Towards Verified Systems: The SAFEMOS Project

Jonathan P. Bowen and He Jifeng

Oxford University Computing Laboratory, Programming Research Group
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
Email: {Jonathan.Bowen,Jifeng.He}@comlab.ox.ac.uk

Roger W. S. Hale and John M. J. Herbert

SRI International Cambridge Research Centre
Suite 23, Millers Yard, Mill Lane, Cambridge CB2 1RQ, UK
Email: {rwsh,jmjh}@sri.cam.com

Abstract

The collaborative safemos project has investigated the formal development of embedded systems from specification through to a real-time programming language, compilation to object code and the formal design (and even automatic compilation) of a hardware machine to execute that code. The project has used Occam and the Transputer as an inspiration for its investigations, with real-time extensions where required. HOL has been used for mechanical verification where appropriate. A close liaison with the related collaborative European ESPRIT **ProCoS** project has been maintained to ensure that research on both projects is coordinated. This paper gives an overview of the work of the project with particular regard to the mathematical techniques used for the specification and verification process.

1 Background

In the last decade, the use of software in safety-critical systems has increased by around two orders of magnitude. However current widely used development techniques are still sadly lacking in their ability to avoid the occurrence of errors in such systems. Formal mathematically based methods provide one means to avoid the introduction of errors into systems at the design stage by increasing the preciseness of descriptions earlier on in the process and allowing the possibility of proven transformations and relationships between the specification and implementation of a system.

There is currently great interest in both academic and industrial circles in the issues involved in the use of formal methods but the techniques still need further investigation and promulgation to make their widespread use

a reality [12, 13]. A number of safety-related standards, are currently being introduced and the recommendations in these could well have a significant impact on the use of formal methods in the development of safety-critical systems. Many standards in this area are now mentioning formal methods as one option to improve the correctness of safety-related software [5]. A few, such as the UK MoD 00-55 draft standard [41], even mandate the use of formal methods although there is still much debate and industrial resistance.

This provides the setting into which the research work briefly presented in this paper is intended to fit in years to come. Here we concentrate on an overview of mathematical techniques used by the safemos project intended to aid formal development of software and hardware for embedded high integrity systems.

2 Project Overview

The collaborative UK IED (Information Engineering Directorate) safemos project (1989–1993) has investigated techniques to aid the formal verification of mixed hardware/software systems. Aspects of system specification and verification from an abstract formal description of the system down to the underlying hardware have been addressed, with particular regard to real-time issues.

The project was influenced and inspired by the simplicity of the Occam programming language [36], with its well established formal underpinning of the process algebra CSP (Communicating Sequential Processes) [31] and a wealth of algebraic laws [47] to aid formal transformation of programs. In addition, the Transputer microprocessor developed by Inmos [37] provides a platform for the implementation of Occam programs.

The HOL (Higher Order Logic) [20, 18, 52] theorem proving system was used to perform machine-checked proofs. The HOL system provides an LCF-style theorem proving environment [21, 45] and supports a version of classical higher-order logic based on Church's formulation of simple type theory [2, 16]. HOL includes ML as a metalanguage: the ML language was originally developed as part of LCF, but is now an independent programming language in its own right [40]. It is an eager-evaluation functional language with a polymorphic type discipline.

Some use of the formal notation Z has been made on the project. Z is based on Zermelo-Fraenkel set theory and first order predicate calculus [50], with the addition of the schema 'box' notation to aid the structuring of the large amount of detailed mathematics that is necessary to specify systems of a realistic size. An advantage of a formal notation like Z is that it has been accepted for international standardisation by ISO which is likely to lead to wider industrial acceptance [14]. Industrial practitioners are often reluctant to use raw mathematics and standardisation is a major

contributor to aiding its infiltration in certain sectors such as those involved with safety-critical systems [12].

The work described here has been undertaken by the following industrial and academic partners in the UK:

- Inmos Limited, Bristol;
- SRI International Cambridge Computer Science Research Centre;
- Oxford University Computing Laboratory, Programming Research Group;
- University of Cambridge, Computer Laboratory.

The project has aimed to address some of the problems facing the designers and users of microprocessors and micro-controllers that are arising as the complexity and power of these devices increase. Microprocessors are being used to perform increasingly complex tasks; as a result the ability to ensure correct design by traditional design techniques, centered around experimental testing, is becoming problematic.

The use of formal design methods seems to offer a way out of this situation by providing a design methodology which prevents the introduction of errors into designs through the rigorous use of proof techniques to validate designs against specifications. The results of the safemos project aim to demonstrate the feasibility of these methods for real-time control systems. In the future, such real-time controllers will increasingly consist of processors running embedded programs along with specialised interface hardware. Thus the project has addressed the problem of verifying both hardware and software.

2.1 Goals

The original three goals of the project, started in 1989, were [48]:

1. to demonstrate that it is feasible and commercially advantageous to verify systems containing both hardware and software by machine checked formal proof;
2. to develop the methodology and tools needed for performing such verifications and for estimating their costs;
3. to gain improved scientific understanding of the practical use of existing formal methods and tools, including HOL [20], Z [51] and CSP [31].

These goals have been addressed by designing an Occam-like real-time language, a program verifier for that language, a verifiable Transputer-like processor design and a verified translator to compile the real-time language into the processor instruction set. A simple demonstrator example has been undertaken show how a program can be verified to meet a specification.

A major research challenge was the co-ordination of the diverse formal proofs needed for diverse system verification within a coherent and uni-

form framework. The aim of the safemos project has thus been to develop formal methods for reasoning about Occam-like programs running on a Transputer-like processor, in the CSP tradition. While perfect compatibility between different components is very hard to achieve on a collaborative project at geographically separate sites, particularly one undertaking research, the early selection of an existing, and real, language and processor, together with the use of a single mechanical verification support tool, has prevented any site from becoming too devolved in their work. The following sections outline some of the most important work areas undertaken on the project.

Other related work is being carried out elsewhere, most notably, on the European collaborative ESPRIT Basic Research **ProCoS** project [8, 9], and at CLInc, a company in the US largely dedicated to the development and use of the Boyer-Moore theorem prover [17, 42]. Contact has been maintained with both these efforts; in particular, the **ProCoS** project is also studying Occam-like languages and Transputer-like machines, so work on both projects is directed towards a common goal. However on the safemos project there is a greater concentration on mechanically assisted proofs centred around a single tool, namely HOL, and a more balanced study of hardware as well as software verification issues.

2.2 The SAFEMOS Tower

In designing a system, a number of different levels of abstraction must normally be considered. For example, a designer must transform a (possibly non-executable) *specification* into an *implementation* in the form of an executable program; a compiler must automatically convert a high-level program into low-level object code; and the underlying hardware must correctly execute that code using simple logic gates and latches (for example). The transformations should ideally be error-free and techniques to avoid the introduction of errors during this process are obviously desirable. The use of formal methods is one such technique since they reduce the ambiguity in the process, help to increase human understanding and allow formal proofs, reasoning and even calculation to be undertaken. Mechanical support for the process reduces the chance of human error which is highly likely in all but the simplest endeavours.

The safemos project has considered such transformations and mechanisation at a number of different levels:

- System specification: State-transition Assertions (STA), outlined in section 3, and Timed Transition Systems [28], described in [23].
- Compilation using an Occam-like sequential programming language and Transputer-like processor: Interval semantics based on Interval Temporal Logic (ITL) [22, 24, 43].
- Microprocessor design: Incremental framework using Higher-Order

Logic (HOL) [20].

- Hardware compilation: A novel refinement algebra approach via a normal form as outlined in section 9, allowing the possibility of hardware/software co-design.

In an ideal world, different levels of abstraction would interface together to form a seamless 'tower' of design descriptions. Each level moves towards the final implementation by being assigned a semantics that is equivalent or 'better' than the previous level w.r.t. some refinement ordering. (Here, 'better' means being more deterministic and/or terminating more often.) This ideal has yet to be fully achieved in practice although the work of the related **ProCoS** project continues to strive for this goal.

A uniform proof environment, namely HOL (Higher-Order Logic), has been used for the mechanisation of proofs on the project, and the adoption of a common tool helps in the linking of different levels of abstraction. This has been the experience of the CLInc effort [17], as previous mentioned, who use the Boyer-Moore theorem prover for proofs of a number of related levels in both software and hardware.

The rest of the paper provides details of specific aspects of the work of the project.

3 State Transition Assertions

A graphical state-transition approach to specifying hard real-time reactive systems has been developed [19]. This is refined to a formal notation based on sentences called *State Transition Assertions* (STAs). These have a set-theoretic semantics that can be used to justify various laws, which combine aspects of *Interval Temporal Logic* (ITL) [22, 24, 43] and and Hoare Logic [30]. The semantics of programs can also be represented by sets of STAs, and then verification is performed by using laws to combine the STAs from the program to obtain the specification.

3.1 Hard real-time

Hard real-time systems are required to meet explicit timing constraints, such as responding to an input within 100 milliseconds of a change. The temporal requirements are an essential part of the required behaviour, not just a desirable property as in the case of *soft real-time* systems. The work aims to combine hardware and software verification techniques to produce a formal method for hard real-time programming. In this method, programs are refined from specifications consisting of a kind of state transition diagram. These diagrams have a precise semantics and the verification that programs implement them is by machine checked formal proof.

A key element of the method is that the program semantics used for verification is determined by what happens when the compiled program runs on the processor being used. This is achieved by defining the semantics

via the compiler and processor specification.

3.2 Method

This work provides a possible foundation of a formal method for developing hard real-time programs. In summary, the method is as follows:

1. Write the specification using annotated state-transition diagrams and interpret them as sets of state transition assertions (STAs).
2. Develop a program by identifying nodes in the diagram with sets of processor states.
3. Verify the program by showing that its transitions (which are mechanically derived from the compiler and processor specification) entail the required transitions using laws for combining STAs.

Specifications are formalised as predicates on machines. A typical specification involves a number of transitions between wait states. These can be represented using a *state transition assertion* (STA), that combines aspects of the 'leads-to' and 'until' operators of temporal logic. The general form of an STA is:

$$\mathcal{M} \models A \xrightarrow[P]{Q} B$$

where:

- $\mathcal{M} : \mathit{inputs} \to \mathit{state} \to \mathit{state}$ is a machine;
- $A : \mathit{state} \to \mathit{bool}$ is called the *state precondition*;
- $B : \mathit{state} \to \mathit{bool}$ is called the *state postcondition*;
- $P : \mathrm{seq}\ \mathit{inputs} \to \mathit{bool}$ is called the *input precondition*;
- $Q : \mathrm{seq}\ \mathit{state} \to \mathit{bool}$ is called the *output postcondition*.

(*bool* represents Boolean values.)

The intuition behind state transition assertions is as follows: if $\mathcal{M}$ is in a state satisfying A and a sequence of inputs arrives that satisfies P, then a state satisfying B will be reached and the sequence of intermediate states will satisfy Q.

A simple example of a multiplier program has been used to demonstrate the technique, mechanically checked using HOL, and is presented in [7].

4 Real-time Programming Language

The safemos programming language, SAFE, is a real-time sequential imperative language with input and output constructs and with deadline constraints. The syntax of SAFE processes is given by the following BNF, where

p is a process, e an expression, b a Boolean, x a variable and t a time.

$$p ::= \texttt{SKIP} \mid \texttt{STOP} \mid x :=_t e \mid \texttt{READ}_t\ x1\ x2 \mid \texttt{WRITE}_t\ x\ e \mid p1 ; p2 \mid \texttt{IF}_{t1,t2}\ b\ p1\ p2 \mid \texttt{WHILE}_{t1,t2}\ b\ p \mid \texttt{LOCAL}\ x\ p$$

This syntax has been formally represented in HOL as a recursive type definition.

The process `SKIP` does nothing and terminates immediately. `STOP` holds the execution forever. The assignment, $x :=_t e$, assigns the expression e to the variable x, and takes at most t time units to complete. Input and output are also time-bounded. The input process, $\texttt{READ}_t\ x1\ x2$, assigns the current input on port $x1$ to variable $x2$, and the output process, $\texttt{WRITE}_t\ x\ e$, outputs the expression e on port x. Input and output in SAFE are memory-mapped; that is, the input and output constructs are simply assignments to I/O registers and do not perform synchronisation. Synchronisation may be achieved by means of protocols implemented in SAFE [15].

Sequential composition, $p1 ; p2$, does not introduce any time overhead. For the conditional, $\texttt{IF}_{t1,t2}\ b\ p1\ p2$, no more than $t1$ time units may be used in testing the Boolean expression b, and no more than $t2$ units following the termination of either $p1$ or $p2$. The loop, $\texttt{WHILE}_{t1,t2}\ b\ p$, is a conventional while loop in which at most $t1$ time units may be used for testing the condition b, and at most $t2$ units for reinitiating the loop. Finally, the construct $\texttt{LOCAL}\ x\ p$ makes the variable x local to process p. A very similar real-time language has also been developed on the **ProCoS** project [25].

We assume here that time constraints are measured in machine cycles but, in general, any appropriate time units could be used – even real time, given a suitable mapping between states and real times.[1] Whether or not a particular time constraint can be achieved in reality obviously depends on the complexity of the expression and on the underlying architecture. The constraints are therefore checked by the compiler; values that are too small are rejected.

5 Interval Semantics

We have defined the formal semantics of SAFE in HOL. It is a model-based semantics based on ITL, which captures timing properties in a natural way and also permits a uniform treatment of programming and assembly language constructs. This facilitates compiler verification.

Each SAFE process is identified with a predicate on intervals, as described below. An interval is a non-empty sequence of states and may be either finite or infinite; its length represents the number of time steps between start and finish.

In the HOL definitions below σ stands for a (finite or infinite) interval;

[1] We assume that a program necessarily executes in a sequence of discrete steps.

$\texttt{len}\ \sigma$ for its length; $\texttt{init}\ \sigma$ and $\texttt{last}\ \sigma$ for the initial and final states of σ; and $\texttt{pfx}\ i\ \sigma$ and $\texttt{sfx}\ i\ \sigma$ for the ith prefix and ith suffix subinterval, respectively. Let us also write $\mathcal{M}^{\mathrm{p}}\, p$ for the meaning of a process p, and $\mathcal{M}^{\mathrm{e}}\, e$ and $\mathcal{M}^{\mathrm{b}}\, b$ to denote the meanings of expression e and Boolean b, respectively.

The process `SKIP` is identified with the predicate `Skip`, which is true on any zero-length interval.

$$\mathcal{M}^{\mathrm{p}}\,\texttt{SKIP}\,\sigma \stackrel{def}{=} \texttt{Skip}\,\sigma$$
where
$$\texttt{Skip}\,\sigma \stackrel{def}{=} (\texttt{len}\,(\sigma) = 0)$$

The process `STOP` never terminates and keeps the internal state, i.e. the set `V` of program variables and output ports, stable (constant).

$$\mathcal{M}^{\mathrm{p}}\,\texttt{STOP}\,\sigma \stackrel{def}{=} \texttt{len}\,(\sigma) = \infty \wedge \texttt{Stb}\,\texttt{V}\,\sigma$$
where
$$\texttt{Stb}\,E\,\sigma \stackrel{def}{=} \forall\, e \in E, \forall\, n \bullet \sigma_n(e) = \sigma_0(e)$$

The assignment process must satisfy the appropriate timing constraint, in addition to assigning the value of an expression to a variable. It also carries the unspoken assumption that the part of the internal state not explicitly changed remains constant.

$$\mathcal{M}^{\mathrm{p}}\,(x\ \texttt{:=}_t\ e)\,\sigma \stackrel{def}{=} (x :=_{0,t} (\mathcal{M}^{\mathrm{e}}\ e))\sigma$$
where
$$x :=_{t1,t2}\ e\,\sigma \stackrel{def}{=} (\texttt{last}\ \sigma)x = e(\texttt{init}\ \sigma) \wedge (t1 \leq \texttt{len}\ \sigma \leq t2) \wedge (\texttt{Stb}\ (\texttt{V} - \{x\})\,\sigma)$$

Input and output are special forms of assignment.

Assignment is easily generalised to sequences of variables and expressions, and a delay, then, is just an assignment with null arguments:

$$\texttt{Dly}_{\,t1,t2} \stackrel{def}{=} \langle\rangle :=_{t1,t2} \langle\rangle$$

The sequential composition of two processes is true on an interval if it can be chopped into two subintervals, such that the first process holds on the prefix and the second on the corresponding suffix. Alternatively, the

first process may be non-terminating.

$$\begin{array}{l}\mathcal{M}^{\mathrm{P}}\,(p1\,;\,p2)\,\sigma \stackrel{def}{=} (\mathcal{M}^{\mathrm{P}}\,p1\;;\;\mathcal{M}^{\mathrm{P}}\,p2)\sigma \\ \text{where} \\ (p1\,;p2)(\sigma) \stackrel{def}{=} p1(\sigma) \wedge \mathtt{len}\,(\sigma) = \infty \;\vee \\ \qquad\qquad \exists\, i \bullet p1(\mathtt{pfx}\;i\;\sigma) \wedge p2(\mathtt{sfx}\;i\;\sigma)\end{array}$$

The time-bounded conditional is defined in terms of a simple conditional with appropriate delays inserted.

$$\begin{array}{l}\mathcal{M}^{\mathrm{P}}\,(\mathtt{IF}_{t1,t2}\;b\;p1\;p2)\,\sigma \stackrel{def}{=} \\ ((\mathtt{Dly}_{\,0,t1}\;;p1\,;\mathtt{Dly}_{\,0,t2}) \lhd\; b \;\rhd (\mathtt{Dly}_{\,0,t1}\;;p2\,;\mathtt{Dly}_{\,0,t2}))\sigma \\ \text{where} \\ (p1 \lhd\; b \;\rhd p2)\sigma \stackrel{def}{=} (b(\mathtt{init}\;\sigma) \Rightarrow p1\,\sigma) \wedge (\neg b(\mathtt{init}\;\sigma) \Rightarrow p2\;\sigma)\end{array}$$

A simple loop predicate may be defined in the conventional way as the least fixed point of the function $\lambda\, X \bullet (p\,;X) \lhd\; b \;\rhd \mathtt{Skip}$.

$$\mathtt{Loop}\;\; b\;p \stackrel{def}{=} \mathtt{FIX}\,(\lambda\, X \bullet (p\,;X) \lhd\; b \;\rhd \mathtt{Skip})$$

where $\mathtt{FIX}\;F$ denotes the least fixed point of a function F under the refinement ordering defined by $\sqsupseteq$, where

$$p1 \sqsupseteq p2 \stackrel{def}{=} \forall\, \sigma \bullet p1\;\sigma \Rightarrow p2\;\sigma$$

Using the choice operator, the least fixed point may be defined non-constructively in HOL and, when F is continuous, the fixed point may be proved equivalent to a limit of an approximating chain in the standard way. Since all SAFE constructs are continuous, the loop is indeed equal to the limit of iteration, as desired, and this has been verified using HOL.

The time-bounded while loop is constructed from the simple loop and appropriate delays.

$$\mathcal{M}^{\mathrm{P}}\,(\mathtt{WHILE}_{t1,t2}\;b\;p)\,\sigma \stackrel{def}{=} (\mathtt{Loop}\;\; b\;(\mathtt{Dly}_{\,1,t1}\;;p\,;\mathtt{Dly}_{\,0,t2})\,;\mathtt{Dly}_{\,1,t1})\sigma$$

Note that the evaluation delay is of minimum length 1. This constraint avoids the possibility of having a non-terminating zero-length loop and is similar to the so-called 'non-Zeno' condition.

5.1 Algebraic laws

A library of algebraic laws which are generally useful in proofs concerning the programming language has been formulated and proved correct. Below are a few examples of such laws.

The empty interval `Skip` is 'better' (w.r.t. the refinement ordering relation $\sqsubseteq$) than an interval during which an expression E must remain stable:

$$\texttt{Skip} \;\sqsupseteq\; \texttt{Stb}\,E$$

The conjunction of two stability constraints is equivalent to a single constraint on the union of the arguments:

$$\texttt{Stb}\,(E1 \cup E2) \;=\; \texttt{Stb}\,E1 \wedge \texttt{Stb}\,E2$$

The conjunction of a stable interval distributes through the sequential composition of two programs:

$$(p1\,;p2) \wedge \texttt{Stb}\,E \;=\; (p1 \wedge \texttt{Stb}\,E)\,;(p2 \wedge \texttt{Stb}\,E)$$

An empty interval sequentially composed before or after a program is indistinguishable from the original program:

$$p\,;\texttt{Skip} \;=\; p \;=\; \texttt{Skip}\,;p$$

Sequential composition is associative:

$$p1\,;(p2\,;p3) \;=\; (p1\,;p2)\,;p3$$

All of these and many other laws have been formally derived in HOL from the interval semantics described above. See [7] for further details.

The law (actually a "law schema") for composing assignments is more complex than might initially be expected because of the need to treat both I/O and internal variables in a consistent way. However, a function has been implemented in HOL to take care of the complexity and automatically compose assignments using this law. The function is perfectly rigorous; it formally proves the required result. For example, given the composition $x :=_{t1,t2} m\,;y :=_{t3,t4} x+n$, the function will prove the theorem (assuming $x \neq y$):

$$x :=_{t1,t2} m\,;y :=_{t3,t4} x+n \;\;\sqsupseteq\;\; x,\; y :=_{t1+t3,t2+t4} m,\; m+n$$

The function may fail if the final values of the variables on the left of the assignment are indeterminate. If r is an input port, it will not reduce the composition $x :=_{t1,t2} e\,;y :=_{t3,t4} r$ without further information about the input value.

These laws allow proofs about the compiler specification to be conducted at a higher level than would otherwise be the case by remaining in the framework of the programming language itself where possible. They could also be used for other purposes (e.g. program transformation for optimisation).

6 Assembly Language

The machine has three registers, `A`, `B`, and `C`, a program pointer, `P`, and an addressable memory. An instruction *ins* has the effect of an assignment in which the machine state becomes some function of the old state.

$$state :=_{T(ins)} f(state)$$

Thus, the meaning $\mathcal{M}^{\mathrm{i}}\, ins$ of an instruction *ins* may be defined in terms of the interval semantics above.

For example, the `LDC` instruction pushes a constant w onto the three-register stack:

$$\mathcal{M}^{\mathrm{i}}\,(\mathtt{LDC}\ \ w) \stackrel{def}{=} \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{P} :=_{T(\mathtt{LDC})}\ w, \mathtt{A}, \mathtt{B}, \mathtt{P}+1$$

The `JMP` instruction performs a jump by updating the program counter relation to the location after the instruction:

$$\mathcal{M}^{\mathrm{i}}\,(\mathtt{JMP}\ \ w) \stackrel{def}{=} \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{P} :=_{T(\mathtt{JMP})}\ \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{P}+w+1$$

`ADD` (for example) is a two-operand instruction that pops the two input values off the register stack (i.e. from registers A and B) and pushes the result back onto the stack (i.e., into register A):

$$\mathcal{M}^{\mathrm{i}}\,\mathtt{ADD} \stackrel{def}{=} \mathtt{A}, \mathtt{B}, \mathtt{P} :=_{T(\mathtt{ADD})}\ \mathtt{A}+\mathtt{B}, \mathtt{C}, \mathtt{P}+1$$

All the instructions for the machine have been specified in this manner [7]. Note that we ignore the variable length of optimised Transputer instructions for simplicity, but this has been handled elsewhere [3, 11, 34].

The behaviour of an assembly language program stored between locations $n1$ and $n2$ in `ROM` is just the combined effect of running the sequence of instructions between these locations:

$$\mathtt{Run}\,\mathtt{ROM}\ n1\ n2 \stackrel{def}{=} \mathtt{Loop}\ (n1 \leq \mathtt{P} < n2)\ (\mathcal{M}^{\mathrm{i}}\,(\mathtt{ROM}\,(\mathtt{P})))$$

6.1 Z semantics

A machine semantics for a subset of the Transputer instruction set at the bit-level has also been specified using the formal Z notation [3, 51]. Z is more readable that HOL since it is designed for this purpose, so a Z specification was produced as an experiment in producing a formal description that is readable enough to be used as documentation for a suitably trained engineer. For example, the state may be recorded in a *schema* box as follows:

State

$$\mathtt{A}, \mathtt{B}, \mathtt{C} : \mathit{Value}$$
$$\mathtt{P} : \mathit{Address}$$
$$\mathtt{ROM}, \mathtt{RAM} : \mathit{Address} \pfun \mathit{Value}$$
$$\mathit{clock} : \mathbb{N}$$
$$\mathrm{dom}\,\mathtt{ROM} \cap \mathrm{dom}\,\mathtt{RAM} = \emptyset$$

The address space of the program memory (`ROM`) and data memory (`RAM`) at modelled as partial functions and their domains must not overlap. The number of *clock* cycles since initialisation is also recorded as a natural number. During a change of state the program in ROM remains the same and an instruction always takes a non-zero number of clock cycles to execute:

Δ*State*

$$\mathit{State}$$
$$\mathit{State}'$$
$$\mathit{cycles} : \mathbb{N}_1$$
$$\mathtt{ROM}' = \mathtt{ROM}$$
$$\mathit{clock}' = \mathit{clock} + \mathit{cycles}$$

Schemas may be 'included' within other schemas and the after state is indicated by dashed variables by convention. The `LDC` instruction may then be specified as follows:

LDC

$$\Delta\mathit{State}$$
$$w : \mathit{Value}$$
$$(\mathtt{LDC}\,, w) = M\,(\mathtt{ROM}\,(\mathtt{P}))$$
$$(\mathtt{A}', \mathtt{B}', \mathtt{C}', \mathtt{P}') = (w, \mathtt{A}, \mathtt{B}, \mathtt{P}{+}1)$$
$$\mathtt{RAM}' = \mathtt{RAM}$$
$$\mathit{cycles} = T(\mathtt{LDC}\,)$$

M and T are functions that decode a binary opcode and return the timing for an instruction respectively. Note that `LDC` does not update the RAM contents.

Such a specification could be of use to a microprocessor designer, acting as the specification of the processor to be implemented. More recently we have investigated the embedding of Z within HOL which enables the efficient mechanisation of proofs about Z specifications [10].

7 Compiler Specification

A compiler can be specified as a relation defined recursively over the syntax of commands and expressions.:

$$\mathcal{C}^{\mathrm{p}}\ q\ S\ n1\ n2\ \mathtt{ROM}$$

The relation is true if the instruction store `ROM` contains the compiled code for the process q starting at location $n1$ up to (but not including) location $n2$ under the symbol table S. We define the constraints on this relation for each high-level program construct. For example, `SKIP` may be implemented by simply making the finish address $n2$ of the matching object code the same as the start address $n1$:

$$\mathcal{C}^{\mathrm{p}}\ \mathtt{SKIP}\ S\ n1\ n2\ \mathtt{ROM} \stackrel{def}{=} n2 = n1$$

Sequential composition is implemented by placing the respective object code segments for the two compiled sub-programs contiguously together in the `ROM`:

$$\mathcal{C}^{\mathrm{p}}\ (p1\,;\ p2)\ S\ n1\ n2\ \mathtt{ROM} \stackrel{def}{=} \exists\, n \bullet \mathcal{C}^{\mathrm{p}}\ p1\ S\ n1\ (n1+n)\ \mathtt{ROM} \wedge \mathcal{C}^{\mathrm{p}}\ p2\ S\ (n1+n)\ n2\ \mathtt{ROM}$$

n is the offset from the beginning of the code of the location at which the two pieces of object code abut and is existentially quantified so that it is not directly visible in the combined object code.

Similar constraints have been formulated for all the program constructs and the compilation relation may be 'executed' (by theorem proving) in HOL to obtain the compiled code. Details are to be found in [7]. For those instructions that have time constraints, the total time required for the relevant sequence of instructions is checked. If this exceeds the bound, the compilation relation is false and the code fails to compile.

7.1 Correctness of compilation

The compilation of a process is correct if the behaviour of the compiled code is an implementation of the program behaviour. If the code for q is compiled in `ROM` between $n1$ and $n2$ using a symbol table S, then the execution of the instruction between $n1$ and $n2$ implements the behaviour of q under the data representation $\Theta\, S$:

$$\forall\, S\ n1\ n2\ \mathtt{ROM} \bullet \mathcal{C}^{\mathrm{p}}\ p\ S\ n1\ n2\ \mathtt{ROM} \Rightarrow (\mathtt{P} :=_{0,0} \dot{n1}\,;\ \mathtt{Run}\ \mathtt{ROM}\ n1\ n2 \sqsupseteq (\mathcal{M}^{\mathrm{p}}\ p) \circ (\Theta\, S)\,;\ \mathtt{P} :=_{0,0} \dot{n2})$$

The compilation scheme specified for the SAFE language has been mechanically proved correct w.r.t. this correctness criterion using algebraic

laws as previously described within the HOL theorem prover. The approach for the verification draws on the ideas of Hoare [32]. A similar approach has been adopted by the **ProCoS** project, although there the proofs have largely been undertake by hand only [34].

Here we have presented a simple unoptimised compilation scheme which is desirable in a high integrity system to avoid errors. However optimisation is important in general and more efficient compilation strategies may be attempted in a similar setting by adding further allowed compilation relations (even including extra program constructs and machine instructions) without necessarily invalidating those that have already been proved correct [26].

7.2 Rapid prototype compilation

It is relatively straightforward to produce a compiler which matches a compilation scheme such as that presented here very directly using a logic programming language like Prolog [4]. The formal compilation description for each programming construct may be implemented as a Horn Clause. For example, the clauses for `SKIP` and sequential composition may be implemented as follows in Prolog:

```
cp(skip,S,N1,N2,ROM) :- N2=N1.

cp(P1;P2,S,N1,N2,ROM) :-
    cp(P1,S,N1,N1plusN,ROM), cp(P2,S,N1plusN,N2,ROM).
```

What is more, it is even possible to produce a *decompiler* which takes the object code (and the symbol table if available) and produces a matching high-level program for non-optimised code [6]. This could be useful in the verification and checking of safety-critical code where optimisation is normal avoided anyway. A logic program specifies a relation in general so such a program may also be used as a compiler *checker* taking a given program, symbol table and matching object code and checking that they are compatible with a formally specified and proven compiling relation. Tools to help verify termination and the validity of the omission of the occurs-check for Prolog implementations for such prototype compilers have been produced elsewhere [38].

8 Microprocessor Design

A machine code program must be executed on some underlying hardware, normally based around a microprocessor in embedded real-time systems. Important considerations for the microprocessor design are that it should be predictable and extensible, and the mathematical models used must support these aims.

Predictable performance is important for hard real-time applications and this goal is met by using a synchronous memory, and by using models

incorporating explicit time for the basic components. This means that the performance of the processor can be calculated in units corresponding to the period of the underlying synchronous clock. For example, a loadable register is defined in HOL as:

```
LOAD_REG(in:time->*, load, out)  def=
    (∀ t. out (t+1) = (load t => in t | out t))
```

The explicit timing information at the lowest level is then propagated up through the higher levels yielding hard real-time information at these levels.

8.1 Support for incremental design

The goal of an extensible design is supported by the incremental framework developed and used in the safemos project [29]. This builds on the idea in [53] for a general framework supporting the formal specification and verification of a range of processors. To support formal methods during the design process the models and techniques allow partial specification, extensions to a design and incremental verification of a design. The main elements of the framework are:

- uniform hierarchy of computation models;
- use of generic arguments;
- verification template;
- the use of incremental models and techniques.

The hierarchy of computation models follows the idea of a hierarchy of interpreters in [1], where an instruction at each level is interpreted by executing a series of instructions at a lower level, and these instructions are in turn executed by an interpreter at the next lower level. Generic arguments represent certain aspects of a design which can remain unspecified and be parameters of the design. For example, rather than specifying a 16-bit processor, the word size may be supplied as a parameter of the design; then a generic n-bit processor may be verified instead.

Having a general hierarchical model of computation such as an interpreter means that templates for specifying the instructions and interpreter can be provided; these encapsulate in a general theorem the combination of individual instruction correctness results at one level to derive correctness of the interpreter at the next higher level. To support flexibility in the specification and verification of the design, incremental models have been introduced replacing the previous functional interpreter models.

8.2 Relational interpreter framework

The basic model of system behaviour is a transition system where the next state is derived from the current state and environment (inputs). In a *relational interpreter* the effect of a certain transition can be defined as a predicate on the next state and the present state and environment. The

behaviour of a transition system can be specified by the desired properties given as a set of (tag, predicate) pairs, and a selection function that indicates whether a particular property tag is chosen by a state and environment. The transition system for a certain set of properties must ensure that whenever the state and environment select a transition in the set then all predicates for the indicated transition must hold on the next state and the current state and environment. The definition in HOL is:[2]

```
TRANSITION_SYS(is_selected,prop_list)s e  def=
  (∀t.
    let a_tag = εtag. is_selected(tag,s t,e t)
    in
     (∀prop.
       prop MEM prop_list ∧ (FST prop = a_tag) ⟹
         SND prop(s(t + 1),s t,e t)))
```

The definition of `TRANSITION_SYS` gives an incremental model since more items can be incrementally added to the property list parameter as the specification is extended with more behaviour.

In addition to the specification of the microprocessor as relational interpreter using `TRANSITION_SYS`, an incremental model for the machine microcode has been devised. This allows the derivation of the following result:

```
ALL_UNIQUE (APPEND mcode1 mcode2) ⟹
  ROM(addr,out)(APPEND mcode1 mcode2) ⟹
    ROM(addr,out)mcode1
```

This states that subject to the condition `ALL_UNIQUE` which demands that no entries clash, a microcode ROM extended with the microcode set `mcode2` allows only behaviour which is compatible with the original set `mcode1`.

8.3 Verification

The relational interpreter framework and incremental implementation models have been developed in the context of doing a simple processor design [29]. As described in section 6, the state consists of a tuple of program counter `P`, three-element stack (`A`,`B`,`C`) and memory `ROM`, and there is an external signals environment represented by a variable `env`. For example, the semantics for the `LDC` instruction presented in section 6 can be described in HOL as follows:

---

[2] Note that `a MEM list` is true if `a` is a member of list `list`; `FST` and `SND` access the first and second elements of a tuple. `is_selected(tag,s t,e t)` is true at time `t`, if `tag` is selected in state `s t` and environment `e t`. ε is Hilbert's choice operator in HOL. If a unique tag is chosen then the term `εtag. is_selected(tag,st,env)` (i.e. choose the tag such that ...) evaluates to this tag.

```
LDC_SEM gen_rep ((P´,A´,B´,C´,ROM´),(P,A,B,C,ROM),env) =
  (let instr = ROM((ADDR_FN gen_rep)P)
    in
   let w = (ARG_FN rep)instr
    in
    ((P´ = (ADD1 gen_rep)P) ∧
     (A´ = w) ∧ (B´ = A) ∧ (C´ = B) ∧ (ROM´ = ROM)))
```

where `gen_rep` contains the generic parameters of the design.

A simple microcoded machine has been designed to implement a complete instruction set [29], and has been shown to correctly implement a relational interpreter for the desired instructions.[3] The HOL description takes the following form:

```
(MICRO_MC MLIST_A gen_rep) ⟹
    TRANSITION_SYS
    (IS_SELECT gen_rep,
    [LDC_op, LDC_SEM gen_rep;
     JMP_op, JMP_SEM gen_rep;
     ADD_op, ADD_SEM gen_rep;
     ... ])
    ((SIG_TUP5(PC,A,B,C,mem)) o
         (Temp_Abs(λ t. mp_index t = 0)))
    (e o (Temp_Abs(λ t. mp_index t = 0)))
```

The microcoded machine is described by `MICRO_MC` and is parameterised by the microcode `MLIST_A`. This means that the incremental models of the implementation and abstract machine can be used in tandem. The machine specification can be extended by adding more instructions to the property list argument of `TRANSITION_SYS`, and the microcode can be extended to implement these instructions thus extending the microcode parameter of `MICRO_MC`. The use of incremental models means that just these extensions need be verified, the previous results being inherited, and the new system can be verified in an efficient manner.

8.4 Inmos processor

In addition to the simple processor for the restricted machine instruction set, a more realistic processor for an enlarged instruction set has been designed and verified using HOL by David Shepherd at Inmos [7]. This was an important part of the project to demonstrate the usefulness of the formal methods developed by the more academic partners in an industrial setting. Inmos have been enthusiastic protagonists of formal methods: in the past they have won a UK Queen's Award for Technological Achievement

[3] This result is subject to the synchronisation condition as described in [29].

jointly with Oxford University for their work on formally developing the microcode for the floating point unit of the T800 Transputer [49] and more recently they have applied formal techniques to critical parts of the T9000 Transputer such as the pipeline architecture and the associated virtual channel processor [39, 46]. The Inmos design produced on the safemos project is intended to be realistic in performance and complexity, as well as predictable and extensible.

The mathematical models underlying the Inmos processor are based on those described for the simple processor presented here, and thus support a generic, incremental approach to design specification and verification. In addition, there is an emphasis on automated proof using HOL which has resulted in the architectural approach employing generic parameters and a microcode verification tool as described in [7].

9 Normal Form and Hardware Compilation

An alternative and novel approach to the implementation of a program on a microprocessor is to compile the program directly into hardware [44]. This technique was not foreseen at the start of the project, and as such is more speculative than the work presented in previous sections, but is presented here because we believe this will be an important development for the future. It has been made possible in a practical sense for small programs (such as those found in embedded systems) by the fast developing and exciting technology of Field Programmable Gate Arrays (FPGAs) which allows the configuration of a hardware circuit to be defined by binary code in a memory in a similar manner to more conventional object code.

The project has investigated an approach to such compilation via a *normal form* [35] which uses a very restricted subset of the high-level programming language being compiled. This splits the compilation process in two which both simplified the individual transformations between the levels and allows the possibility of compiling the normal form to either hardware or software or even a combination of the two.

A normal form program comprises three sequential programs where the first one designates the initial control state of the circuit normally taken from the environment (an assumption), and the last one the final state normally returned back to the environment (an assertion). The middle one is a loop with a multiple assignment as its body which specifies state change of the circuit during execution.

$$
\begin{array}{l}
\mathcal{N}(n1,\ n2,\ K,\ C,\ V) \stackrel{def}{=} \\
\quad \mathtt{LOCAL}\ c\ (c :=_0\ n1\ ; \\
\quad \mathtt{WHILE}\,(c \in K)\,(c,\ v\ :=_1\ C(c),\ V(c))\ ; \\
\quad (c\ =\ n2)_{\perp})
\end{array}
$$

$n1$ indicates the starting control state, $n2$ indicates the finishing control

state, K is a set of possible control states, C a K-indexed family of expressions describing the next control state and V a K-indexed family of expressions specifying the new value of data path v. In a circuit, control states are normally recorded in wires, possibly connected to latches, whereas in a microprocessor this information is typically recorded by a special register known as the program counter.

9.1 Normal form reduction theorems

As in section 7, each high-level programming construct must be handled. As before the finishing control state should be the same as the starting control state for `SKIP` and there is no change of state of the variables in the program:

$$\texttt{SKIP} \sqsubseteq \mathcal{N}(n,\, n,\, \emptyset,\, \emptyset,\, \emptyset)$$

Typically, the low-level implementation for this would be a single wire labelled by n, taking no time to 'execute'.

For assignment, the control flow is recorded by the C parameter and the new state of the data is recorded by the V parameter. Let $t \geq 1$:

$$v :=_t e \sqsubseteq \mathcal{N}(n1,\, n2,\, \{n1\},\, \{n1 \mapsto n2\},\, \{n1 \mapsto e\})$$

In a clocked hardware implementation, t would indicate the number of clock cycles used to perform the update and the state would be held in a number of latches. A minimum of one clock cycle is required to allow time for the evaluation of the expression e in case this has changed during the previous cycle.

The composition of two circuits already in valid normal form is implemented by taking the union of the K, C and V parameters. The control states in the two circuits must be suitably disjoint. If $K1 \cap K2 = \emptyset$ and $n2 \notin K1$:

$$\begin{aligned}&\mathcal{N}(n1,\, n,\, K1,\, C1,\, V1)\,;\mathcal{N}(n,\, n2,\, K2,\, C2,\, V2)\\ &\quad\sqsubseteq \mathcal{N}(n1,\, n2,\, K1 \cup K2,\, C1 \cup C2,\, V1 \cup V2)\end{aligned}$$

The intermediate control state n is effectively absorbed by the circuit. Typically this would be implemented as an internal wire in the circuit directly connecting the output control state of the first circuit to the input control state of the second one.

A more complete set of theorems for a simple sequential programming language may be found in [27]. A pleasing feature of the approach if mapped to synchronous clocked circuits is that the timing properties of programs are very simple. For example, it is possible to execute all assignment statements in a single clock cycle (if made long enough to accommodate the slowest expression or control state to be evaluated) and for all

control constructs to effectively take no clock cycles since control states are determined in parallel to other computation during each clock cycle [44].

Hardware compilation is an active area of research in which we foresee much possible progress, the fusion of mathematical techniques with the practical concerns of engineers, and also an increased potential and desire for hardware/software co-design. The natural parallelism of hardware can be exploited to the full. Many algorithms which would be too slow if implemented using a microprocessor but are too expensive or would be too inflexible if implemented in custom hardware could benefit from such an approach. In addition, the number of levels of abstraction to be handled is reduced, thus helping to decrease the overall possibility for error.

10 Conclusion

This paper has briefly presented a number of approaches investigated by the safemos project to aid the verification of mixed hardware/software systems, especially considering real-time aspects where possible and appropriate. Most of the areas covered are elaborated further in a forthcoming book [7] and other papers referenced here.

A number of formalisms were investigated of the project and not all the parts presented are connected in a satisfactory way. Further work to unify the approaches at different levels of abstraction is certainly required. For example, both HOL and Z were used on the project but the use of HOL was prevalent because of the desire to mechanise proofs. There is now more support for the mechanisation of proofs in Z [10] so were the project to have started today, more use of Z might have been possible with the same goals of mechanisation in mind.

Whilst the safemos project has now formally finished, further collaborative work on the support of Z using HOL has continued [10] and the related **ProCoS** project continues to aim for the goal of connecting different levels of abstraction in the development process in a mathematical manner. In particular, the approach presented in section 9 looks like a promising one for the development of a provably correct combined hardware/software compiler, thus helping to straddle the development of hardware and software in a unified framework.

Acknowledgements

The research described here was largely supported by the UK Information Engineering Directorate safemos project (IED3/1/1036), which was jointly funded by the UK government DTI (Department of Trade and Industry) and the SERC (Science and Engineering Research Council). We are grateful for the contributions of others involved with the safemos project whose names may be found in [7]. In particular, Mike Gordon (University of Cambridge) and David Shepherd (Inmos) were other key members of

the project. Prof. Tony Hoare (Oxford University) and Prof. David May (Inmos) were inspirational in the work undertaken. Finally, John Buckle was an extremely helpful monitoring officer for the project.

Jonathan Bowen is currently funded by the newly formed Engineering and Physical Sciences Research Council (EPSRC) on grant no. GR/J 15186. He Jifeng has been funded by the ESPRIT Basic Research **ProCoS** project and its follow-on (nos. 3104 and 7071).

Bibliography

1. F. Anceau. *The Architecture of Microprocesors.* Addison-Wesley Publishing Company, Wokingham, 1986.
2. P.D. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Computer Science and Applied Mathematics Series. Academic Press, 1986.
3. J.P. Bowen. Formal specification of the ProCoS/safemos instruction set. *Microprocessors and Microsystems*, 14(10):631–643, December 1990.
4. J.P. Bowen. From programs to object code using logic and logic programming. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques, Proc. International Workshop on Code Generation*, Workshops in Computing, pages 173–192. Springer-Verlag, 1992.
5. J.P. Bowen. Formal methods in safety-critical standards. In *Proc. 1993 Software Engineering Standards Symposium*, pages 168–177. IEEE Computer Society Press, 1993.
6. J.P. Bowen. From programs to object code and back again using logic programming: Compilation and decompilation. *Journal of Software Maintenance: Research and Practice*, 5(4):205–234, December 1993.
7. J.P. Bowen, editor. *Towards Verified Systems.* Real-Time Safety Critical Systems Series. Elsevier Science Publishers, 1994.
8. J.P. Bowen et al. A ProCoS II project description: ESPRIT Basic Research project 7071. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 50:128–137, June 1993.
9. J.P. Bowen, M. Fränzle, E.-R. Olderog, and A.P. Ravn. Developing correct systems. In *Proc. 5th Euromicro Workshop on Real-Time Systems*, pages 176–187. IEEE Computer Society Press, 1993.
10. J.P. Bowen and M.J.C. Gordon. Z and HOL. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 141–167. Springer-Verlag, 1994.
11. J.P. Bowen, He Jifeng, and P.K. Pandya. An approach to verifiable compiling specification and prototyping. In P. Deransart and J. Małuszyński, editors, *Programming Language Implementation and*

Logic Programming, volume 456 of *Lecture Notes in Computer Science*, pages 45–59. Springer-Verlag, 1990.

12. J.P. Bowen and V. Stavridou. The industrial take-up of formal methods in safety-critical and other areas: a perspective. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 183–195. Springer-Verlag, 1993.
13. J.P. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993.
14. S.M. Brien and J.E. Nicholls. Z base standard. Technical Monograph PRG-107, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK, November 1992. Accepted for ISO standardization, ISO/IEC JTC1/SC22.
15. J.A. Camilleri. Symbolic compilation and execution of programs by proof: A case study in HOL. Technical Report 240, University of Cambridge, Computer Laboratory, UK, December 1991.
16. A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.
17. D.I. Good and W.D. Young. Mathematical methods for digital system development. In S. Prehn and W.J. Toetenel, editors, *VDM'91: Formal Software Development Methods, Volume 2*, volume 552 of *Lecture Notes in Computer Science*, pages 406–430. Springer-Verlag, 1991.
18. M.J.C. Gordon. HOL: A proof generating system for Higher-Order Logic. In G. Birtwistle and P.A. Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.
19. M.J.C. Gordon. A formal method for hard real-time programming. In J.M. Morris and R.C. Shaw, editors, *Proc. 4th Refinement Workshop*, Workshops in Computing. Springer-Verlag, 1991.
20. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A Theorem-proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
21. M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
22. R.W.S. Hale. Programming in Temporal Logic. Technical Report 173, University of Cambridge, Computer Laboratory, UK, 1989.
23. R.W.S. Hale, R. Cardell-Oliver, and J.M.J. Herbert. An embedding of Timed Transition Systems in HOL. *Formal Methods in System Design*, 3:151–174, August 1993.

24. J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals. In *Proc. 10th International Colloquium on Automata, Languages and Programming, Barcelona, Spain*, 1983.

25. He Jifeng and J.P. Bowen. Time interval semantics and implementation of a real-time programming language. In *Proc. 4th Euromicro Workshop on Real-Time Systems*, pages 110–115. IEEE Computer Society Press, 1992.

26. He Jifeng and J.P. Bowen. Specification, verification and prototyping of an optimized compiler. *Formal Aspects of Computing*, to appear.

27. He Jifeng, I. Page, and J.P. Bowen. Towards a provably correct hardware implementation of Occam. In G.J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods (CHARME'93)*, volume 683 of *Lecture Notes in Computer Science*, pages 214–225. Springer-Verlag, 1993.

28. T.A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J.W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

29. J.M.J. Herbert. Incremental design and formal verification of microcoded microprocessors. In V. Stavridou, T.F. Melham, and R.T. Boute, editors, *Theorem Provers in Circuit Design*, IFIP Transactions A-10, pages 157–174. North-Holland, 1992.

30. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, October 1969.

31. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, 1985.

32. C.A.R. Hoare. Refinement algebra proves correctness of compiling specifications. In C.C. Morgan and J.C.P. Woodcock, editors, *3rd Refinement Workshop*, Workshops in Computing, pages 33–48. Springer-Verlag, 1991.

33. C.A.R. Hoare and M.J.C. Gordon, editors. *Mechanized Reasoning and Hardware Design.* Prentice Hall International Series in Computer Science, 1992.

34. C.A.R. Hoare, He Jifeng, J.P. Bowen, and P.K. Pandya. An algebraic approach to verifiable compiling specification and prototyping of the ProCoS level 0 programming language. In *ESPRIT '90 Conference Proceedings*, pages 804–818. Kluwer Academic Publishers, 1990.

35. C.A.R. Hoare, He Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.

36. INMOS Limited. *Occam 2 Reference Manual.* Prentice Hall International Series in Computer Science, 1988.

37. INMOS Limited. *Transputer Instruction Set: A compiler writer's guide*. Prentice Hall, 1988.
38. M.R.K. Krishna Rao, P.K. Pandya, and R.K. Shyamasunder. Verification tools in the development of provably correct compilers. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 442–461. Springer-Verlag, 1993.
39. D. May, G. Barrett, and D.E. Shepherd. Designing chips that work. In Hoare and Gordon [33], pages 3–19.
40. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.
41. MoD. The procurement of safety critical software in defence equipment (part 1: Requirements, part 2: Guidance). Interim Defence Standard 00-55, Issue 1, Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown Street, Glasgow G2 8EX, UK, 5 April 1991.
42. J.S. Moore et al. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, December 1989.
43. B.C. Moszkowski. A Temporal Logic for multi-level reasoning about hardware. *IEEE Computer*, 18(2):10–19, February 1985.
44. I. Page and W. Luk. Compiling Occam into field-programmable gate arrays. In W. Moore and W. Luk, editors, *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, pages 271–283. Abingdon EE&CS Books, 15 Harcourt Way, Abingdon OX14 1NV, UK, 1991.
45. L.C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.
46. A.W. Roscoe. Occam in the specification and verification of microprocessors. In Hoare and Gordon [33], pages 137–151.
47. A.W. Roscoe and C.A.R. Hoare. Laws of Occam programming. *Theoretical Computer Science*, 60:177–229, 1988.
48. SAFEMOS: Demonstration of the possibility of totally verified systems. Proposal for an IED Research Project, 1989. INMOS Ltd, SRI International Cambridge Computer Science Research Center, Oxford University Computing Laboratory and Cambridge University Computer Laboratory.
49. D.E. Shepherd. Verified microcode design. *Microprocessors and Microsystems*, 14(10):623–630, 1990.
50. J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

51. J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.
52. SRI International Cambridge Research Center and DSTO Australia. *The HOL System: Description, Tutorial, Libraries, Reference Manual*, 1991. Revised version, four volumes.
53. P.J. Windley. A hierarchical methodology for verifying microprogrammed microprocessors. Technical Report CSE-89-27, University of California, Davis, 1989.