) + 1
          Q
        p=l
          p := m
          R),
        m]
```

Figure 5.

All the laws that we have illustrated on a particular program can be generalised to apply to all programs expressed in the language. Each law can be proved algebraically from simpler laws of programming, and each proof can be checked individually by a computer algebraic system like OBJ3. Even better, each theorem is itself an algebraic transformation that can be directly executed by OBJ3. The result is an automatic general-purpose compiler that has been constructed as a byproduct of its own proof of correctness, thereby achieving the goal set by Dijkstra [4] many years ago. A prototype implementation of this philosophy has been explored by Augusto Sampaio in his recently completed Doctoral project at Oxford [5]. The technology is being further developed for a significantly larger subset of occam by a team of researchers in Kiel, under the leadership of Hans Langmaack.

4 Hardware from Software

The same philosophy and very similar techniques can close the gap which remains between the machine code produced by the compiler and the actual hardware circuits of the machine which executes the code. The intended behaviour of the machine has already been conveniently specified by an interpreter, written in the same high level source language. So the obvious solution is to translate this interpreter into circuit notations that can be

implemented directly by gates and flip-flops printed onto the surface of a silicon chip. Such a translator [6] has been constructed by one of the authors (IP). For reasons of efficiency, it slightly generalises the parallel constructs of occam, and uses a global and discrete model of time [7]. It has been used successfully [8] for automatic synthesis of microprocessors.

In many ways this translation is much more direct than translation to machine code. The global variables of the program denote registers that are implemented directly as flip-flops, whose number and size are determined by the needs of each individual program. So there is no need for a symbol table of numbered locations in an external `ram`. Similarly, there is no fixed function unit or instruction code: hardware is allocated to implement exactly the assignments and expressions of the source program. Except in a microcoded architecture, there is no stored program and no interpreter. All these phases of conventional compilation can be omitted, leaving as the main task just the reduction to normal form, and the new and difficult task of component layout and routing of wires.

The normal form for hardware is structurally and conceptually the same as for software. But since there is no code storage the conditional testing the value of the `p`-register has to be distributed through to the expressions on the right hand sides of the assignments, which must then be merged. The relevant transformations are governed by the law

```
IF
  p = 20
    A, p := e, f
  ELSE
    A, p := g, h

=  A, p := (e, f) ◁ (p=20) ▷ (g, h).
```

The body of the loop is now a single assignment with a rather complicated expression on the right hand side. But in principle it can be implemented in the manner illustrated in section 2.

In practice, a final optimisation must be made. The test `(p=20)` and the addition `(p:=p+1)` are quite expensive to implement directly in hardware. The solution is to use only powers of two as values of `p`. Thus each test only has to test a single bit of the register, and incrementation is replaced by left shift, using a Delay element exactly as illustrated in section 2. This simple change of representation removes a major disadvantage of hardware over software implementation of normal sequential programs.

But the main advantage of hardware is that the hardware components operate naturally and automatically and without overhead in parallel with each other; indeed, it is sequential execution in hardware that requires careful organisation of control signals and other overhead. Fortunately, the parallel constructions of occam are designed at a sufficiently high level of abstraction to be translatable efficiently in both hardware and in software. The advantage of this abstraction is not just conceptual: the real practical pay-off is that a complete embedded system can be designed and implemented in the same programming language; and later, certain parts of it can be designated for translation into special purpose hardware, and other parts into machine code for execution on an attached general-purpose computer. In this way the gap between hardware and software has been safely closed, and there is no longer any possibility of errors congregating in the traditionally error-prone interfaces between them.

As an example of such a computation we consider part of an interface from a robot

position-measuring system [9]. The code shown here is extracted from an executable program which simulates the behaviour of a hardware interface that was required to link a set of shaft encoder devices to a controlling computer. This fragment reads a two-bit code from a shaft encoder and alters a variable depending on whether the shaft has rotated by one unit.

```
SEQ
  encoder ? current
  IF current  ≠ previous
    THEN
      IF current BIT 0  ≠ previous BIT 1
        THEN
          angle := angle - 1
        ELSE
          angle := angle + 1
  previous := current
```

This program could be compiled and executed on a conventional microprocessor, but it would be perhaps a hundred times slower than was required for our application. It is thus necessary to have an implementation in special-purpose hardware whose behaviour has been captured by our program. We therefore proceed by transforming it into normal form:

```
SEQ
 start,c0,c1,c2,c3,c4,finished := 1,0,0,0,0,0,0
 WHILE ¬ finished
  PAR
    previous := current ◁ c2 ∨ c3 ∨ c4 ▷ previous
    current  := encoder ◁ (start ∨ c0) ∧ encoder_rdy ▷ current
    angle    := (angle+1 ◁ incr ▷ angle-1) ◁ decr ∨ incr ▷ angle
    c0       := (start ∨ c0) ∧ ¬ encoder_rdy
    c1       := (start ∨ c0) ∧ encoder_rdy
    c2       := decr
    c3       := incr
    c4       := c1 ∧ ¬ changed
    finished := c2 ∨ c3 ∨ c4
    start    := 0
 WHERE
    bitdiff  = current BIT 1  ≠ previous BIT 0
    changed  = current  ≠ previous
    incr     = (c1 ∧ changed) ∧ bitdiff
    decr     = (c1 ∧ changed) ∧ ¬ bitdiff
```

Our normal form program consists of a single parallel assignment (written here with `PAR`) embedded in a `WHILE` loop. This program captures the parallelism available in the original program, and was generated automatically from the original. It uses the control state variables (`c0..c4, start`, and `finished`) to implement the ‘distributed program counter’ optimisation mentioned previously; this is more commonly known by hardware

engineers as 'one-hot' control encoding. In practice, we would do one further transformation on the normal form program to replace the `start` variable with another which is its logical inverse. In this way, the initialisation step ensures all control variables are 0, and conveniently this facility is directly supported by the hardware we use to implement our circuits. No optimisations have been applied to this normal form program although a number are possible.

A normal form program such as this one can now be interpreted directly as a hardware circuit. Everything on the left hand side of the parallel assignment can be interpreted as hardware storage devices (flip-flops), and everything on the right hand side can be interpreted as a set of logic gates. In all but syntax, this program is the circuit diagram of hardware we can implement. The interpretation of this program as a circuit is shown in Fig. 6 for comparison.

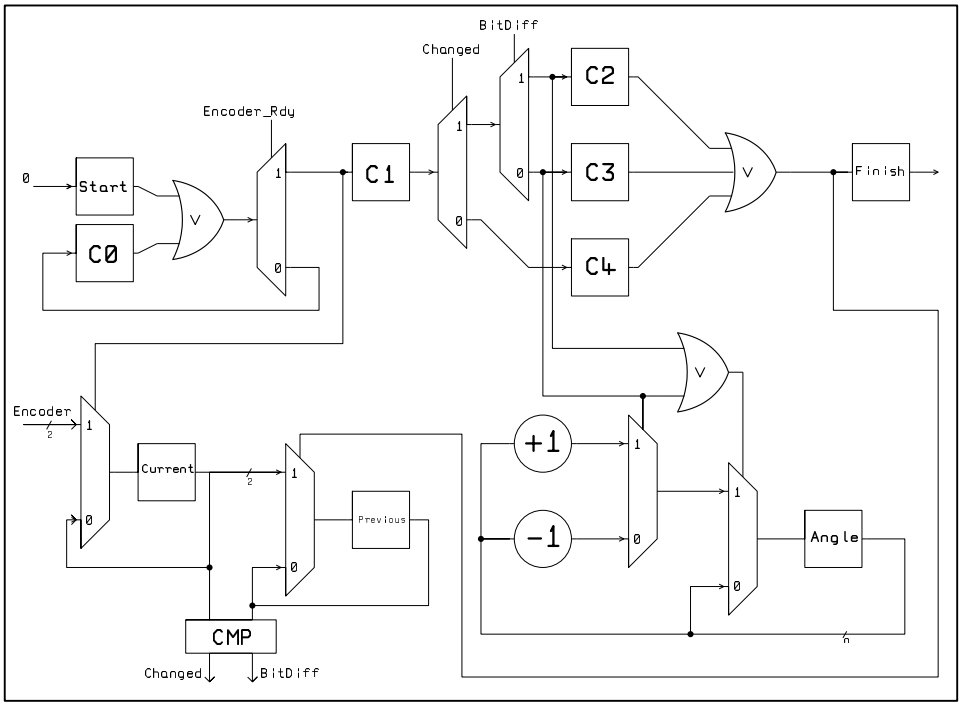

Figure 6.

In the circuit diagram, the abstract variables of the program are shown as flip-flops; for clarity, the global clock input has been supressed and state is maintained over clock events by explicit feedback of current state rather than by the use of a clock-enable inputs. The conditional expressions are implemented with multiplexors, and some of the boolean combinations of control variables are implemented with decoders in order to reduce the number of symbols in the diagram. The hardware to generate the comparison signals has also been hidden to simplify the diagram.

The normal form program, when interpreted as hardware, proceeds with the parallel assignment being executed once each clock cycle. The number of clock cycles for the program to complete represents the sequentiality of the program and the complexity of the

assignment statement represents its parallelism. Using the algebraic laws, we can trade off the sequential and parallel aspects against each other to change the cost/performance ratio for the hardware. This is typically, but not exclusively, accomplished by transformation of the original source program before conversion to normal form.

5 Hardware/Software Co-Design

Since we have encompassed software and hardware in the same theoretical framework, it is possible to develop implementations of programs that are realised partly in hardware and partly in software. We can also use this framework to trade off the hardware and software components against each other to achieve desirable cost/performance measures. We have developed a number of small-scale systems using our hardware compiler to produce the hardware components, and the Inmos occam2 compiler to produce the software components. We hope at some future date we will be able to compile complete hardware/software applications from a single source text, when we have solved the remaining problems of bridging the gap between these two closely-related languages.

Making it easier for programmers to produce circuits corresponding to their programs would be of little use unless there were some way of building these hardware circuits easily. Fortunately, there is a newly-available technology which does exactly this.

The Field Programmable Gate Array, or FPGA, is a relatively new form of integrated circuit which can be programmed to act as almost any digital circuit, subject to the physical limitations of its size. The type of FPGA we use is essentially a gate array whose programmable elements are all implemented with static RAM. Changing the hardware configuration is accomplished by writing an appropriate set of bits to the RAM.

The essential difference between this and previous hardware implementation technologies, is that the hardware can be reprogrammed rapidly by a purely software process. Today's FPGA chips can each implement a circuit with an equivalent complexity of some 20,000 gates in a matter of milliseconds. Even one of these chips is capable of hosting a small application program kernel and the size and speed of FPGAs is increasingly rapidly as a result of progress in VLSI technology. Their use allows design decisions to be deferred, or implementations to be upgraded while in service. These advantages have already made FPGA sales the fastest-growing segment of the digital integrated circuit market. It was the emergence of this technology which directly stimulated our interest in hardware compilation as a research topic.

However, many current uses of FPGAs ignore one of their most interesting characteristics. In a typical application, one or more FPGA chips will be configured on system power-up and will remain near-permanently in that state. The possibility of reconfiguration during system operation holds out the possibility that tomorrow's computing systems can use such reprogrammble hardware as a flexible resource which can be deployed to support whatever computation is running at the time. Thus, reconfigurable hardware may bring benefits similar to those offered by 'reconfigurable software' in its usual guise of the stored-program computer.

In order to explore hardware/software co-designs, we have designed and built the HARP board. This is a small, 17 x 9 cm, printed circuit board which conforms to the Inmos TRAM standard (Fig. 7). It is a daughter-board which can very simply be integrated with commercially available host boards for many computers, and with a wide variety of other parallel computing and input/output modules. These modules can readily be joined

together to make larger systems. Our HARP module consists of two intimately-connected

Photograph of HARP board

Figure 7.

computing systems; one is based on a transputer, and the other on a large Xilinx FPGA. The transputer could be replaced by any other fast, 32-bit microprocessor, but it is convenient for us because it supports parallelism in general, and the occam language in particular. Both systems have their own local memory and can run completely independently if required. However, they also share the same 32-bit bus, so that data exchange between them is fast and efficient. Some algorithms may be implemented with deep combinational hardware, such as exhibited by a parallel multiplier; others may be heavily pipelined with only tiny amounts of logic between the registers, such as in a systolic signal-processor. Consequently, it is necessary to operate the FPGA at different speeds, depending on the architectural style of our algorithm implementation. Thus the HARP board also has a 100MHz frequency synthesiser, controlled by the transputer, which can generate an arbitrary clock signal. The FPGA can be reconfigured by the transputer so that at one instant it might be supporting graphics or image-processing algorithms, and at the next it might be supporting data compression, spell-checking, or pattern matching.

As one example of the system implications of FPGA technology closely coupled with a conventional microprocessor, we briefly mention memory sharing on our HARP board. Since the FPGA chip has full access to the transputer bus and also its own local memory,

it is simple to load the FPGA with a configuration which makes the local FPGA memory appear in the address space of the transputer. By loading this 'memory-map configuration', a transputer-based application can process its data, leaving the results in FPGA memory. A computing configuration can then be loaded into the FPGA, leaving it to process the data further. When the FPGA program has terminated, it can inform the transputer which then reloads the memory-map configuration to gain access to the data. We had originally planned that communications between the transputer and the FPGA would always be mediated by occam-style channels implemented by transfers over the bus. It was a pleasant discovery that this additional mode of operation was available to us. We take it as indicative of the greatly increased flexibility that reconfigurable logic imparts to systems which use it.

In our HARP system, the FPGA subsystem is perhaps best regarded as a flexible co-processor. Once the transputer has loaded a computing configuration into the FPGA, they may be equal partners in the computation, but unlike a normal co-processor, the hardware in the FPGA can be changed completely at any instant. If this style of flexible hardware/software implementation becomes widely used, we can also expect to see a corresponding development of microprocessor architecture in which a substantial amount of reconfigurable hardware is embedded in the microprocessor core. Indeed we regard the whole HARP board itself as a prototype of a future generation of microprocessors. We are currently investigating the implications of FPGA and hardware compilation technologies on computer and system architectures in a joint proect with Inmos.

In coupling FPGA technology with hardware compilation, we are trying to provide environments where programmers, as well as hardware engineers, can implement their programs to take advantage of the greater speed offered by hardware implementations. In a recent example, a programmer in our group having little understanding of hardware, has produced a small, elegant, spelling-checker in hardware that runs more than 25 times faster than a similar implementation of the same algorithm on a transputer. We could not pretend that the development of this application was nearly as easy or straightforward as we would like, but we have demonstrated feasibility, and it gives us hope that our main goals ultimately will be achieved.

One use of FPGA chips which we are beginning to investigate is the use of on-the-fly reconfiguration, for which we might use the term 'virtual hardware' by analogy with virtual memory. In this scenario, an application runs as far as it can with whatever hardware is loaded. When it can run no further, the environment swaps in further hardware on demand to enable processing to continue. The occam SEQ constructor provides the natural structure boundary for partitioning algorithms to work in this way. We also expect our theoretical models and proof techniques to extend to such systems.

A particular application area which is attracting our interest is that of multi-media applications. Such applications are characterised by (i) a wide variety of information representations, many of them needing large amounts of data storage, with consequent emphasis on compression and efficient data transfer, (ii) computationally-intensive conversion programs which can extract high-level information from one domain of data and inject it into another, and (iii) interfaces with a wide variety of input/output devices. The reconfigurable FPGA can lend support in each of these areas. As one example, the accuracy and detail of graphics rendering algorithms embedded in FPGAs could be traded off against throughput if the system became overloaded due to a request for a high-speed animation sequence. Similarly, the same physical FPGA hardware could be supporting speech synthesis at one moment and video compression at the next. The occam CASE constructor provides a natural

way of exploiting this style of reconfiguration.

We have looked at a number of applications of hardware compilation coupled with implementation via FPGA technology. Some of these have been speculative; others are a practical reality already in our laboratory and are being actively transferred into industrial practice. Our goal is to use these devices and our hardware/software compilation techniques to build flexible and dependable computing systems. We look forward to building a system in the future where users write programs or invoke applications, and literally have no idea whether the implementation generated is in software, in hardware, or in some judicious mixture of the two; the compilation system will have ensured that the user's high-level constraints on cost and performance are met as closely as possible by the implementation.

6 Conclusions

The gap between hardware and software is not the only gap that needs to be closed by further theoretical and practical investigations. We began with a summary of a number of the phases in the design of a complex engineering product, like a real-time process control system. It is our hope that the earlier stages of design, including specification and analysis of requirements, can be assisted by a systematic transformational approach that precludes the possibility of error; and that the gaps between all the phases may be securely closed in the same way as we have shown for the gap between hardware and software.

REFERENCES

[1] C.A.R. Hoare, *Mathematical Logic and Programming Languages*, chapter 'Programs are Predicates', 141–154, Prentice-Hall, 1985.
[2] C.A.R. Hoare et al., 'Laws of programming', *Comm. ACM*, **30**(8), 672–686, (1987).
[3] A.W. Roscoe and C.A.R. Hoare, 'Laws of occam programming', P.R.G. Monograph, Oxford University Computing Laboratory, (1986).
[4] E.W. Dijkstra, 'A constructive approach to the problem of program coirrectness', *BIT*, **8**, 174–186, (1968).
[5] He Jifeng, C.A.R. Hoare, and A. Sampaio, 'Normal form approach to compiling specifications', *Acta Informatica*, (1994). to appear.
[6] Ian Page and Wayne Luk, 'Compiling occam into FPGAs', in *FPGAs*, 271–283, Abingdon EE&CS Books, (1991).
[7] Michael Spivey and Ian Page, 'How to Program in Handel', Technical Report, Oxford University Computing Laboratory, (1993).
[8] Ian Page, 'Parametrised Processor Generation', in *FPGAs 93*, to be published by Abingdon EE&CS Books (probably), (1993).
[9] Ian Page and Wayne Luk and Henry Lau, 'Hardware Compilation for FPGAs: Imperative and Declarative Approaches for a Robotics Interface', in *Proc. IEE Colloquium on Field-Programmable Gate Arrays – Technology and Applications, Ref. 1993/037*, pp. 9.1–9.4. IEE, (1993).