Downloaded from rsta.royalsocietypublishing.org

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Designing Chips that Work [and Discussion]

David May, Geoff Barrett, David Shepherd, W. A. Hunt and E. M. Clarke

Phil. Trans. R. Soc. Lond. A 1992 **339**, 3-19
doi: 10.1098/rsta.1992.0022

Email alerting service

Receive free email alerts when new articles cite this article - sign up in the box at the top right-hand corner of the article or click **here**

To subscribe to *Phil. Trans. R. Soc. Lond. A* go to:
http://rsta.royalsocietypublishing.org/subscriptions

This journal is © 1992 The Royal Society

Downloaded from rsta.royalsocietypublishing.org

Designing chips that work

By David May, Geoff Barrett and David Shepherd
Inmos Ltd, 1000 Aztec West, Almondsbury, Bristol BS12 4SQ, U.K.

The complexity of integrated circuits continues to grow, and chips with over 10^8 transistors will be in widespread use by the late 1990s. These chips will combine general purpose processors with subsystems for communications and other specialized tasks. They will be far too complex for the design to be tested, and manufacturing volumes will be far too high for the design to be wrong!

Mathematical techniques have already been applied to the design of parts of VLSI chips. Most of this work is experimental, and requires an unusual combination of engineering, mathematical and programming skills. Sometimes new theoretical work is needed, and specialized tools may have to be constructed. Despite these difficulties, mathematical techniques are playing an important role in the design of microprocessors at Inmos, and techniques suitable for incorporation in standard computer-aided design systems are emerging.

1. Introduction

The idea of using formal program specification and proof techniques in an Inmos product first arose at the Royal Society, in a Discussion Meeting on 'Mathematical Logic and Programming Languages' (Hoare & Shepherd 1985). One of the lectures concerned the Gypsy program verification system, and sparked off a discussion about whether program verification was practical, in view of the extra time needed over the 'standard method' of program design. The lecturer, D. Good, suggested that verified programming took about five times as long, and tried to justify the extra time spent to a rather sceptical audience.

At the time of the meeting, we had become concerned about a rather short program which had taken very little time to write, but was taking a long time to test. We had extended our programming language, Occam, so as to include floating point operations. As there was no floating point hardware in the transputer, we had written a small subroutine package to provide the necessary operations. The package was not large, but some complexity had arisen from the need to support the emerging IEEE floating point standard. The package was being tested by executing all of the arithmetic operations, with all rounding modes, on a vast number of operand values, comparing results with a reference implementation.

It seemed that it would be worth trying to develop a formally verified version of the package. Even if this were to take ten times as long as the original package had taken, it would eliminate the time-consuming testing, and reduce the overall length of time. At the time, we had no clear idea how to proceed with the verification, but we discussed the idea with several members of the Oxford University Programming Research Group. The result was that Geoff Barrett (then working at the PRG, now at Inmos) re-expressed the IEEE standard in Z, and developed a version of the package from the Z specification.

Phil. Trans R. Soc. Lond. A (1992) **339**, 3–19
Printed in Great Britain
© 1992 The Royal Society

The Royal Society is collaborating with JSTOR to digitize, preserve, and extend access to *Philosophical Transactions: Physical Sciences and Engineering.* www.jstor.org

Downloaded from rsta.royalsocietypublishing.org

The formally derived package was completed whilst the testing of the original package was still in progress. We were pleased about this for two reasons. Firstly, it demonstrated a much more reliable and cost-effective technique for developing algorithms for computer arithmetic. Secondly, we had become concerned about the rapidly approaching problem of verifying the *hardware* floating point unit of a new transputer, the T800.

2. The T800 floating point unit

The T800 floating point unit consists of a datapath controlled by microcode. The microcode includes intricate algorithms for floating point multiplication and division, and handles a large number of special cases resulting from denormalized numbers and infinities.

It is common practice to check the design of floating point arithmetic units by simulation. This has the same problems as testing a software package, and is more time-consuming because a simulator executes the tests much more slowly than software compiled and executed directly. For this reason, bugs in floating point arithmetic units are widespread.

In the design of the floating point unit of the T800, the formally derived Occam floating point package was used as a high-level implementation. A low-level implementation, also written in Occam, was used to describe the detailed operation of the hardware of the floating point unit. The high-level and low-level implementations were then compared by using Occam transformations. By transforming one representation into the other, the two representations were shown to be equivalent. At the time of the T800 design, an Occam transformation system (Goldsmith 1988) was already under development at Oxford University, and this was used to mechanize these transformations.

The various stages of simple development used are shown in the next section as an example. For the example we consider the development of microcode that implements the conditional jump, *cj*, instruction on the current transputer. The development described here follows the same path as was used for parts of the IMS T800 floating point unit microcode.

(*a*) *High level specification*

We first introduce the variables representing the processor state:

CPU

$$Areg, Breg, Creg : WORD$$

$$Iptr : WORD$$

This schema introduces names for the values which define the state of the CPU. The names *Areg*, *Breg*, *Creg* denote the values in a stack of registers, and *Iptr* denotes the value of the instruction pointer.

Downloaded from rsta.royalsocietypublishing.org

The conditional jump instruction is specified by the following schema:

CJ

$$\Delta CPU$$
$$Operand? : WORD$$

$$Areg = 0 \Rightarrow Iptr' = (Iptr + 1) + Operand?$$
$$Areg \neq 0 \Rightarrow Iptr' = Iptr + 1$$
$$Areg' = Breg$$
$$Breg' = Creg$$

This schema specifies how the CPU state changes as a conditional jump is executed. In the upper part of the schema, ΔCPU denotes that the CPU state changes. By convention, the values of the state variables are represented by *Areg*, *Breg* and *Iptr* before the operation and by *Areg'*, *Breg'* and *Iptr'* after the operation. The instruction operand is denoted by an input to the schema, *Operand?*.

The bottom portion of the schema consists of predicates which constrain the values of the state variables. The predicates here state that if the value of *Areg* is 0 (the machine representation of false) then the jump is taken, using the *Operand?* as an offset of the next instruction. In this case, the final value of the register stack is unspecified. If the value of *Areg* is not 0 then no jump is taken and the register stack is popped up one place.

(*b*) *Occam transformations*

The algebraic semantics of Occam given in Roscoe & Hoare (1988) consists of a set of laws which define the language constructs. The algebraic semantics has been shown to be consistent with the denotational semantics (Roscoe 1985) establishing the validity of the laws. The set of laws is complete so any pair of equivalent programs can be transformed into one another using them.

A transformation law can be used to transform one program into another whose observable behaviour is equivalent. Many transformation laws are 'obviously true' and are regularly used by programmers; for example sequential composition of processes is associative:

```
SEQbinassoc:
SEQ          SEQ
  P            SEQ
  SEQ    ≡       P
    Q            Q
    R          R
```

This is the law *SEQbinassoc*. Others are more complex and include preconditions for validity but, with a bit of effort, can be seen to be true.

If a sequence of transformations can be found to transform one program into another then the two programs are known to be equivalent. If, in addition, one of these programs is known to be a correct implementation of a specification then the correctness of the other can be inferred.

Downloaded from rsta.royalsocietypublishing.org

(i) *An example transformation*

As an example consider the following program fragment

```
SEQ
  X := A
  Y := Y+X
```

These two assignment statements can be merged into one multiple assignment statement. First the law *ASid* is used to add an identity assignment to each statement.

ASid:
$\underline{x}, \underline{y} := \underline{e}, \underline{y} \quad \equiv \quad \underline{x} := \underline{e}$
for any $\underline{y}$ distinct from $\underline{x}$ and assignable in the current scope.

giving the program:

```
SEQ
  X, Y := A, Y
  Y, X := Y+X, X
```

Next the law *ASperm* is applied to the second statement

ASperm:
$$<x_i \mid i = 1 \ldots n> := <e_i \mid i = 1 \ldots n>$$
$$\equiv$$
$$<x_{\pi_i} \mid i = 1 \ldots n> := <e_{\pi_i} \mid i = 1 \ldots n>$$
for any permutation π of $\{1 \ldots n\}$

giving

```
SEQ
  X, Y := A, Y
  X, Y := X, Y+X
```

Finally these two statements are merged by the law *SEQcomb*

SEQcomb:
SEQ
$\underline{x} := \underline{e} \quad \equiv \quad \underline{x} := \underline{f}[\underline{e}/\underline{x}]$
$\underline{x} := \underline{f}$

giving

```
X, Y := A, Y+A
```

The transformation performed by the sequence of steps given above is the trivial type of transformation that programmers often do automatically and the use of proof rules and a transformation system may seem excessive in this case. However, people often make mistakes with 'trivial' operations and as the programs being manipulated become larger the use of a formal basis for the program transformation becomes vital.

(c) *The Occam transformation system*

To aid the process of transforming programs a simple interactive transformation system has been implemented in the language ML (Goldsmith 1988; Harper *et al.* 1986). A program can be parsed into this system and then manipulated by the user.

Downloaded from rsta.royalsocietypublishing.org


```
VAL cj.entry IS 0 :
VAL cj.true IS 1 :
VAL cj.false IS 2 :
VAL FINISHED IS 3 :
BYTE micro.inst:
SEQ
  micro.inst := cj.entry
  iptr := iptr + 1
  INT  ALUresult :
  BOOL ALUresultZ:
  WHILE (micro.inst <> FINISHED)
    SEQ
      aregslave, bregslave := areg, breg
      IF
        micro.inst = cj.entry
          SEQ
            ALUresult := 0 - areg
            ALUresultZ := (ALUresult = 0)
            IF
              ALUresultZ
                micro.inst := cj.false
              (NOT ALUresultZ)
                micro.inst := cj.true
        micro.inst = cj.true
          SEQ
            areg := bregslave
            breg := creg
            micro.inst := FINISHED
        micro.inst = cj.false
          SEQ
            ALUresult := iptr + operand
            iptr := ALUresult
            micro.inst := FINISHED
```

Figure 1. Low level Occam representation of microcode.

All the basic laws of Roscoe & Hoare (1988) are implemented inside the system along with some extra ones; the system is extensible and new laws (that have been proven correct) can be added if required. Regularly executed sequences of transformations can be programmed as ML functions to give more powerful higher level transformations. The example transformation shown above was programmed as the transformation law *combas* which itself was used in more powerful transformations. The basic transformations often have only a small and localized effect but when suitably combined they can perform significant transformations which, being constructed from correct component transformations, are known to be correct themselves.

The transformation system user can select which transformation laws to apply and examine the effects of these transformations. The fact that the transformation system is being used provides the verification of the equivalence between the initial program and the transformed end result. It would be feasible to modify the system so that it produced the list of transformations which constitute the proof for independent verification. This would allow the transformation that was performed to be provided as part of the completed design.

Downloaded from rsta.royalsocietypublishing.org


```
cj.entry:
   XbusFrom0     YbusFromAreg
   ALUminus
   NextInst      (COND ALUresultZ -> cj.false, cj.true)
cj.true:
   AregFromBregSlave
   BregFromCreg
   OregFrom0
   NextInst      FINISHED
cj.false:
   XbusFromIptr YbusFromOreg
   ALUplus
   IptrFromALU
   OregFrom0
   NextInst      FINISHED
```

Figure 2. Microcode assembler source.

(*d*) *Occam representation of microcode*

The conditional jump instruction can be specified at a high level in Occam as the program:

```
{areg = Areg ∧ breg = Breg ∧ creg = Creg ∧ iptr = Iptr ∧ operand =
        Operand ?}
IF
  areg = 0
    SEQ
      iptr := (iptr +1) +operand
      {areg = 0 ∧ breg = Breg ∧ creg = Creg ∧ iptr = (Iptr+1)+Operand ?}
  areg <> 0
    SEQ
      iptr := iptr+1
      {areg = Areg ∧ breg = Breg ∧ creg = Creg ∧ iptr = Iptr+1}
      areg, breg := breg, creg
      {areg = Breg ∧ breg = Creg ∧ creg = Creg ∧ iptr = Iptr+1}
```

Annotations from the pre/post condition analysis have been added to show that this has the desired effect of taking the jump if the value of `areg` was 0 and popping the top value off the stack otherwise.

The task is now to derive from this 'specification' level Occam an equivalent 'implementation' level program. This derivation is achieved through a sequence of refinements which use the semantics preserving transformations of Occam. The microcode program is shown in figure 1.

This program can be shown to be equivalent to the high level program by a series of transformations. In practice, the number of transformations is large, and in some cases the intermediate programs produced are also large. This means that it is essential that the transformations are performed by computer.

The low-level program can easily be converted into the microcode actually used to generate the microcode read-only memory shown in figure 2. However, it is interesting to note that a bug in the translation program used for this purpose resulted in a fault in an early version of the T800. Our experience has been that it

Downloaded from rsta.royalsocietypublishing.org

is essential either to be able to verify the output of such programs, or to be able to verify the programs themselves.

(e) *Was it effective?*

It is now four years since the T800 was introduced. Two bugs have been discovered in the floating point microcode. One of these arose from an error in the translation program used to convert microcode Occam to microcode assembler. The other resulted from manual 'tidying' of the generated assembler. The use of formal development techniques and program transformation was therefore highly successful as no errors have been found in the areas covered by these techniques.

The main shortcoming of the work performed on the IMS T800 was that the formal machine-assisted correctness process only handled part of the design – the transformation from high level Occam to low level 'microcode Occam'. Reasoning about the correctness of the algorithms used was performed by hand, although in principle such proofs could have been machine-checked. Also, no attempt was made to prove the correctness of the implementation of the hardware used to execute the microcode.

3. Totally verified systems

The successful verification of the T800 floating point microcode led us to speculate about the possibility of verifying other parts of the transputer. We wanted to be able to verify the hardware controlled by the microcode, and indeed the combination of the microcode and hardware. At the same time, others had become interested in verifying programs and compilers, especially for safety critical systems. As a result, in 1988 Inmos, Oxford, Cambridge and SRI initiated a collaborative project to investigate totally verified systems.

We believe that over the next decade the use of formal verification will increase in a wide range of microprocessor applications. The majority of 32-bit microprocessors are currently used in personal computers but it is predicted that this will shift to embedded applications as early as 1993. Some of these applications are high-volume and will involve significant amounts of software. The costs of correcting design faults will therefore be high, and the costs of losing customers (who will not expect to have to reboot a telephone) will be even higher. In addition, many embedded applications will require a combination of a microprocessor and specialized hardware to be integrated on a single chip, giving rise to the need for verification of hardware and software in combination.

The SAFEMOS project aims to provide a demonstration of a totally verified system. Such a system consists of a specification language to capture design requirements, a programming language which is amenable to formal reasoning, a program verifier to verify that a program is an implementation of a specification, a verified processor to run the programs on, and a verified compiler to translate the programming language into the processor instruction set.

Correctness is ensured by first verifying programs against their specifications. Correctness of compiled code is ensured by implementing the compiler algorithm inside a theorem prover; this means that compilation consists of constructing a proof that the compiled code is correct. This approach avoids the danger of having a correct compiler algorithm incorrectly implemented. Finally the correct execution of the compiled code is ensured by a verification of the processor. All the verification work is done inside the HOL theorem prover (Gordon 1986).

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES
PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Downloaded from rsta.royalsocietypublishing.org

An emphasis has been placed on mechanical verification for two reasons. The first is that, from experience, there is a real danger of errors in hand verification; often when attempting to prove an obvious 'theorem' in HOL it turns out to be untrue! The second and, in the context of hardware design, the more important reason is that it is desirable to be able to rerun the proof of a design after a small modification to ensure that no unexpected dependencies have introduced errors. This task needs to be as automatic as the final design-rule checking phase in current CAD design systems.

On the hardware development side the intention is to provide tools that can work alongside existing CAD tools. This means providing interfaces to existing CAD standards such as VHDL and design databases. Verification needs to be introduced as extra tools for the designer rather than a new encapsulated system.

The fact that the project has looked at all levels of a system design has produced some interesting ideas. For example, the issue of modelling real time behaviour of a program has some architectural implications on the processor design. If instruction execution times do not depend on instruction operands, it is possible to simplify the verification of real-time programs by separating the timing verification (in temporal logic) from the functional verification (in Hoare logic).

The results of a project like SAFEMOS may not be particularly impressive in themselves (for example a small subset of a six-year-old processor like the T414), but the importance is in building up the basic techniques, theories and tools which will form a basis for more realistic design verification later. Of particular importance are techniques which allow verification of hardware and software in combination, as this capability is needed not only for microcode and embedded control programs, but also for the construction of verified compilers and hardware synthesizers.

4. A new design: the T9000

In 1989, Inmos embarked on the design of a new transputer, now launched as the T9000. The T9000 was to be a completely new design, retaining instruction-set compatibility with the T800, but with a complete re-implementation of the processor, and with a new virtual channel processor. The virtual channel processor would allow any number of logical channels to be multiplexed onto the four physical communication links. The T9000 would be implemented on a new manufacturing process, and the combined result of extra interconnect layers and finer geometries was to allow about five times as many transistors as the T800.

Although we had already successfully used formal design methods in the T800 work, it was not easy to introduce them into the T9000 design. Many engineers are accustomed to verifying designs by simulation, and have little experience in the use of formal methods. In addition, many of the notations and tools produced by mathematicians for formal design work are unfamiliar to engineers, with the result that engineers will avoid using them until the complexity of the design makes it essential. Bringing together engineers and mathematicians was a continuing challenge throughout the T9000 work.

The first problem to emerge was that of establishing that the heavily pipelined processor of the T9000 actually implemented the same operations as the unpipelined T800. Variations would obviously run the risk of making existing software and compilers useless, eliminating the claimed advantage of 'compatibility with the T800'. Equally important was the problem of ensuring that the pipeline did not

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Downloaded from rsta.royalsocietypublishing.org

deadlock or livelock. The task of verifying the correct operation of the pipeline has been performed at Oxford, and is described in A. W. Roscoe's paper. One problem in the pipeline verification has been that a substantial amount of design had taken place before the need for formal design and verification techniques became clear. This has had the well-known effect of complicating the verification process.

The second major problem to arise was that of designing and verifying the virtual channel processor. The design of the relatively simple communications hardware of the T800 had given rise to several problems, including a design fault which took over a month to locate. This is not surprising, as the communications hardware of the T800 has complex interactions with the processor and with the environment. It consists of a collection of concurrent state machines, and is not amenable to the methods used for the verification of the floating point unit.

5. Designing the T9000 channel processor

The T9000 virtual channel processor (VCP) is a device which allows any number of logical connections between two processors to be implemented by a single physical connection. Whereas the specification of a single logical connection is straightforward, the protocols which are needed in the hardware implementation are sufficiently complicated that some degree of automatic verification is essential.

The theory and methods which are involved in the VCP verification are derived from research which has been carried on over the past 10 years. The emergence of a theory which is suitable for the automatic verification of the VCP implementation has been made possible by the conjunction of two lines of research. The first of these results from Hoare's work on communicating sequential processes (CSP). This culminated in the mathematical model of CSP reported in Brookes & Roscoe (1985). The mathematical model for CSP captures a theory of *correctness* in a precise and formal way. The theory of correctness formalizes a set of criteria by which to judge when an implementation meets the conditions of its specification. This fundamental basis for automatic verification has proved to be a controversial subject. If the correctness criteria are too strict, then some implementations which intuitively feel correct will be ruled out. If the correctness criteria are too lenient, then implementations which are, intuitively, incorrect will be allowed. The model for CSP captures a set of criteria which is most appropriate for our purposes.

The second line of research which has contributed to the verification is that which grew from Milner's calculus of communicating systems (CCS). The model for this theory involves the formalization of state machines. A state machine can be in one of several different states. In each state, the machine has a number of transitions available. Associated with each transition is the name of event (an interaction with its environment or an internal action). Having made a transition, the machine may find itself in a new state. The theory of CCS is explained in Milner (1989).

An early attempt to produce a method for judging correctness of state machines can be found in Barrett (1988). Barrett's main object is to find a method by which to prove the correctness of the transputer implementation of Occam. To do this, both the semantics of Occam and the transputer implementation are expressed in terms of state machines. The correctness of the implementation is shown using an *ad hoc* adaptation of a technique which is used to prove the equivalence of state machines. This technique turns out to be *incomplete*. In other words, there are some correct implementations which cannot be proved using this technique. However, following

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES
PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Downloaded from rsta.royalsocietypublishing.org

the work of Gardiner & Morgan (1992), we have developed a complete method of proof for the correctness of state machines.

Although the complete theory of correctness is accessible to and can be applied by an engineer with a good understanding of mathematics, this process is time consuming and does not, in general, exercise the skills which make a good engineer. In the course of the verification of the VCP, it became evident that it would be useful to ignore the full power of the correctness theory and, instead, to produce a fully automatic tool which can be used to decide the correctness of finite state machines. This subset of the theory identifies a subset of state machines (those which can assume a finite number of states) for which there is a *decidable*, complete theory. A decidable theory is one which may be fully automated. This means that it is possible to implement a computer program which can ascertain the correctness of an implementation. The advantage of this over the mathematical theory is that the engineer is free to use her intuition in the design of correct implementations but may then submit her implementation to formal verification.

The correctness-checking tool has been successfully applied to the design of the T9000 VCP and is now being applied to the design of the control link (the mechanism which regains control of a processor which is running a bad program). As yet, however, the tool is still in a prototype phase. Work is in progress to improve the user interface, particularly in the area of diagnostics for incorrect implementations. The problem here is to provide some useful clues as to what has gone wrong when an implementation is not correct; a 'yes' or 'no' answer is often not enough. There are great opportunities here to be able to pinpoint very precisely the source of an error, thereby decreasing the amount of time spent debugging an implementation.

(*a*) *The VCP specification*

The T9000 VCP can be specified as the interleaving of a number of channels:

$$VCP = |||_{i=0}^{2^{17}-1} i:CHAN.$$

This expression specifies that each of the channels should behave independently. We will not devote much attention to the specification of the VCP as a whole but rather, we will study the behaviour and implementation of individual channels.

A channel provides a point-to-point link between two processes. A single processor may execute any number of processes, but at most one may use any particular channel for input or output. When a process ouputs a value to a channel, it is descheduled until a corresponding input process has received the value. Conversely, when a process inputs a value from a channel, it is descheduled until a corresponding output process has sent a value along the channel. While a process is descheduled, it may not engage in any actions. In particular, a descheduled process may not input or output from a channel.

In the formal specification of the channel, we consider a number of variables which describe, for instance, the number of inputs which have been started but not yet completed. The inequalities which govern the channel behaviour are given below. The expressions $\#input$ and $\#output$ stand for the number of inputs or outputs which have begun and $\#run(input)$ and $\#run(output)$ stand for the number of inputs or outputs which have terminated:

(1) no input terminates before it has begun:

$$0 \leqslant INRI = \#input - \#run(input)$$

Downloaded from rsta.royalsocietypublishing.org

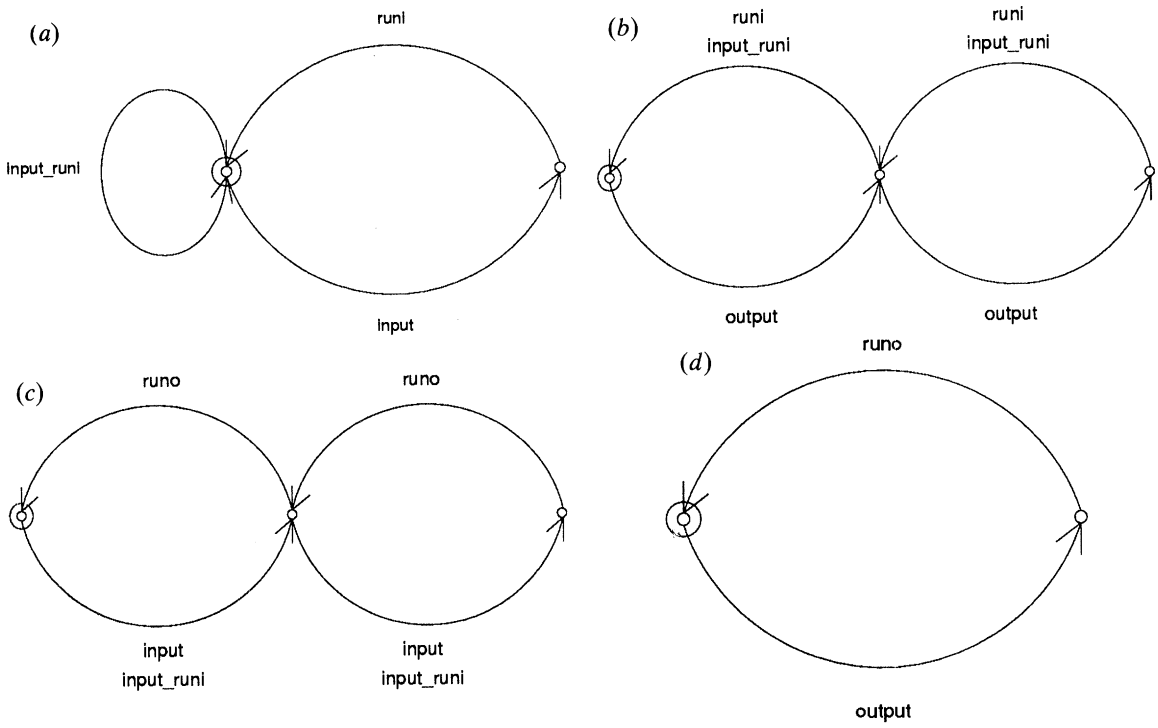

Figure 3. State machines corresponding to the channel specification. (*a*) $0 \leqslant INRI = \#input - \#run(input) \leqslant 1$. (*b*) $0 \leqslant OUTRI = \#output - \#run(input) \leqslant 2$. (*c*) $0 \leqslant INRO = \#input - \#run(output) \leqslant 2$. (*d*) $0 \leqslant OUTRO = \#output - \#run(output) \leqslant 1$.

(2) an input cannot terminate before it has been matched by an output:

$$0 \leqslant OUTRI = \#output - \#run(input)$$

(3) an output cannot terminate before it has been matched by an input:

$$0 \leqslant INRO = \#input - \#run(output)$$

(4) no output terminates before it has begun:

$$0 \leqslant OUTRO = \#output - \#run(output).$$

Part of the specification formalizes the assumptions which are made about the way processes behave and how channels are joined up:

(1) there is never more than one active input:

$$INRI \leqslant 1$$

(2) there is never more than one active output:

$$OUTRO \leqslant 1.$$

To be able to define a finite state machine using these variables, we must be able to prove that each variable may only take on a finite range of values. This is clearly true of the variables *INRI* and *OUTRO*, but we must prove that the other variables are also bounded. The proof proceeds as follows:

(1) the combined number of active inputs and outputs is given by both $OUTRO + INRO$ and $INRI + OUTRO$. Therefore:

$$OUTRI + INRO = INRI + OUTRO.$$

Downloaded from rsta.royalsocietypublishing.org

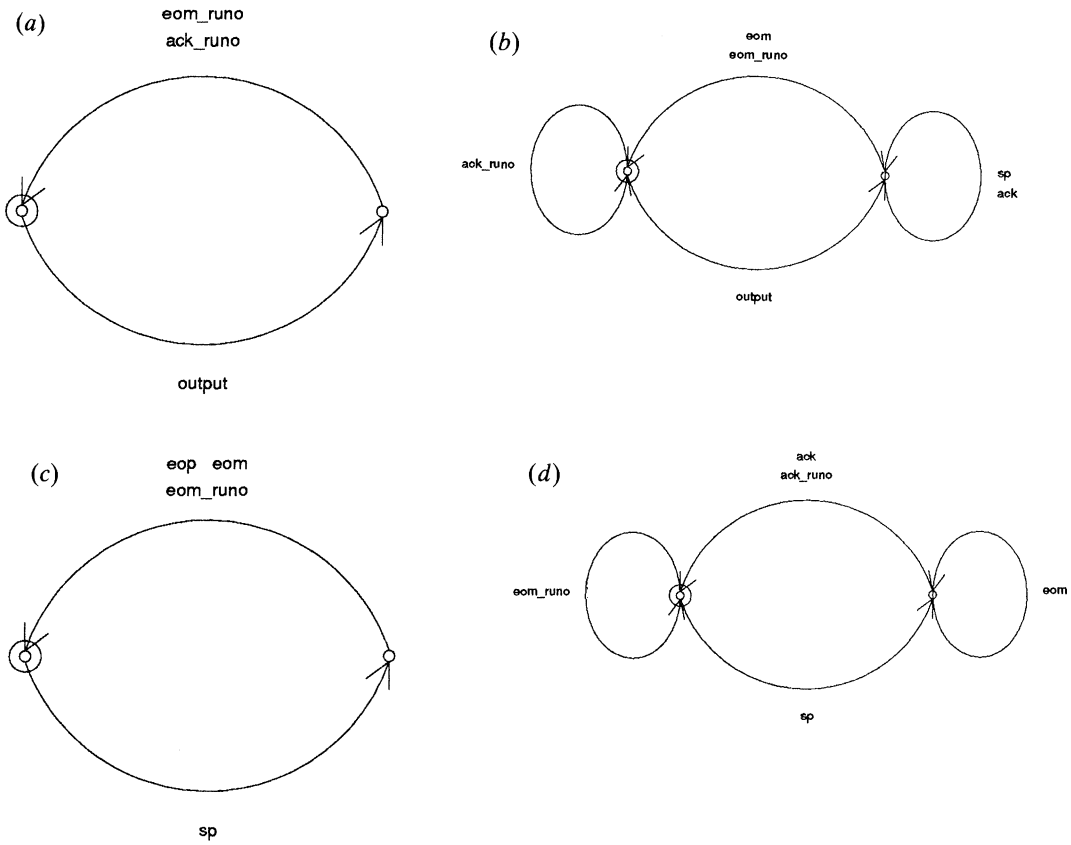

Figure 4. Specification of the output machine. (*a*) No output terminates before it has begun; (*b*) messages begin with output and end with an end of message token; packets may only be sent when there is an unterminated message; acknowledges cause a reschedule if and only if there are no unterminated messages; (*c*) packets are terminated with an end of packet or end of message token; (*d*) start of packet alternates with acknowledgements; an output is rescheduled when sending end of message if and only if all packets have been acknowledged.

(2) the following inequalities can be summed to give a bound of 2 on the combined number of active outputs and inputs:

$$0 \leqslant INRI \leqslant 1 \quad 0 \leqslant OUTRO \leqslant 1$$
$$0 \leqslant INRI + OUTRO \leqslant 2.$$

(3) using the equality and the inequality from above gives bounds on the number of inputs and outputs which have not caused a scheduling action:

$$0 \leqslant OUTRI \leqslant 2$$
$$0 \leqslant INRO \leqslant 2.$$

The fact that there may be two output commands before an input is scheduled arises from the sequence of events: *input*, *output*, *run*(*output*), *output*. At this stage, the only action which is possible is to reschedule the input.

The state transition diagrams which correspond to each of the variables are shown in figure 3. Each of the diagrams corresponds to one of the specification variables. The commands and responses of the VCP have the effect of changing the values of the

Downloaded from rsta.royalsocietypublishing.org

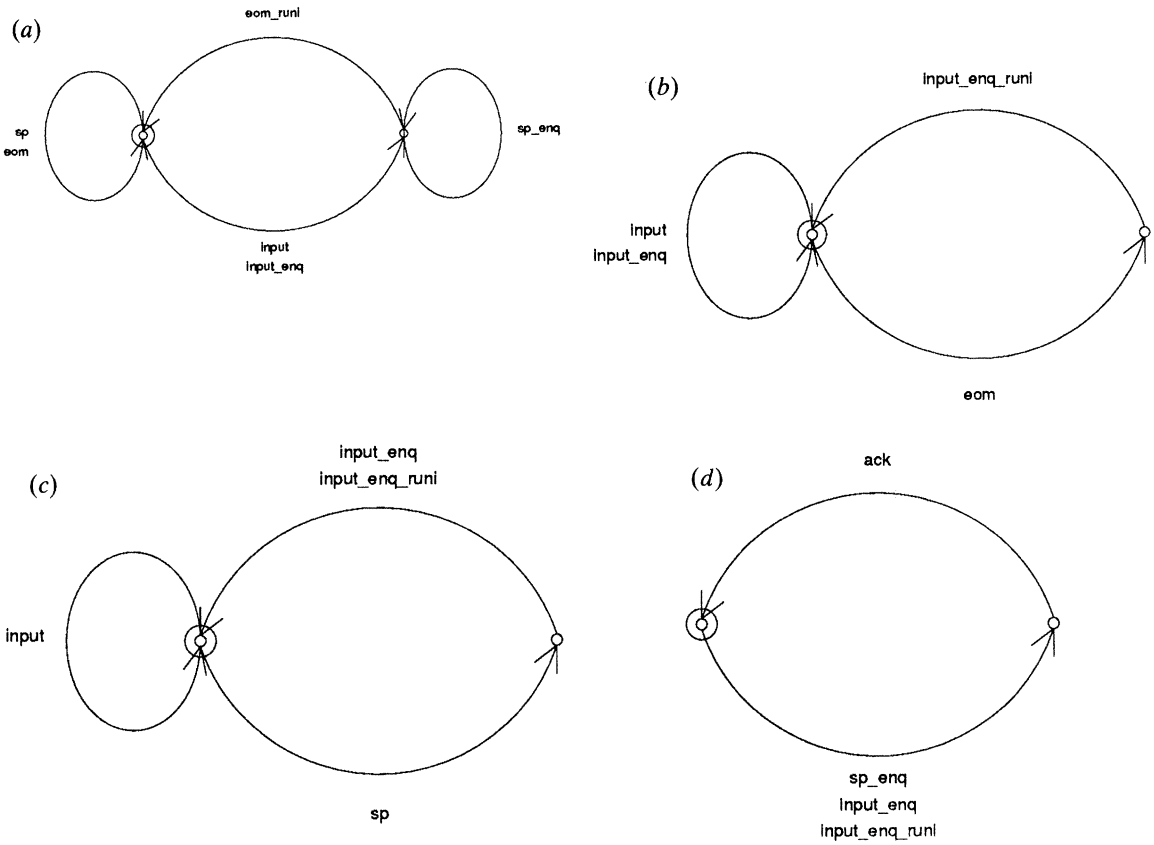

Figure 5. Specification of the input machine. (*a*) No input terminates before it has begun; receiving a start of packet causes an acknowledge to be queued if and only if there is an active input; receiving an end of message token causes an input to be rescheduled if and only if there is an active input; (*b*) an input is rescheduled if and only if it has been matched by a complete message; (*c*) input causes an acknowledge to be queued if and only if there is a packet with no queued acknowledge; (*d*) acknowledges are only sent if and only if they have been queued.

variables (for instance, *input* increases the variable *INRO*) and this is noted on the diagrams by an arrow labelled with the action. Each of the diagrams may be interpreted as a CSP process which restricts the overall behaviour of the channel. All of the restrictions are combined in parallel to give the specification of a channel:

$$CHAN = INRI \parallel OUTRI \parallel INRO \parallel OUTRO.$$

(*b*) *The transputer link protocol verification*

A channel must be implemented in a distributed way between two processors. One processor *IN* is at the input end of the channel and has responsibility for rescheduling input processes. The other processor *OUT* is at the output end and has responsibility for rescheduling output processes. The processes are connected by a link, *LINK*. Each message along the channel is broken up and transmitted through the link as a sequence of packets.

The output side processor transmits a packet which is terminated with a symbol which indicates whether more packets are to come. The input side acknowledges each packet.

Downloaded from rsta.royalsocietypublishing.org

Table 1. *Events which the machine records*

event	meaning
runi	reschedule input
runo	reschedule output
input	receive input command
input_enq	receive input command and queue a packet acknowledgement
input_enq_runi	receive input command, queue a packet acknowledgement and reschedule input
output	receive output command
sp	send/receive start of packet
sp_enq	receive start of packet and queue a packet acknowledgement
eop	send/receive end of packet token
eom	send/receive end of message token
eom_runi	receive end of message token and reschedule input
eom_runo	send end of message token and reschedule output
ack	send/receive packet acknowledgement
ack_runo	receive acknowledgement and reschedule output

The diagrams which correspond to the input and output machines are shown in figures 4 and 5. The events which the machines recognize are given in table 1.

The final element in the implementation is the link which joins the input machine to the output machine. The function of the link is to pass the packets from the output side to the input side and to pass acknowledgements in the other direction. The specification of the link is not formalized here, we will only refer to the formalization as *LINK*.

The inner part of the VCP implementation can now be described by the parallel composition of the three parts:

$$ILO = IN \parallel LINK \parallel OUT.$$

Some of the link transactions have external side-effects which cause processes to be rescheduled. These actions are renamed so that they reflect only the external side-effect:

$$EXT = ILO[\text{runi/eom\_runi, runo/eom\_runo, runo/ack\_runo}].$$

The packet actions which do not have any external side effects are hidden:

$$IMP = EXT \backslash \{\text{sp, eop, eom, ack}\}.$$

The correctness of the implementation is established automatically according to the correctness theory. Formally, this states that the traces of the specification and the implementation are identical:

$$traces(CHAN) = traces(IMP).$$

6. The design process

Any component as complex as the T9000 contains a number of different subcomponents such as the processor, floating point unit and communications processor. Each of these has an associated set of design techniques and representations. There are many different representations in common use and these operate at varying levels of abstraction. Examples are an English specification, a

Downloaded from rsta.royalsocietypublishing.org

timing diagram, a Z specification, an Occam functional simulation, a VHDL description, the silicon layout, a circuit diagram. It is inevitable that a design will be expressed in several of these notations.

In our experience, it is *essential* that the descriptions of a design in different notations can be checked against each other. Many of the descriptions are large, so that it is desirable to use notations suitable for computer-assisted manipulation to simplify the construction of checkers and simulators; in this context pictures and English are not suitable. The available techniques for checking the equivalence of two descriptions are the following:

1. Visual inspection. This widely used technique is inconsistent with the scale of current designs. It is extremely error prone and time consuming, and is not consistent with the need for fast design iteration.

2. Simulation of the design in two notations and comparison of the results (for example to compare an Occam functional description with an HDL implementation). This technique has its uses, but requires the construction of a test-suite of data to be run on the two simulations. It is easy to forget that the construction of the test-suite can easily take more time than the design itself, and also that the speed of simulation can be a major problem.

3. Automatic compilation (synthesis) of one description from another. This is likely to be reliable and fast, but depends on the correctness of the compiler.

4. Automatic comparison of the two descriptions. This is likely to be reliable and fast, but again depends on the correctness of the comparison tool.

5. Proof by hand that one description (the implementation) satisfies another (the specification). This technique is only possible where current theory is well enough developed. Also, it is not consistent with the need for fast design iteration (but there are cases where re-working a proof by hand can be much faster than simulation).

6. Computer-assisted or computer-checked proof. This technique is only applicable where current theory and tools are adequate.

The scale of current designs makes it important to design the design process itself. In particular, it is important to plan the notations to be used to represent each part of the design at each of the various levels of abstraction, and to identify the techniques to be used to establish equivalence between the representations. This makes it possible to expose weaknesses in the design process, especially those which are likely to cause delays as a result of time-consuming (and unanticipated) verification processes. At an early stage, it is often possible to develop new tools to assist in the process.

New capabilities are needed in CAD systems to support the use of verification tools. Any design will involve a number of standard tools, together with the need for specialized tools to be constructed as the design progresses. There is therefore a need for design databases in which the method for comparing two representations of a design is held as a program which takes as its parameters the two representations together with other information specific to the comparison being made.

7. Future directions

In the past 10 years, VLSI designs have moved from 10^5 to 10^6 transistors. This has been achieved not only by advances in manufacturing, but also by advances in design techniques. Automated checking of silicon design rules and circuit connectivity have emerged and become a standard part of commercial CAD systems.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Downloaded from rsta.royalsocietypublishing.org

In the next 10 years, designs will move to over 10^7 transistors, and this will require a further advance in design techniques. Designs of this size must be constructed from independent modules, and the behaviour of these modules and their interfaces must be specified. Automated checking that the composition of such modules corresponds to an overall specification will become an essential part of future CAD systems.

An important development is the increasing use of synthesis tools. Much of the control logic in the T9000 is produced by synthesis from VHDL descriptions. Synthesizers will, in future, also support the construction of structures such as microcoded datapaths. The result will be that the role of the engineer in VLSI design will change. The design of the basic cells used by the synthesizer is clearly an electronic design task, with the cells being verified by simulation. The design of the input to the synthesizer will not require skills in electronic design, but will require extensive use of behavioral descriptions and programs, with associated specification, transformation and proof tools.

References

Barrett, G. 1988 The semantics and implementation of OCCAM. D.Phil. thesis, Oxford University.

Barrett, G. 1989 Formal methods applied to a floating point number system. *IEEETSE* **15**, 611–621.

Brookes, S. D. & Roscoe, A. W. 1985 An improved failures model for communicating processes. In *Seminar on concurrency* (ed. S. D. Brookes, A. W. Roscoe & G. Winskel), LNCS 197, pp. 281–305. Berlin: Springer-Verlag.

Gardiner, P. & Morgan, C. 1992 A single complete rule for data refinement.

Goldsmith, M. H. 1988 The Oxford OCCAM transformation system (version 0.1). Draft user documentation, Oxford University Programming Research Group.

Gordon, M. J. C. 1986 The HOL system description. Cambridge Research Centre, SRI International, Cambridge.

Harper, R. M., MacQueen, D. B. & Milner, A. J. R. G. 1986 Standard ML. Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Edinburgh University.

Hoare, C. A. R. & Shepherdson, J. C. 1985 Mathematical logic and programming languages. Prentice Hall.

Milner, A. J. R. G. 1989 Operational and algebraic semantics of concurrent processes. In *Handbook of theoretical computer science* (ed. J. van Leeuwen). Amsterdam: North-Holland.

Roscoe, A. W. 1985 A denotational semantics for OCCAM. In *Seminar on concurrency* (ed. S. D. Brookes, A. W. Roscoe & G. Winskel), LNCS 197, pp. 306–329. Berlin: Springer-Verlag.

Roscoe, A. W. & Hoare, C. A. R. 1988 *The laws of OCCAM programming*, TCS 60, pp. 177–229. North Holland.

Spivey, J. M. 1989 *The Z notation, a reference manual.* Prentice Hall.

Discussion

W. A. HUNT (*Computational Logic, Inc., U.S.A.*). How is VHDL formally related to silicon and Occam?

M. D. MAY. The compilation from VHDL into silicon is performed by commercial silicon compilation programs (such as SYNOPSYS) and we rely on their correctness. We can provide some confidence in the correctness of the results of these programs by simulation.

When considering the relation between Occam and VHDL it must be remembered that we are using Occam programs written in a special style as source and we produce a subset of VHDL. In this context it is possible to reason about correctness in a 'semi-

Downloaded from rsta.royalsocietypublishing.org

formal' way; the correctness statement is along the lines of: if you hold the input lines to the circuit at the values of the input variables of the procedure then at some time the output wires will stabilize and will hold the same values as the output variables of the procedure.

An important factor in the correctness is that the translation process consists of a series of small 'correct' transformations on the Occam code followed by a simple translation of a very small subset of Occam into VHDL which amounts to little more than pretty-printing. In this way translation correctness is isolated from the issues of program structure transformation (e.g. changing multidimensional arrays to one dimension).

W. A. HUNT. Why didn't Dr May try to 'cross-fertilize' the engineers and the formal methods people?

M. D. MAY. The abilities required for the electronic design of state-of-the-art VLSI products are quite different from those required for the formal verification of the designs. There is no quick way to train an electronic engineer in the use of formal methods, nor to train a formal methods expert in electronic engineering. We have therefore created design teams which include both areas of expertise. In time, it is inevitable that some cross-fertilization will occur. This has happened at Inmos, where such 'experiments' are encouraged!

E. M. CLARKE, JR (*Carnegie Mellon University, U.S.A.*). I am interested in the technique that is used for verifying finite-state hardware controllers. I realize that a 'state explosion' may occur when such controllers execute in parallel, and that it may be difficult to compute the exact number of reachable states in the parallel composition. However, is Dr May able to give an *upper bound* on the number of states of a parallel composition, perhaps by computing the product of the sizes of the component controllers? New model-checking algorithms (based on the use of binary decision diagrams) can handle systems with enormous numbers of states.

M. D. MAY. The moderate number of small state machines which are composed in parallel to form the specification of a virtual channel does lead to a potentially large number of states. The full specification contains eight state machines with two or three states each, giving a potential number of $2^6 \times 3^2$ states. This is approaching the performance limit of the most naive algorithms. We are currently working on incorporating more sophisticated model-checking algorithms into our tool. However, the problem which we are trying to solve is known to be NP-complete and we believe that we will always be able to achieve a problem size which is beyond the abilities of sophisticated algorithms and powerful supercomputers. Instead, we intend to use the tool to check the correctness of a sequence of smaller refinements.

There are three benefits to our approach: it scales to any size of problem; the refinement steps produce useful documentation about the structure of the design which aids maintenance and comprehensibility; the tools which will be necessary do not require powerful supercomputers and will be widely available on standard workstations.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES