Formal Verification of Fault-Tolerant Software Design: The CSP Approach

W. L. Yeung
Lingnan University
Hong Kong

S. A. Schneider
Royal Holloway, University of London
United Kingdom

Abstract

Software design techniques for tolerating both hardware and software faults have been developed over the past few decades. Paradoxically, it is essential that fault-tolerant software is designed with the highest possible rigour to prevent faults in itself. Such rigour is provided by formal methods and aided by model checking. We illustrate an approach to fault-tolerant software design based on Communicating Sequential Processes (CSP) through a running example.

Keywords Fault tolerance; Formal verification; Model checking; Software design

1 Introduction

With ever more powerful microprocessors comes the need to put an ever increasing amount of functionality in our software. Yet computing systems from office desktops to stock market trading systems to aircraft navigation control systems are suffering from design faults in their software. Nor is the hardware totally exempted from design faults; and hardware is also susceptible to physical faults developed during operation. Formal methods play an important role in ensuring the reliability of computing systems by bringing mathematical rigour to software development. For instance, the UK Ministry of Defence places great emphasis on the use of formal development methods in its standards document 00-55 on the procurement of safety critical software [17]. Formal methods have also helped reduce overall costs in realistic commercial projects by uncovering defects

during early development steps [8].

A formal specification captures the functional requirements of a system concisely and unambiguously in the form of a mathematical expression that describes the required behaviour of the system in terms of its external interface (input/output) variables. Formal methods provide techniques and tools for the verification of a system design against its formal specification. The goal of formal verification is to eliminate any faults in the system design before it is implemented and put into operation. We call this the fault prevention approach to software reliability. However, as software becomes increasingly more complex, that goal is often too difficult to achieve. Furthermore, while computer hardware is considered very reliable nowadays, design faults are not restricted to software; "bugs" in today's VLSI/ULSI design could escape testing and go into production shipments (e.g. see [7] about a bug in the circuit design of the floating point unit of Intel's Pentium processor).

Fault tolerance is another important approach to building reliable systems. In this approach, redundancy is incorporated into a system to prevent failures during its operation in spite of faults in the system. As far as software faults are concerned, the goal of fault tolerance is to ensure that the software drives a system correctly according to its specification even if some (redundant) components of the software are not bug-free.

As often suggested in the literature, fault prevention and fault tolerance are complementary approaches and hence we should apply fault prevention techniques and particularly, formal methods to the design of fault-tolerant software. This is illustrated in this article with Communicating Sequential Processes, or CSP [10, 23, 27], which is a member of a class of formal methods known as process algebras. Using a running example of system design, we illustrate the effective use of CSP and its support tools in the development process. No prior knowledge of CSP is assumed for the reader.

The rest of this article is organised as follows. The next section introduces a subset of the CSP language used in this paper. Section 3 outlines the application of the CSP method to the program design for a microprocessor-based device. Section 4 describes a fault-tolerant version of the program design and highlights the role of formal verification in eliminating design faults. Section 5 considers the specification and verification of another version of the program design that tolerates software design faults during operation. Section 6 discusses the use of a model checking

tool as an automated means of verifying the fault-tolerant design. Sections 7 and 8 extend the design for a dual-processor implementation with enhanced fault-tolerance capability. Section 9 concludes with some remarks on the use of CSP in larger systems and other complexity issues.

2 The language of CSP

In the language of CSP, a process is described in terms of the possible interactions it can have with its environment, which may be thought of as another process or set of processes. Interactions are described in terms of instantaneous atomic synchronisations, or *events*. A process can be considered as a "black box" with an interface containing a number of events through which it interacts with other processes. The set of all events in the interface of a process P, written αP, is called its *alphabet*. It is important to note that interface events are intended as synchronisations between the participating processes and not as autonomous actions under the control of a single process.

The following paragraphs briefly introduce the CSP operators used in this paper. A comprehensive description of the language is found in [10, 22, 27]. The language of CSP used in this paper is defined by the following pseudo Backus-Naur form definitions:

$$
\begin{aligned}
P \quad ::= \quad & \mathit{Stop} \mid a \rightarrow P \mid P \,\Box\, P \mid P \sqcap P \mid a : A \rightarrow P_a \mid \\
& P \underset{A}{\parallel} P \mid P \setminus A \mid P // P \mid \\
& \mathit{Skip} \mid P;\ P \mid P^* \mid \text{if } b \text{ then } P \text{ else } P \text{ fi} \mid P \triangle_a P \mid \\
& P[\![a/a]\!]
\end{aligned}
$$

where Σ is the set of all possible events, a is in Σ, A in $\mathbb{P}(\Sigma)$, and b is a boolean expression.

The process *Stop* is the deadlocked process, unable to engage in any events or make any progress. The prefix process $a \rightarrow P$ is ready to engage in event a (and in no other event). It will continue to wait until its environment is also ready to perform a, at which point synchronisation on this event will occur. Once the event is performed, the subsequent behaviour of $a \rightarrow P$ will be that of process P. An external choice $P \,\Box\, Q$ is initially ready to engage in events that either P or Q is ready to engage in. The first event performed resolves the choice in favour of the component that

was able to perform it, and the subsequent behaviour is given by this component. In contrast, an internal choice $P \sqcap Q$ behaves either as P or as Q but, unlike the external choice, the environment cannot influence the way the choice is resolved.

The prefix choice $a : A \to P_a$ remains willing to perform any event from set A until one is chosen. Its subsequent behaviour, described by P_a, is dependent on that event. A construct can be defined ($\widehat{=}$) to allow the input on channel in of any item x in a set M, and the value x determines the subsequent behaviour:

$$in?x : M \to Q(x) \mathrel{\widehat{=}} a : in.M \to P_a$$

where the set $in.M = \{in.m \mid m \in M\}$ and $P_{in.m} = Q(m)$ for every $m \in M$. The atomic synchronisation events here are of the form $in.m$. The complement is the output prefix which has the form $out!x \to P$ and this is simply a shorthand for $out.x \to P$. The parallel combination $P \underset{A}{\parallel} Q$ synchronises on events in A, and interleaves on all other events. The hiding operator $P \setminus A$ makes the events in the set A internal to the process, thus removing them from the control of the environment. The only participants will then be the components of P. The asymmetric parallel combination $P//Q$ is a special form of the parallel combination in which Q is only capable of synchronising with P and all synchronisations between P and Q are hidden from the environment, ie.:

$$P//Q \mathrel{\widehat{=}} \left(P \underset{\alpha Q}{\parallel} Q \right) \setminus \alpha Q$$

where $\alpha Q \subseteq \alpha P$.

Skip is the process that does nothing but terminates successfully. In the sequential composition $P;\ Q$, the combined process first behaves as P and Q becomes active immediately after the successful termination of P. The indefinite loop construct P^* starts with P and repeats itself after every successful termination of P. The if b then P else Q fi construct selects either P or Q depending on the boolean expression b in the usual way. In $P \triangle_a Q$, the combined process behaves initially as P and is interruptible at any time with the occurrence of event a, after which Q becomes active. Finally, $P[\![c/a]\!]$ is the process P with every event a renamed as c. A special form of renaming "$c : P$" renames every event a in P as $c.a$.

$$
\begin{array}{rcl}
Prog & \widehat{=} & P^* \\
P & \widehat{=} & in?x : Float \rightarrow P(x) \\
P(x) & \widehat{=} & \left(\begin{array}{l} \text{if } x < 0 \text{ then} \\ \quad sub.in!(-x) \rightarrow sub.out?y : Float \rightarrow out!y \rightarrow Skip \\ \text{else} \\ \quad sub.in!x \rightarrow sub.out?y : Float \rightarrow out!y \rightarrow Skip \\ \text{fi} \end{array} \right) \; // sub : SQRT \\
SQRT & \widehat{=} & in?z : Float \rightarrow out!\texttt{sqrt}(z) \rightarrow Skip
\end{array}
$$

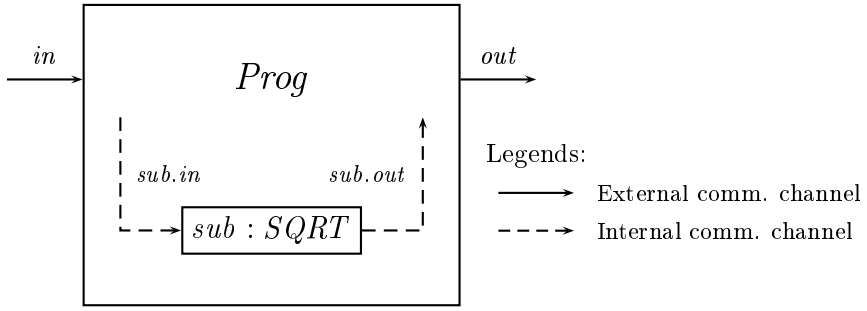

Figure 1: A (sequential) program design for the filter

3 A real-time DSP filter

A microprocessor-based digital signal processing (DSP) filter can be programmed to perform various signal processing functions on a stream of numerical input signals. We can specify a program for the filter to carry out a specific function using a specification written in the CSP language as follows:

$$Spec \mathrel{\widehat{=}} in?x : Float \rightarrow out!\sqrt{|x|} \rightarrow Spec$$

A CSP specification is composed of one or more (mutual-)recursively defined *processes*. *Spec* is the name given to the CSP process that specifies the program in terms of the ordering of the input and output operations to be carried out, as well as how input and output values relate to each other functionally. We shall consider the timing aspect later in Section 8. The CSP process is defined by the expression on the right-hand-side of the definition "$\widehat{=}$" symbol. *Spec* specifies that the filter initially inputs a floating point number x through channel *in* and then outputs $\sqrt{|x|}$ through channel *out*; after the output, the filter returns to its inital state and is ready for the next input. Note that, as a specification, *Spec* does not mention anything about the components or internal structure of the actual program.

We assume that a pre-written software library routine with the name `sqrt` is available for computing the square root function and can be invoked by a conventional subroutine call in the

code implementation. The program design involving input and output operations as well as the subroutine call and execution can be expressed as another CSP process *Prog* as shown in Figure 1. The program is designed as an indefinite loop (P^*), with P defined as a set of operations for processing one input. P begins with an input operation and then the processing of the input by $P(x)$, which involves two processes: the left-hand-side process (in brackets) is responsible for calling a subroutine with the absolute value of the input whereas the right-hand-side process ($sub : SQRT$) "models" the subroutine that receives the call and executes `sqrt`. The result returned from the subroutine is then output by the left-hand-side process. Note that the two processes are conceptually running in parallel and communicating with each other through channels $sub.in$ and $sub.out$—hence, the asymmetric parallel operator "//"—and both of them terminate after the processing of one input. Termination is modelled by the *Skip* process whereas the subroutine call "$sub.in!(-x) \rightarrow sub.out?y : Float$" is actually simulated by a pair of (synchronised) output and input operations between the calling process and the subroutine process. The asymmetric parallel operator "//" implicitly designates the simulated call and return as internal I/O operations to be performed without the participation or notice of the environment, as distinguished from the system's input and output operations.

It is the designer's obligation to verify that, with its internal behaviour hidden from the environment, the filter's observable behaviour as prescribed by *Prog* is consistent with that by *Spec*. This can be done through logical reasoning based on the mathematical theory of CSP and written down as a formal proof. Figure 2 shows the proof for our design.

3.1 Remarks on code implementation

There are some salient points about the verified program design in relation to its code implementation:

1. The design (and also its specification) assumes the set of floating point numbers (*Float*) as the only possible input values to the program. That means its code implementation is expected to be executed in an environment that will provide those input values only. If the implementation is expected to respond to other (invalid) input values from the environment, we would have to modify the design and revise the specification accordingly.

Lemma *Spec* = *Prog*.
Proof First, we rewrite *Prog* into a single CSP expression.

$$Prog = \left(\begin{array}{l} in?x : Float \rightarrow \left(\begin{array}{l} \left(\begin{array}{l} \text{if } x < 0 \text{ then} \\ \quad sub.in!(-x) \rightarrow sub.out?y : Float \\ \qquad \rightarrow out!y \rightarrow Skip \\ \text{else} \\ \quad sub.in!x \rightarrow sub.out?y : Float \\ \qquad \rightarrow out!y \rightarrow Skip \\ \text{fi} \end{array} \right) \\ // \\ sub : \left(\; in?x : Float \rightarrow out!\texttt{sqrt}(x) \rightarrow Skip \; \right) \end{array} \right) \end{array} \right) \; ; \; Prog$$

The following algebraic laws can then be applied to "simplify and eliminate" the internal subroutine call and return operations.

$$\begin{array}{rcl} (m.c!v \rightarrow P)//m : (c?x \rightarrow Q(x)) & = & P//m : Q(v) \\ (m.d?x \rightarrow P(x))//m : (d!v \rightarrow Q) & = & P(v)//m : Q \end{array}$$

Applying the above laws we get:

$$Prog = \left(\begin{array}{l} in?x : Float \rightarrow \left(\begin{array}{l} \left(\begin{array}{l} \text{if } x < 0 \text{ then} \\ \quad out!\texttt{sqrt}(-x) \rightarrow Skip \\ \text{else} \\ \quad out!\texttt{sqrt}(x) \rightarrow Skip \\ \text{fi} \end{array} \right) \\ // \\ sub : Skip \end{array} \right) \end{array} \right) \; ; \; Prog$$

We may further simplify the above expression based on the following algebraic laws concerning the *Skip* process and sequential composition.

$$\begin{array}{rcl} (a \rightarrow P)//m : Skip & = & a \rightarrow (P//m : Skip) \text{ if } a \text{ is not in the form } m.a \\ Skip//m : Skip & = & Skip \\ (a \rightarrow P);\; Q & = & a \rightarrow (P;\; Q) \\ Skip;\; P & = & P \end{array}$$

Apply these laws further, we have:

$$Prog = in?x : Float \rightarrow \left(\begin{array}{l} \text{if } x < 0 \text{ then} \\ \quad out!\texttt{sqrt}(-x) \rightarrow Prog \\ \text{else} \\ \quad out!\texttt{sqrt}(x) \rightarrow Prog \\ \text{fi} \end{array} \right)$$

Comparing the simplified definition of *Prog* with the definition of *Spec*, we can observe that they prescribe similar patterns of action *provided that* the `sqrt()` routine faithfully implements the $\surd$ function. This last step of proof of equivalence between *Prog* and *Spec* can be completed by appealing to the unique fixed points (UFP) rule which is explained in detail in [10, 23].

Figure 2: A proof of equivalence between *Prog* and *Spec*

2. The design is basically an infinite loop with no provision for any startup, shutdown, or suspension operations. If it is desirable to provide such operations in the code implementation, which may involve an operating system, care needs to be taken to ensure the proper ending of the processing cycle and flushing of the last output, etc.

3. The design has no provision for any kinds of exception or interrupt, e.g. numeric overflow. The code implementation may involve interrupt handling routines that need to be considered in the program design. Section 3 illustrates this with an example.

4. While the program design has been verified as correct, its code implementation will involve the code of the library routine `sqrt` which is assumed to implement the $\surd$ function faithfully and faultlessly. If this assumption cannot be guaranteed or verified, the entire code implementation is in doubt. In Section 4, we shall consider a fault-tolerant software design whose implementation involves redundant code.

4 Fault tolerance

The program design *Prog* assumes a faultless implementation which would correctly and reliably operate as intended. In reality, there are various faults that could arise in the implementation before or after it has become operational: the software routine `sqrt` may contains residual "bugs" and does not compute $\surd$ accurately enough; the hardware may suffer from component failures due to physical faults, e.g. the circuitry of the microprocessor may burn out due to excessive heat, or electricity glitches may cause transient faults in the RAM component, leading to incorrect computation.

Fault tolerance involves four phases, namely:

1. error detection,

2. damage confinement,

3. error recovery, and

4. fault treatment.

Consider the scenario in which the RAM component of our system is disturbed by electricity glitches, resulting in numerical computation errors. To detect such errors, we may choose to use a

RAM component with parity bits. Whenever a storage byte is accessed, its parity bit is checked for integrity and a hardware interrupt will be generated if the storage byte has the wrong parity bit. In response to such an interrupt, the microprocessor stops what it is doing and invokes an interrupt handling routine from the ROM component to reload all the software routines on to the RAM; it then resumes execution for the last input, which has been saved in a register of the microprocessor itself.

In the above fault-tolerant design, error detection is provided in hardware (parity bits); damage confinement is achieved by stopping execution and dumping any intermediate results; error recovery is done by rolling back the system and reloading all program code; fault treatment is unnecessary, assuming that the fault is transient. Furthermore, we assume that the possibility of having two or more consecutive transient faults in one processing cycle is negligible, i.e. at most one parity interrupt can occur after each input.

$$
\begin{array}{rcl}
Prog' & \widehat{=} & P'^{*} \\
P' & \widehat{=} & in?x : Float \rightarrow \Big(P(x) \mathbin{\triangle_{\circledR}} P(x) \Big) \\
P(x) & \widehat{=} & \left(\begin{array}{l} \text{if } x < 0 \text{ then} \\ \quad sub.in!(-x) \rightarrow sub.out?y : Float \rightarrow out!y \rightarrow Skip \\ \text{else} \\ \quad sub.in!x \rightarrow sub.out?y : Float \rightarrow out!y \rightarrow Skip \\ \text{fi} \end{array} \right) \mathbin{//} sub : SQRT \\
SQRT & \widehat{=} & in?z : Float \rightarrow out!\texttt{sqrt}(z) \rightarrow Skip
\end{array}
$$

Figure 3: A fault-tolerant program design expressed in CSP

Figure 3 shows our fault-tolerant program design *Prog′* in CSP. *Prog′* differs from *Prog* in that the main processing part of the design, $P(x)$, is now subject to the interrupt of the ® event, after which it will immediately return to its initial state in which x has the last input value. In the absence of an interrupt, "$P(x) \mathbin{\triangle_{\circledR}} P(x)$" simply behaves as $P(x)$ as in *Prog* before.

The fault tolerance capability of our program design is meant to be perfect, i.e. it ensures the "proper" behaviour of the filter as prescribed by *Spec* in spite of transient faults in the RAM component. To show that the fault-tolerant design *Prog′* is consistent with *Spec*, we can regard the parity interrupt as an internal event (similar to the subroutine call and return operations) and show that the observable behaviour of *Prog′* is consistent with *Spec*. Formally, this is done by

proving the following lemma:

$$Spec = Prog' \setminus \{®\}$$

where the interrupt event ® is explicitly designated by the hiding operator "\" as an internal event that takes place without any intervention from the environment.

4.1 A time-sensitive "bug"

Unfortunately, the fault-tolerant program design *Prog′* contains a "bug": the interruptible subprocess, $P(x)$, could be interrupted even after it has successfully computed and output a result, but just before it terminates, resulting in duplicate outputs for a single input. Formal verification, in the way that has been shown in Figure 2 and with the help of a model checking tool (as we shall discuss later), can help detect such highly time-sensitive bugs[1], which can easily escape testing.

$$
\begin{array}{rcl}
FTD & \widehat{=} & \left(OUT \underset{\{|outbuff, goahead, done|\}}{\|} FTP^* \right) \setminus \{|outbuff, goahead, done, ®|\} \\
OUT & \widehat{=} & outbuff?y : Float \rightarrow ((goahead \rightarrow out!y \rightarrow done \rightarrow OUT) \,\Box\, OUT) \\
FTP & \widehat{=} & in?x : Float \rightarrow \Big((FTP(x) \,\triangle_{®}\, FTP(x));\ goahead \rightarrow done \rightarrow Skip \Big) \\
FTP(x) & \widehat{=} & \left(\begin{array}{l} \text{if } x < 0 \text{ then} \\ \quad sub.in!(-x) \rightarrow sub.out?y : Float \rightarrow outbuff!y \rightarrow Skip \\ \text{else} \\ \quad sub.in!x \rightarrow sub.out?y : Float \rightarrow outbuff!y \rightarrow Skip \\ \text{fi} \end{array} \right) // sub : SQRT \\
SQRT & \widehat{=} & in?z : Float \rightarrow out!\texttt{sqrt}(z) \rightarrow Skip
\end{array}
$$

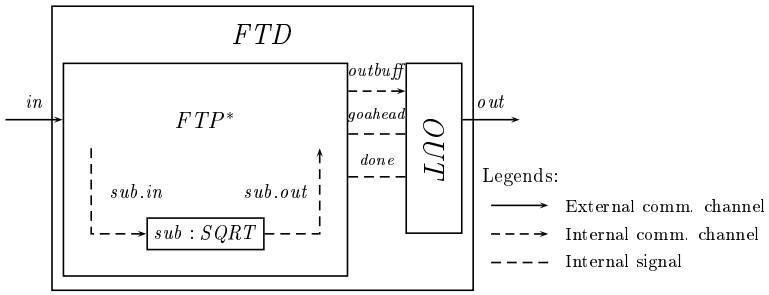

Figure 4: A fault-tolerant program design satisfying *Spec*

An obvious solution is to disable parity interrupts just before the output operation. But what if the RAM component does suffer from a glitch, with the parity interrupt disabled *and* before the output? Our solution involves an output buffer, typically found in a programmable I/O controller, which is independent from the RAM component as far as the parity interrupt is concerned. The

[1]This bug was indeed uncovered by the authors in verifying the design against its specification in CSP with the help of the FDR model checking tool.

microprocessor is now programmed to output its results only to the output buffer; each time after a result is sent to the buffer, the parity interrupt is turned off and a *goahead* signal is then sent to the I/O controller which will then actually output the result to the environment. When the I/O controller finishes outputting the result, it issues a *done* signal to notify the microprocessor. The parity interrupt is enabled again every time after an input from the environment. Note that the above arrangement actually allows the output buffer to be overwritten with duplicate results but guarantees that only the last one is output.

Figure 4 shows the revised design with a new name *FTD*, which can be verified as consistent with the specification *Spec* . Notice that the output buffer under the control of the I/O controller is modelled by process OUT. Following every input, OUT is ready to accept either the *goahead* signal or a fresh input. The choice is expressed using the external choice operator "$\Box$" in the (recursive) definition of OUT.

The output buffer synchronises with the microprocessor on communications along channel *outbuff* as well as the *goahead* and *done* signals under the arrangement of the parallel composition "$\underset{\{|outbuff, goahead, done|\}}{\|}$". Furthermore, writing to the output buffer through its *outbuff* channel and issuing the *goahead* and *done* signals together with the interrupt event are explicitly designated as internal operations by the hiding operator "\". Also note that although the design involves an output buffer, the strict alternation of input and output remains unchanged due to the "*goahead*/*done*" handshake between the microprocessor and the I/O controller.

4.2 Remarks on code implementation

The program design now involves interrupt handling and a minimal form of checkpointing, which are often implemented as operating system services. Abstraction for interrupt handling is also provided in high-level programming languages (e.g. Ada) in the form of exception handling [4].

5 A recovery block design

So far, we have assumed that the library routine `sqrt` computes the $\surd$ function correctly. "Bugs" in the code of this routine are regarded as software design faults. There are various fault tolerance approaches to handling software design faults. One of them is the recovery block approach [21]

which is based on the idea of design diversity [12].

A recovery block contains two or more software modules developed independently according to the same functional specification. An assumption of this approach is that these modules are independent from each other in the causes of their failures.

The mechanism of a recovery block works as follows. Upon entry to a recovery block, a checkpoint of the system state is first established. After that, the software modules take turn to compute a result which has to pass an acceptance test. The first module to run is called the primary module. If the primary module's result passes the acceptance test, the recovery block terminates; otherwise, the checkpoint is restored and a second module, called an alternate module, will compute another result. If the second module's result passes the acceptance test, the recovery block terminates; otherwise, the checkpoint is restored again and a third module (also called an alternate module) will try again and so on. If the primary module and all alternate modules fail to compute an acceptable result, an exception will be raised and either an (external) exception handler will take over or the exception is propagated to a higher system level for recovery.

Ideally, the acceptance test will accept only results that satisfy the functional specification. In practice, the design of the acceptance test must be kept simple to avoid any design faults in itself, and its complexity must be low in order to minimise run-time overheads.

$$RBS \;\widehat{=}\; in?x : Float \rightarrow \left(\left(\bigsqcap_{atsqrt(x,y)} out!y \rightarrow RBS \right) \sqcap (error \rightarrow RBS) \right)$$

where

$atsqrt(x, y)$ is implemented by the following (pseduo-)instructions:

1. store $y * y$ in $r0$;
2. store $abs(x)$ in $r1$;
3. XOR (exclusive-or) the first ten mantissa bits of $r0$ and $r1$;
4. if the result is "0000000000" and the exponent bits are the same in $r0$ and $r1$
   then $atsqrt(x, y) = true$
   otherwise $atsqrt(x, y) = false$;

Figure 5: Specification for the recovery block design

Unlike the hardware fault tolerant design discussed in the preceding section, the recovery block design is not meant to be "perfect"—it does not assume or guarantee that an acceptable result will always be produced by some software module each time. Instead, it assumes that the software modules could all fail at the same time, in which case an exception will be raised. Since these

assumptions or limitations of the recovery block design affect the external/observable behaviour of the filter, they need to reflect on the specification. Figure 5 shows the revised specification for the recovery block filter design.

Note that we have used the nondeterministic choice operator ("$\sqcap$") in specifying two possible responses to every input; both responses are observable and the choice between them is nondeterministic from the specification's point of view: the specification only demands that an input is followed by either an acceptable output or an exception, but not both; it is the designer's responsibility to ensure that an exception is only raised as the last resort.

On the other hand, using the indexed version of the nondeterministic choice operator, the output result is now specified to be within a range of acceptable values, corresponding to the acceptance test, *atsqrt*, of the recovery block design. The acceptance test is designed to take advantage of the IEEE floating point representation: it checks whether the square of the result matches the absolute value of the input for the first eleven significant *binary* digits[2] .

Figures 6 and 7 show the new filter design obtained by incorporating a recovery block with two software modules, `sqrt1` and `sqrt2`, into the previous fault-tolerant design. The new design also inherits the same fault tolerance capability associated with the parity interrupt. Again it is the designer's responsibility to verify that the new design is consistent with its specification and does not suffer from any time-sensitive "bugs" as we have seen. The verification can follow the same approach as shown in Figure 2 in which a proof is constructed by hand. Such a proof would, however, be far more elaborate and would be as error-prone as the design itself. Automated verification support becomes a necessity and the FDR model checking tool [5] provides such necessary support for CSP, as discussed in Section 6 below.

5.1 Remarks on code implementation

The code implementation of the recovery block design involves checkpointing and state-restoration. This can be achieved with a set of pre-defined classes in C++ as discussed in [31] and the coding has to follow certain conventions in making explicit calls for the checkpointing and state-restoration operations.

[2]The first eleven significant binary digits of a floating point IEEE floating point representation are actually represented by the first ten mantissa bits—the leading significant bit is always an implicit "1".

$$
\begin{array}{rcl}
RBD & \widehat{=} & \left(RBOUT \underset{A'}{\parallel} RBP^* \right) \setminus A' \\
 & & \text{where } A' = \{| outbuff, errbuff, goahead, done, ® |\} \\
RBOUT & \widehat{=} & \left(\begin{array}{l} outbuff?y : Float \rightarrow \\ \quad ((goahead \rightarrow out!y \rightarrow done \rightarrow RBOUT) \,\Box\, RBOUT) \\ \Box \\ errbuff \rightarrow ((goahead \rightarrow error \rightarrow done \rightarrow RBOUT) \,\Box\, RBOUT) \end{array} \right) \\
RBP & \widehat{=} & in?x : Float \rightarrow \left(\left(RBP(x) \;\triangle_{®}\; RBP(x) \right) ;\; goahead \rightarrow done \rightarrow Skip \right) \\
RBP(x) & \widehat{=} & \left(\begin{array}{l} \text{if } x < 0 \text{ then} \\ \quad rb.in!(-x) \rightarrow \\ \qquad \left(\begin{array}{l} rb.out?y : Float \rightarrow outbuff!y \rightarrow Skip \\ \Box \\ rb.error \rightarrow errbuff \rightarrow Skip \end{array} \right) \\ \text{else} \\ \quad rb.in!x \rightarrow \\ \qquad \left(\begin{array}{l} rb.out?y : Float \rightarrow outbuff!y \rightarrow Skip \\ \Box \\ rb.error \rightarrow errbuff \rightarrow Skip \end{array} \right) \\ \text{fi} \end{array} \right) //rb : RBSQRT \\
RBSQRT & \widehat{=} & (in?x : Float \rightarrow DSQRT(x)) //m : MSQRT \\
DSQRT(x) & \widehat{=} & \left(\begin{array}{l} \left(\begin{array}{l} m.next.in!x \rightarrow m.next.out?y : Float \rightarrow \\ \text{if } atsqrt(x,y) \text{ then} \\ \quad out!y \rightarrow m.ok \rightarrow Skip \\ \text{else} \\ \quad DSQRT(x) \\ \text{fi} \end{array} \right) \\ \Box \\ \left(\begin{array}{l} m.last.in!x \rightarrow m.last.out?y : Float \rightarrow \\ \text{if } atsqrt(x,y) \text{ then} \\ \quad out!y \rightarrow Skip \\ \text{else} \\ \quad error \rightarrow Skip \\ \text{fi} \end{array} \right) \end{array} \right) \\
MSQRT & \widehat{=} & \left(\begin{array}{l} next.in?z : Float \rightarrow next.out!\texttt{sqrt1}(z) \rightarrow \\ \left(\begin{array}{l} ok \rightarrow Skip \\ \Box \\ last.in?z : Float \rightarrow last.out!\texttt{sqrt2}(z) \rightarrow Skip \end{array} \right) \end{array} \right)
\end{array}
$$

Figure 6: A fault-tolerant design incorporating a recovery block

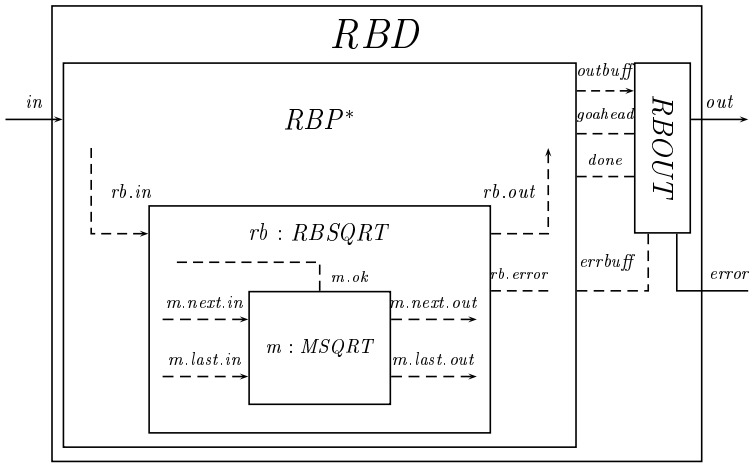

Figure 7: Hierarchical structure of the recovery block design

Recovery blocks were also incorporated into the programming language Pascal on an experimental basis [28].

$$
\begin{array}{rcl}
AbsRBD & \widehat{=} & \left(AbsRBOUT \underset{A'}{\parallel} AbsRBP^{*} \right) \setminus A' \\
 & & \text{where } A' = \{outbuff, errbuff, goahead, done, ®\} \\
AbsRBOUT & \widehat{=} & \left(\begin{array}{l} outbuff \rightarrow \\ \quad ((goahead \rightarrow out \rightarrow done \rightarrow AbsRBOUT) \,\Box\, AbsRBOUT) \\ \Box \\ errbuff \rightarrow \\ \quad ((goahead \rightarrow error \rightarrow done \rightarrow AbsRBOUT) \,\Box\, AbsRBOUT) \end{array} \right) \\
AbsRBP & \widehat{=} & in \rightarrow \left(\left(AbsRBP' \,\triangle_{®}\, AbsRBP' \right) ;\; goahead \rightarrow done \rightarrow Skip \right) \\
AbsRBP' & \widehat{=} & rb.in \rightarrow \left(\begin{array}{l} rb.out \rightarrow outbuff \rightarrow Skip \\ \Box \\ rb.error \rightarrow errbuff \rightarrow Skip \end{array} \right) //rb : AbsRBSQRT \\
AbsRBSQRT & \widehat{=} & (in \rightarrow AbsDSQRT)//m : AbsMSQRT \\
AbsDSQRT(x) & \widehat{=} & \left(\begin{array}{l} \left(\begin{array}{l} m.next.in \rightarrow m.next.out \rightarrow \\ \quad (out \rightarrow m.ok \rightarrow Skip \sqcap AbsDSQRT) \end{array} \right) \\ \Box \\ \left(\begin{array}{l} m.last.in \rightarrow m.last.out \rightarrow \\ \quad (out \rightarrow Skip \sqcap error \rightarrow Skip) \end{array} \right) \end{array} \right) \\
AbsMSQRT & \widehat{=} & \left(next.in \rightarrow next.out \rightarrow \left(\begin{array}{l} ok \rightarrow Skip \\ \Box \\ last.in \rightarrow last.out \rightarrow Skip \end{array} \right) \right)
\end{array}
$$

Figure 8: A data-independent abstract version of the recovery block design

6 Model checking with FDR2

Model checking involves exploring the entire state space of a process and hence only works for processes with finite numbers of states in general [3] and there are practical limits on the size and complexity of a system to be checked. Model checking has been implemented with symbolic algorithms that can handle systems with more than 10^{20} states [16]. The early versions of FDR explicitly and fully expand the state-space of a process and "*can deal with processes with approximately* 10^7 *states in about 4 hours on a typical workstation*" [24]. The new version, FDR2, incorporates "*the ability to build up a system gradually, at each stage compressing the subsystems to produce an equivalent process with (hopefully) many fewer states*" [5]. This allows FDR2 to check exponentially larger systems, such as a network of 10^{20} dining philosophers [24].

The data type *Float* (assuming an IEEE 32-bit single precision floating-point number representation), though finite, presents a major impact on the number of states of our designs. Furthermore,

floating point numbers are not supported by FDR2 at the time of writing (July, 2004) and hence we cannot apply FDR2 directly to the full design as shown in Figure 6. Nevertheless, FDR2 can play a useful role in verifying the *data-independent* aspect of the design [14].

From an abstract point of view, we can see quite clearly that the specification RBS (Figure 5) requires that, regardless of the values of data involved, each input operation is to be followed by either an output operation or an exception but not both. This abstract behavioural requirement is even more evident in the following abstract version of RBS:

$$AbsRBS \mathrel{\widehat{=}} in \rightarrow ((out \rightarrow AbsRBS) \sqcap (error \rightarrow AbsRBS))$$

where *in* and *out* stand for the input and output, respectively, of any data. The role of FDR2 lies in verifying that our filter design does, *at the data-independent level of abstraction*, conform to the above abstract specification. This helps safeguard the design against the kind of time-sensitive "bugs" that we have discussed earlier, regardless of any design faults in the library routines.

Figure 8 shows a data-independent abstract version of the recovery block design. Notice that not only the input variables and output expressions, but also the functional expressions associated with the two software modules (`sqrt1` and `sqrt2`) have been removed in the abstract design. In addition, the boolean (if-then-else) choice based on the acceptance test (*atsqrt*) has become a nondeterministic choice. Compared with the original design in Figure 6, the abstract version has a much smaller state-space that can be readily handled by FDR2.

Verifying the abstract design against the abstract specification, both expressed as CSP processes, involves using FDR2 to explore automatically the state-spaces of both processes and determine whether the former process is a *refinement* of the latter process according to the semantics of CSP. The standard semantic model of CSP is called the failures-divergence model [2] in which the possible behaviours of a process are denoted by a set called its *failures-divergence*. A process Q is a refinement of another process P, written $P \sqsubseteq Q$, if (and only if) the failures-divergence set of the former is contained in the failures-divergence set of the latter.

While a proper introduction to the semantics of CSP is beyond the scope of this paper, it is useful to mention one useful property about with the refinement operator $\sqsubseteq$ here: if two processes P and Q refine each other, i.e. $Q \sqsubseteq P$ and $P \sqsubseteq Q$, they are observably equivalent and interchangeable, i.e. $P = Q$. In fact, using FDR2, we are able to show that $AbsRBS$ and $AbsRBD$ are equivalent.

Figure 9 shows a screen shot of using FDR2 for the verification.

For further information about FDR2, data-independence, and the semantics of CSP, the reader is referred to [23].

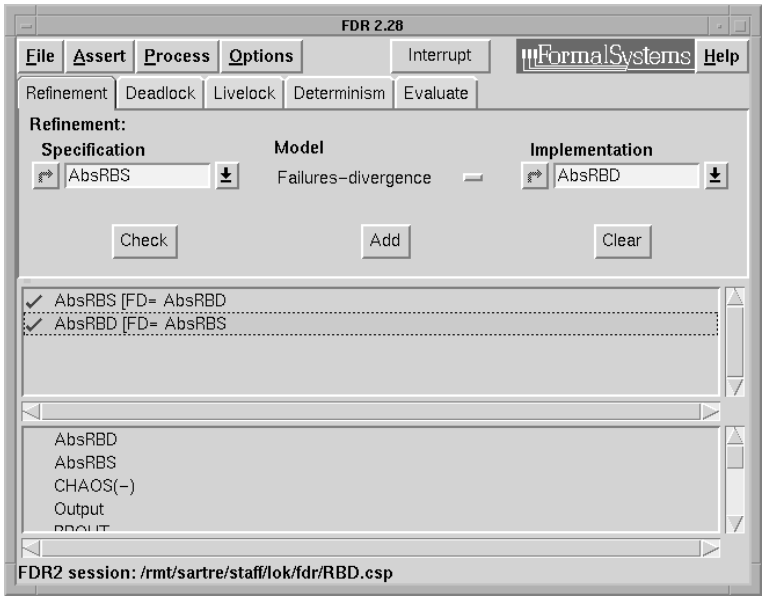

Figure 9: A screen shot of using the FDR2 model checking tool

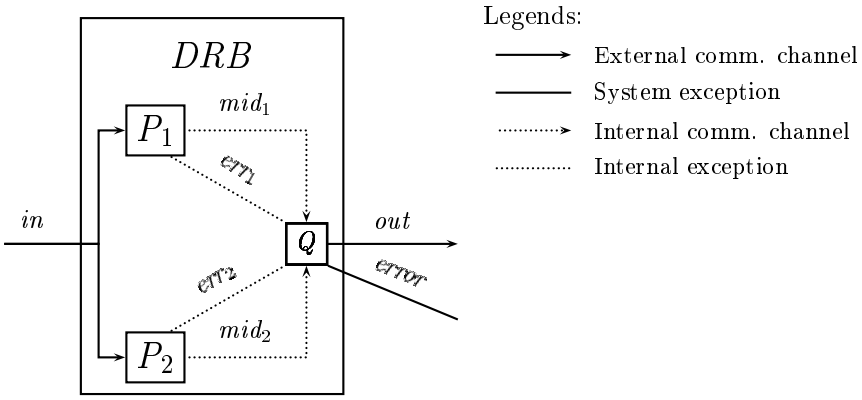

Figure 10: Structure of the fault-tolerant filter design

7 A distributed recovery block design

Figure 10 shows the structure of a dual-processor design for the DSP filter. It involves two processors, P_1 and P_2, running in parallel. The design of P_1 and P_2 follows the recovery block design discussed in Section 4. Every input signal value is fed into both P_1 and P_2. The custom-designed circuitry represented by Q is responsible for relaying the first available result from either P_1 or

P_2 to the output channel. Since only one result is to be output for each input signal, Q simply discards the second available result.

With a dual-processor configuration, the filter can be tolerant to not only software design faults (in the library routines) and parity-related transient RAM component faults, but also other transient hardware faults that prevent one of the two processors from producing an acceptable result. Only when both processors fail to produce an acceptable result and raise the internal exceptions (err_1 and err_2) in the same cycle will Q generate the system failure exception *error*. The above design technique is called the distributed recovery block (DRB) scheme [13].

Despite the redundant hardware, the DRB design is not meant to be perfect—it does not assume or guarantee that an acceptable result is always produced. In fact, it is meant to give the same observable behaviour as the recovery block design except that the two processors may begin a new processing cycle with an input operation ahead of the output operation (or exception) in the previous cycle. This amounts to giving the filter a buffer of one input. To account for the additional buffering, we may revise the specification for the filter as follows:

$$\begin{aligned} DRBS &\mathrel{\widehat{=}} \left(BUFF \underset{\{\!|b|\!\}}{\|} RBS[\![b/in]\!] \right) \setminus \{\!|b|\!\} \\ BUFF &\mathrel{\widehat{=}} in?x : Float \rightarrow b!x \rightarrow BUFF \end{aligned}$$

where "$[\![b/in]\!]$" renames the input channel of RBS as defined in Figure 5 to b. $BUFF$ models a simple one-place buffer for the filter inputs. The parallel composition "$\underset{\{\!|b|\!\}}{\|}$" specifies the concurrent operation of the buffer and the renamed process as well as their synchronised communication on channel b. Since the specification is only concerned with the observable behaviour of the filter in terms of its external communications, channel b is designated as internal.

The three components of the DRB design can each be specified by a CSP process. The behaviour of processor P_i (for $i = 1, 2$) is specified by the following process:

$$P_i \mathrel{\widehat{=}} RBS[\![mid_i, err_i/out, error]\!] \tag{1}$$

which is essentially the same as the specification for the recovery block design RBS (see Figure 5) except that the output channel (*out*) and the exception (*error*) have been renamed to mid_i and err_i, respectively. The behaviour of the inter-processor circuitry Q is specified by the following process:

$$
Q \;\widehat{=}\; \left(\begin{array}{l}
mid_1?y : Float \rightarrow out!y \rightarrow \left(\begin{array}{l} mid_2?z : Float \rightarrow Q \\ \Box \\ err_2 \rightarrow Q \end{array}\right) \\
\Box \\
mid_2?y : Float \rightarrow out!y \rightarrow \left(\begin{array}{l} mid_1?z : Float \rightarrow Q \\ \Box \\ err_1 \rightarrow Q \end{array}\right) \\
\Box \\
err_1 \rightarrow \left(\begin{array}{l} mid_2?z : Float \rightarrow out!y \rightarrow Q \\ \Box \\ err_2 \rightarrow error \rightarrow Q \end{array}\right) \\
\Box \\
err_2 \rightarrow \left(\begin{array}{l} mid_1?z : Float \rightarrow out!y \rightarrow Q \\ \Box \\ err_1 \rightarrow error \rightarrow Q \end{array}\right)
\end{array}\right)
$$

The combined behaviour of the components is given by:

$$
DRB \;\widehat{=}\; \left(\left(P_1 \underset{\{\!|in|\!\}}{\|} P_2\right) \underset{A}{\|} Q\right) \setminus A \tag{2}
$$

where $A = \{\!|mid_i, err_i \mid i \in \{1,2\}|\!\}$. The parallel composition "$\underset{\{\!|in|\!\}}{\|}$" specifies that the two processors, P_1 and P_2, synchronise with each other on receiving inputs from the *in* channel, whereas the inter-processor circuitry Q synchronises with the two processors on communications along the mid_i channels and on the err_i exceptions (collectively defined as A). These inter-processor communications and exceptions are designated as internal operations (see Figure 10).

With the help of the parallel composition ("||") and other CSP operators, we have constructed a distributed recovery block design by reusing a verified design component. The verification of the more complex design is simplified by the composition properties of the CSP algebra: once we have verified the top-level design as correct, i.e. $DRBS \sqsubseteq DRB$, we can elaborate it by replacing its component specifications (P_i) with their detailed designs without having to verify it further. Formally, given $RBS \sqsubseteq RBD$ from Section 4, and with (2) and (1) above, we have:

$$
DRBS \sqsubseteq \left(\left(RBD[\![mid_1, err_1/out, error]\!] \underset{\{\!|in|\!\}}{\|} RBD[\![mid_2, err_2/out, error]\!]\right) \underset{A}{\|} Q\right) \setminus A
$$

where $A = \{\!|mid_i, err_i \mid i \in \{1,2\}|\!\}$.

The actual verification of the top-level design can be carried by algebraic manipulation as shown in Figure 2 or by model checking, at the data-independent abstract level, with the help of the FDR2 tool as discussed in the preceding section.

8 Timing

Finally, let us consider the case when one of the two processors in the distributed recovery block design breaks down as a result of some permanent hardware fault (e.g. circuit breakdown). We need to modify the design of the inter-processor circuitry Q because we have previously assumed that, for each input, each processor always produces something, either a result or an exception. Q would simply wait forever for that something from a broken processor. The problem can be solved by incorporating a timer device into the inter-processor circuitry and setting up appropriate "timeout" interrupts. This solution, however, has a serious implication to our system design: the synchronisation of components will rely on not only their readiness to communicate, but also the passage of time. Since only the ordering of events but not their precise timing is modelled in CSP, we can neither describe nor reason about the new system design adequately. This calls for the use of a design notation that supports the expression and reasoning of passage of time, in addition to what we have been doing so far with CSP.

Fortunately, there is a timed version of CSP [27] with syntax and semantics compatible with those of (the untimed version of) CSP. In fact, all the CSP operators and constructs that we have seen in this article are also available in the timed version, although their semantics and associated verification methods are inevitably more intricate and involving information about the timing of events. The benefit is that we can model and reason about the behaviour of a system more precisely, taking into account the progress of time. For instance, the timeout value to be set for the timer device has to be determined based on the minimum and maximum amounts of time required for both processors to compute their results and deliver them to Q, as well as the patterns of interaction between the processors and Q. Given these minimum and maximum time values, the calculation of the timeout value can then be formally verified based on the system design with the help of proof rules associated with the Timed CSP notation.

Furthermore, we can always derive an abstract untimed CSP description of a timed CSP design. This amounts to "stripping" all timing information from the timed design. The consequence is that whenever a process is choosing between alternative courses of action depending on time (e.g. when timeout is enabled), the choice will become a nondeterministic one in the abstract untimed description. Formal verification supported by model checking in (untimed) CSP can then be applied

to such abstract untimed descriptions in the same way as we have discussed earlier.

9 Concluding remarks on larger systems and complexity issues

CSP has previously been applied to the design and verification of a fault-tolerant system [20] and a distributed recovery algorithm [9] but only hardware faults are considered in these previous studies. A more elaborate system design case study that deals with both hardware and software faults using CSP can be found in [32]. This paper complements [32] with an overview of the formal development process together with an emphasis on the role of model checking in the process. A related application domain for CSP and FDR2 is on the verification of security protocols for which more realistic sizable case studies can be found in [25] and the MAFTIA project [15].

The FDR2 model checking tool plays an instrumental role in scaling up the application of CSP to more complex systems, as pointed out in Section 6. On the other hand, the theory of data-independence [14] (also discussed in Section 6) provides a powerful abstraction mechanism for us to handle complex system behaviour that would otherwise defeat the most powerful model checkers. There are other formal methods that also emphasise model checking as a means of automated verification, e.g. CTL/SMV [3] and Promela/SPIN [11]. An account of these and some other related formal approaches to fault tolerance can be found in [29]. CSP and other process algebras such as LOTOS [1] distinguish themselves with their support for algebraic manipulation which can help "simplify" designs as illustrated in Figure 2.

System behaviour becomes even more complex when real-time is taken in account, as discussed in Section 8. While there is currently no model checkers for the timed version of CSP, the theoretical foundation for such model checkers has already be laid [18, 19] . Meanwhile, there is the theory of timewise refinement [26] that allows us to apply FDR2 to the verification of those "untimed" aspects of the system behaviour, thereby reducing the burden on the developer/verifier.

Finally, apart from the FDR2 model checking tool, other support tools for CSP include a type-checker and an animator (ProBE [6]) as well as the JCSP programming environment [30] that incorporates CSP into Java for (more) direct code implementation of CSP designs.

Acknowledgements

We would like to thank Iain Bate and the anonymous reviewers for their comments and suggestions. The first author would like to thank the staff and research students of the Formal Methods Group at Royal Holloway for their interesting and helpful discussion during his visit there when completing this work. The visit is supported by a study leave from Lingnan University, Hong Kong.

References

[1] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1988.

[2] S. D. Brookes and A. W. Roscoe. An improved failures model for CSP. In *Proceedings of the NSF-SERC Seminar on Concurrency*, volume 197 of *LNCS*. Springer, 1984.

[3] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[4] F. Cristian. Exception Handling and Software Fault Tolerance. *IEEE Transactions on Computers*, C-31(6):531–540, June 1982.

[5] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, 2003.

[6] Formal Systems Europe (Ltd.). *Process Behaviour Explorer: ProBE User Manual*, 2003.

[7] Tom R. Halfhill. The Truth Behind the Pentium Bug. *Byte*, March 1995.

[8] Anthony Hall and Roderick Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Software*, pages 18–25, January/February 2002.

[9] Jifeng He and C.A.R. Hoare. Algebraic specification and proof of a distributed recovery algorithm. *Distributed Computing*, 2:1–12, 1987.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[11] G. Holzmann. The model check SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[12] John P. J. Kelly, Thomas I. McVittie, and Wayne I. Yamamoto. Implementing Design Diversity to Achieve Fault Tolerance. *IEEE Software*, pages 61–71, July 1991.

[13] K.H. Kim and H. O. Welch. Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications. *IEEE Transactions on Computers*, 38(5):626–636, May 1989.

[14] R. S. Lazić and A. W. Roscoe. Data independence with generalised predicate symbols. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 319–325. CSREA Press, 1999.

[15] The MAFTIA Project. http://www.newcastle.research.ec.org/maftia/.

[16] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[17] Ministry of Defence, Directorate of Standardization, Glasgow, U.K. *The Procurement of Safety Critical Software in Defence Equipment, Interim Defence Standard 00-55*, April 1991.

[18] Joel Ouaknine. *Discrete Analysis of Continuous Behaviour in Real-Time Concurrent Systems*. PhD thesis, Oxford University, 2001.

[19] Joel Ouaknine. Digitisation and full abstraction for dense real-time model checking. In *Proceedings of TACAS 2002*, volume 2280 of *LNCS*. Springer, 2002.

[20] Jan Peleska. Design and verification of fault tolerant systems with CSP. *Distributed Computing*, 5:95–106, 1991.

[21] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[22] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

[23] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[24] A. W. Roscoe et al. Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 1995.

[25] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *Modelling and Analysis of Security Protocols.* Addison Wesley, 2001.

[26] S.A. Schneider. Timewise refinement for communicating processes. *Science of Computer Programming*, 28:43–90, 1997.

[27] S.A. Schneider. *Concurrent and Real-time Systems: The CSP Approach.* John Wiley, 2000.

[28] S. K. Shrivastava. Sequential Pascal with recovery blocks. *Software—Practice and Experience*, 8(2):177–185, March/April 1978.

[29] J. Vytopil. *Formal Techniques in Real Time and Fault Tolerant Systems.* Kluwer, 1993.

[30] P. H. Welch and P. D. Austin. The JCSP Home Page. http://www.cs.ukc.ac.uk/projects/ofa/jcsp/.

[31] J. Xu, B. Randell, C. Rubira-Calsavara, and R. J. Stroud. Towards an object-oriented approach to software fault tolerance. In *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 275–290, Newcastle-upon-Tyne, 1994.

[32] W. L. Yeung and S. A. Schneider. Design and Verification of Distributed Recovery Blocks with CSP. *Formal Methods in System Design*, 22(3):225–248, May 2003.

Biography

W. L. Yeung completed his Ph.D. in the UK and has taught in universities in both the UK and Hong Kong. His research interests include the application of formal methods to fault-tolerant systems and also the integration of formal and semi-formal methods. He is a member of the IEEE Computer Society, the ACM, and the BCS.

Contact information:
Department of Information Systems
Lingnan University
Tuen Mun, Hong Kong
Tel +852 2616 8095
Fax +852 2892 2442
Email wlyeung@ln.edu.hk

Steve Schneider is a Professor of Computer Science at Royal Holloway, University of London, where he leads the Formal Methods Group. His research interests include process algebra, real-time systems, and integrating formal methods, as well as formal methods for security protocols and non-interference. He is a member of the IEEE Computer Society.

Contact information:
Department of Computer Science
Royal Holloway, University of London
Egham, Surrey TW20 0EX
United Kingdom
Tel +44 1784 443431
Fax +44 1784 439786
Email S.Schneider@cs.rhul.ac.uk