, , 1–25 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Design and Verification of Distributed Recovery Blocks with CSP

W.L. YEUNG wlyeung@ln.edu.hk
Lingnan University, Hong Kong

S.A. SCHNEIDER steve@dcs.rhbnc.ac.uk
Royal Holloway, University of London, Egham, Surrey, TW20 0EX, United Kingdom

Editor:

Abstract. A case study on the application of Communicating Sequential Processes (CSP) to the design and verification of fault-tolerant real-time systems is presented. The distributed recovery block (DRB) scheme is a design technique for the uniform treatment of hardware and software faults in real-time systems. Through a simple fault-tolerant real-time system design using the DRB scheme, the case study illustrates a paradigm for specifying fault-tolerant software and demonstrates how the different behavioural aspects of a fault-tolerant real-time system design can be separately and systematically specified, formulated, and verified using an integrated set of formal techniques based on CSP.

Keywords: Real-time systems; fault-tolerance; distributed recovery block scheme; CSP; formal specification and verification; timewise refinement

1. Introduction

Formal methods based on state-machines [36], assertional proof systems [5, 6, 20], process algebras [25, 2, 15, 31, 38], and other formalisms [17, 3, 8, 27] have been applied to the development of fault-tolerant real-time systems. Many case studies (e.g. [35, 28, 14, 34, 21, 4]) can be found in the literature but none of them treats system design involving the tolerance of both hardware and software faults, while addressing real-time requirements simultaneously.

As the reliability of computing hardware continues to increase, error-prone software remain ubiquitous, even in safety-critical applications. Formal methods aid in producing software designs that are correct and reliable. Should a design technique helps to produce software that can tolerate its own design faults, it is important that formal methods provide the necessary framework for that design technique to be applied as rigourously as possible.

The formalism of Communicating Sequential Processes, widely known by its acronym CSP[2, 15, 31, 38], has been applied to concurrent, distributed, and real-time systems that tolerate hardware faults [14, 28, 34, 33, 39]. This paper presents a case study on the application of CSP to the design and verification of fault-tolerant real-time system designs that handle both hardware and software faults. The design technique used here is the distributed recovery block (DRB) scheme [18], which was proposed for the uniform treatment of hardware and software faults in real-time computer systems. While the rationale behind the DRB scheme is clearly

discussed and illustrated in the original proposal, the design examples are described in an informal manner without using any formal notation and their behaviour is not subject to detailed consideration.

The present case study illustrates a paradigm for specifying fault-tolerant software and demonstrates how the different behavioural aspects of a fault-tolerant real-time system design can be separately and systematically specified, formulated, and verified using an integrated set of formal techniques based on CSP. Specifically, an untimed version of CSP is used to specify the behaviour together with some essential safety and liveness properties of the design. A fault-tolerant software design is then formulated and verified using a variety of techniques including the use of a model checking tool. Real-time requirements are then addressed using a timed version of CSP. The two versions of CSP are linked together by the theory of timewise refinement [38, 39].

The paper is organised as follows: section 2 briefly introduces the language of CSP; sections 3 and 4 consider the specification for a simple computing system which is designed to tolerate faults in its software components; section 5 presents the design in its basic form without assuming any hardware faults; section 6 describes the verification of the design against its specification; section 7 considers the real-time dimension and refines the basic design *timewise* with reference to the theory of timewise refinement [38, 39]; section 8 makes some explicit assumptions about hardware faults and "adjusts" the basic design for the fault assumptions; section 9 gives a conclusion. For reasons of space, this paper contains only selected proofs; [40] is a full version of this paper, containing all the proofs.

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 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. In addition, when time is taken in account explicitly, we have the following assumptions:

- Maximal progress: A synchronisation event occurs as soon as all participants are ready to perform it.

- Maximal parallelism: Every process has a dedicated processor; processes do not compete for processor time. It is a reasonable assumption in the context of distributed recovery blocks where independent processes do have their own processors. In general, time-sliced processing can be modelled explicitly if required.

- Finite variability: No process may perform infinitely many events, or undergo infinitely many state changes, in a finite interval of time.
- Instantaneous events: Events have zero duration. Actions with non-zero duration can also be modelled if required.
- Real-time: The time domain is taken to be the non-negative real numbers. Since the reals are dense, our maximal parallelism assumption above means that there is no positive lower bound on the time difference between two independent events occurring at different times.
- Newtonian time: Time progresses in all processes at the same rate, and all with respect to the same unique global time frame.

The following paragraphs briefly introduced the CSP operators used in this paper. A comprehensive description of the language is found in [15, 33, 39]. The language of CSP used in this paper is defined by the following pseudo Backus-Naur form definitions:

$$\begin{aligned} P \ ::= \ & \mathit{Stop} \mid a \rightarrow P \mid P \,\Box\, P \mid P \sqcap P \mid a : A \rightarrow P_a \mid \\ & P \underset{A}{\parallel} P \mid P \setminus A \mid X \mid \mu X \bullet P \mid \\ & a \xrightarrow{t} P \mid P \rhd \{t\}\, P \end{aligned}$$

Σ is the set of all possible events, a is in Σ, A in $\mathbb{P}(\Sigma)$, t in $[0, \infty)$, and X is a process variable. CSP processes are terms with no free process variables (i.e. every process variable is bound by some μ 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. By default, there is no delay between the occurrence of a and the beginning of 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 $a : A \rightarrow P_a$ 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:

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

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 $out!x \rightarrow P$ and this is simply a shorthand for $out.x \rightarrow P$. 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. From the maximal progress assumption, the internal events will occur as soon as P is ready to perform them.

A recursive term $\mu X \bullet P$ behaves as P, with every occurrence of X in P representing an immediate recursive invocation. Thus we will have the usual law

$$\mu X \bullet P = P[\mu X \bullet P / X]$$

Note that the timed-semantics (see [39]) requires that every recursive term is *time-guarded* to ensure finite variability, i.e. there is some $t > 0$ of which any execution must take at least t unit(s) of time to reach a recursive invocation. (This requirement, however, does not apply to the untimed semantics of CSP). The delayed prefix $a \xrightarrow{t} P$ and the timeout operator $\triangleright\{t\}$ are two operators that can provide time-guarding. $a \xrightarrow{t} P$ is ready initially to engage in a; but once that event is performed, there is a delay of t before it behaves as P. The timeout choice $P \triangleright\{t\}\ Q$ initially behaves as process P. If an event is performed before time t, then the choice is resolved in favour of P, which continues to execute, and Q is discarded. If no such event is performed, then the timeout occurs at time t, and the subsequent behaviour is that of Q.

A hierarchy of semantic models have been defined for CSP [30]. For the purpose of the case study, only two of these models, namely, the failures model ($\mathcal{M}_{UF}$) and the timed failures model ($\mathcal{M}_{TI}$), are considered in this paper. Their formal definition can be found in [2, 37, 26, 39].

2.1. Specification

An untimed specification is a predicate on failures and is written in terms of a pair of variables (tr, ref) which range over all observable pairs of trace and refusal of the process being specified. A trace is a finite sequence of events from the set Σ whereas a refusal is a set of events from the set Σ. If the failure (tr, ref) is present in the semantic set of the system, then the system may perform trace tr and *then* refuse to engage in any event from ref. Process P meets specification $S(tr, ref)$, written P **sat** $S(tr, ref)$, if $S(tr, ref)$ holds for every observable pair of trace and refusal of P.

Similarly, a timed specification is a predicate on timed failures and is written in terms of a pair of variables $(s, \aleph)$ which range over all observable pairs of timed trace and timed refusal, respectively, of the process being specified. (See [39] for the definition of timed trace and timed failure).

Writing specifications directly as predicates upon timed failures can be cumbersome. Also, there are many similar specification patterns for safety, liveness and

commonly occurring assumptions about the environment of the process. It is convenient to define a number of specification macros or idioms [9] as a shorthand for these patterns and for use with proof rules to reason about specifications at the higher level of abstraction. The following nine macros are used in the present case study; their formal definition can be found in [39]:

$$\begin{array}{ccc} a \ \mathsf{at}\ t(s,\aleph), & a\ \mathsf{live}\ t(s,\aleph), & a\ \mathsf{live\ from}\ t_1\ \mathsf{until}\ t_2(s,\aleph), \\ a\ \mathsf{ref}\ t(s,\aleph), & a\ \mathsf{live\ from}\ t\ \mathsf{until}\ A(s,\aleph), & a\ \mathsf{open}\ t(s,\aleph), \\ A\ \mathsf{at}\ [t_1,t_2)(s,\aleph), & a\ \mathsf{live\ within}\ \delta\ \mathsf{of}\ t\ \mathsf{until}\ A(s,\aleph), & \mathsf{name\ of\ last}\ A\ \mathsf{before}\ t(s,\aleph). \end{array}$$

"a at t" is true for a particular execution if event a occurs at time t. "a ref t" is true if event a is refused at time t. For "A at $[t_1, t_2)$", a process performs any event in the set A during the interval $[t_1, t_2)$. For "a live t", a process is prepared to perform a at time t and, for "a live from t until A", it will remain so until disabled by some event from the set A. For "a live within δ of t until A", a process could be ready to perform a at any time starting from t but certainly not later than $t + \delta$; whereas, for "a live from t_1 until t_2", the process is prepared to perform a from time t_1 and will remain so until time t_2. "a open t" states that event a is open to a process at time t, i.e. the environment of the process is ready to see a performed at that time. Finally, "name of last A before t" is a function that reports the last event performed before time t for a particular execution.

2.2. Proof system

The compositional nature of the semantics of CSP, including both the timed and untimed versions, leads to proof systems that are also compositional, allowing a CSP design to be verified against its specification in a structured and top-down manner [10, 39]. The following proof rule for parallel combination of processes is a typical one that is being used in the present case study.

RULE 1 *Parallel combination*

$$\frac{\begin{array}{l} P\ \mathbf{sat}\ S(s,\aleph) \\ Q\ \mathbf{sat}\ T(s,\aleph) \end{array}}{\begin{array}{l} P \underset{A}{\parallel} Q\ \ \mathbf{sat}\ \ \exists\, s_P, s_Q, \aleph_P, \aleph_Q \bullet \\ \qquad\qquad S(s_P,\aleph_P) \wedge T(s_Q,\aleph_Q) \wedge \\ \qquad\qquad (s,\aleph) \in (s_P,\aleph_P) \underset{A}{\parallel} (s_Q,\aleph_Q) \end{array}}$$

This proof rule for the parallel operator $P \underset{A}{\parallel} Q$ relates the behaviour of the combined process with behaviours of P and Q. If we can prove that processes P and Q satisfy $S(s,\aleph)$ and $T(s,\aleph)$, respectively, then the above rule allows us to conclude that the combined behaviour of the two processes satisfies the predicate expressed in terms of S and T below the horizontal line (ie. $\exists\, s_P, s_Q, \aleph_P, \aleph_Q \bullet \ldots$). $(s_P,\aleph_P) \underset{A}{\parallel} (s_Q,\aleph_Q)$ is the set of all compound behaviours of $P \underset{A}{\parallel} Q$ that can arise

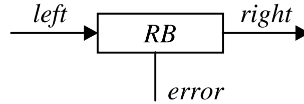

Figure 1. A sequential computing device

from those concurrent behaviour of P and Q. The Appendix gives four auxiliary rules (Rules 2—5) that help deduce information about s and $\aleph$ from the component specifications and *vice versa*.

3. Specifying recovery blocks

The diagram in Figure 1 represents a computing device, namely RB, connected to its environment via communications *left* and *right* plus a signal line *errror*. RB receives some items of input data via *left* and for each item, computes a function θ and outputs the result via channel *right*. We may specify the behaviour of RB by the following CSP process:

$$Spec \mathrel{\widehat{=}} left?x \rightarrow right!\theta(x) \rightarrow Spec$$

Spec is an abstract behavioural specification for RB in the sense that it only mentions the input and output data, how they should relate to each other, and how the input and output operations should alternate. Ideally, the actual device RB operates reliably and continuously as intended by *Spec*, processing every input into a desired output. (Note that even though RB is equipped with an error signal line, the specification *Spec* takes no account of any possible failures and does not mention *error* at all).

In reality, there are situations in which RB could fail due to:

Permanent hardware faults The hardware breaks down for physical reasons (e.g. curcuit burnt-out).

Transient hardware faults The hardware experiences intermittent “glitches”, affecting the results of computation occasionally.

Circuit design faults “Bugs” in today’s very large scale integrated circuit design can escape testing and go into production phase (e.g. see [12] about a “bug” in the circuitry of the floating point unit of Intel’s Pentium).

Software design faults The program that drives RB contains residual “bugs” and does not exactly compute θ.

The recovery block scheme was proposed [16, 29] for organising diverse designs in order to support fault tolerant software. Using this scheme, the design of RB

may contain two or more alternative software modules which are written independently according to the same functional specification (i.e. θ) . For each input, these modules take turn to compute a result which has to pass a suitable *acceptance test* before it can be output. If all software modules fail to compute an acceptable result, an *error* signal is raised. What happens after an error signal has been raised will depend on the application and its environment. Here are some possibilities:

1. Abort, let the environment handle the erroneous situation.

2. Output nothing but continue with the next input.

3. Output a result, though unacceptable, and continue with the next input.

We have assumed the last possibility (3) in our design. This might seem unreasonable at first sight but we can easily think of situations where the mere availability of a result, though unacceptable, is necessary for the system to keep functioning (e.g. avoiding a deadlock) while corrective actions are being taken. Such a design choice also renders our case study slightly more straightforward and readable (since, for every input, there must be an output). However, we must emphasize that the other possibilities are just as amenable to the formal approach.

Assuming that all communication channels are unbuffered (synchronised), and only two software modules (one primary and one alternative) are used, we can describe the behaviour of such a sequential computing device in CSP as follows:

$$SpecRB \;\widehat{=}\; left?x \rightarrow \left(\begin{array}{l} right!f(x) \rightarrow SpecRB \\ \not< T(x, f(x)) \not> \\ \left(\begin{array}{l} right!g(x) \rightarrow SpecRB \\ \not< T(x, g(x)) \not> \\ error \rightarrow right!g(x) \rightarrow SpecRB \end{array} \right) \end{array} \right)$$

f and g are computable functions representing the primary and alternative software modules, respectively. The operator $P \not< b \not> Q$ is the infix version of the boolean choice `if` b `then` P `else` Q. T is the acceptance test. Here, we assume that the same acceptance test is applied to the result of all software modules; it should be pointed out that some variants of the recovery block scheme allow for different acceptance tests for each software module. Note that, after an *error* signal has been raised, the second result, though unacceptable, is output anyway.

Comparing *SpecRB* with *Spec*, we can see that the former alternates between input and output like the latter but embodies the possibility of raising the *error* signal after an input. Also, *SpecRB* does not really specify the computation of the required function θ but rather incorporates a run-time acceptance test of the results of f and g, both of which may or may not correctly compute θ. *SpecRB* can be generalized to include n alternative software modules, $f_1, f_2, \ldots, f_n$, plus a primary module f_0, as follows:

$$
SpecRB_n \;\widehat{=}\; left?x \rightarrow \left(\begin{array}{l} right!f_0(x) \rightarrow SpecRB_n \\ \nprec T(x, f_0(x)) \nsucc \\ \left(\begin{array}{l} right!f_1(x) \rightarrow SpecRB_n \\ \nprec T(x, f_1(x)) \nsucc \\ \left(\begin{array}{l} \cdots \\ \cdots \left(\begin{array}{l} right!f_n(x) \rightarrow SpecRB_n \\ \nprec T(x, f_n(x)) \nsucc \\ error \rightarrow right!f_n(x) \rightarrow SpecRB_n \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right)
$$

Throughout the case study, we assume that each of the computable functions $f, g, f_0, f_1, \ldots, f_n$ always terminates with a result, although the result may not be correct. The possibility of hardware faults is considered later in section 8.

4. Specifying distributed recovery blocks

The distributed recovery block (DRB) scheme allows the uniform treatment of hardware faults and software design faults in real-time systems. It involves distributing the execution of the software modules of a typical recovery block design on multiple processors, thereby providing tolerance of certain permanent and transient hardware faults in addition to software design faults.

Suppose we reformulate the design of the computing device RB in Figure 1 using the DRB scheme with $n+1$ (one primary and n alternative) software modules. We may consider using $SpecRB_n$ as a specification but its purely sequential structure makes it too strong for the DRB design. The question is how we could weaken $SpecDB_n$ by not mentioning (restricting) too much about the internal structure of the DRB design. Here is one possible answer:

$$
SpecDRB_n \;\widehat{=}\; left?x \rightarrow \left(\begin{array}{l} \left(\bigsqcap_{0 \leqslant i \leqslant n} right!f_i(x) \rightarrow SpecDRB_n \right) \\ \sqcap \\ \left(error \rightarrow \bigsqcap_{0 \leqslant i \leqslant n} right!f_i(x) \rightarrow SpecDRB_n \right) \end{array} \right)
$$

$SpecDRB_n$ mentions nothing about the internal structure of system but simply specifies that for each input, the result of one of the f_i is output, possibly preceded by an *error* signal. Like $SpecDB_n$, $SpecDRB_n$ guarantees the alternation between input and output and allows the possibility of raising an error signal. However, it fails to address some important properties:

1. An output must be acceptable if it is not preceded by an *error* signal.

2. An output must be unacceptable if it is preceded by an *error* signal.

3. If at least one f_i produces an acceptable result, it must be an acceptable result that is output without an error signal.

The above properties can be expressed in terms of the trace and failure semantics as follows:

$$
\begin{aligned}
S_1(tr) \;\widehat{=}\;& \mathsf{channel}(foot(tr)) = right \wedge \\
& \quad T((tr \Downarrow left)@(\#(tr \Downarrow right) - 1), \mathsf{value}(foot(tr))) \\
& \Rightarrow \#(tr \upharpoonright error) = \\
& \quad \#\{i \mid 0 \leqslant i < \#(tr \Downarrow right) \wedge \neg\, T((tr \Downarrow left)@\, i, (tr \Downarrow right)@\, i)\} \\
S_2(tr) \;\widehat{=}\;& \#\{i \mid 0 \leqslant i < \#(tr \Downarrow right) \wedge \neg T((tr \Downarrow left)@i, (tr \Downarrow right)@i)\} \\
& \leqslant^1 \#(tr \upharpoonright error) \\
L(tr, ref) \;\widehat{=}\;& \mathsf{channel}(foot(tr)) = left \wedge \\
& \quad \exists\, i \bullet 0 \leqslant i \leqslant n \wedge T(\mathsf{value}(foot(tr)), f_i(\mathsf{value}(foot(tr)))) \\
& \Rightarrow \{right.f_i(\mathsf{value}(foot(tr))) \mid \\
& \qquad 0 \leqslant i \leqslant n \wedge T(\mathsf{value}(foot(tr)), f_i(\mathsf{value}(foot(tr))))\} \setminus \\
& \qquad ref \neq \emptyset \wedge \\
& \quad error \in ref
\end{aligned}
$$

where $\mathsf{channel}(c.v)$ and $\mathsf{value}(c.v)$ extract the channel c and value c, respectively, of the communication event $c.v$; $foot(tr)$ returns the last event of trace tr; $\#tr$ gives the length of the trace tr; $tr \Downarrow c$ returns the sequence of values passed along channel c; $tr@i$ returns the i-th event (starting from zeroth) of tr; $tr \upharpoonright A$ returns the subsequence of events of tr in A.

S_1 essentially says that whenever an acceptable result is output, the number of unacceptable results that have been output must be equal to the number of error signals raised. S_2 states that the number of error signals raised must be one greater than or equal to the number of unacceptable results output. S_1 and S_2 together ensure the properties (1) and (2). L guarantees that acceptable results have priority over the error signal (and unacceptable results). Note that S_1, S_2 are safety properties expressed in the trace semantics of CSP whereas L is an liveness property expressed in the failure semantics of CSP.

5. Design

Figure 2 shows the structure of a design using the distributed recovery block scheme. The software modules which compute $f_0, f_1, \ldots, f_n$ have been distributed to $n+1$ sequential processors, whose behaviours are described by processes $P_0, P_1, \ldots, P_n$, respectively. The acceptance test T is now carried out by another processor, described by R. R receives the results of computations by $f_0, f_1, \ldots, f_n$ via channels $p_0, p_1, \ldots, p_n$, respectively, and outputs final results on channel $right$. The design can be described in CSP as follows:

$$
\begin{aligned}
P_i \;&\widehat{=}\; left?x \rightarrow p_i!f_i(x) \rightarrow P_i \qquad \text{for } 0 \leqslant i \leqslant n \\
R \;&\widehat{=}\; left?x \rightarrow R_{x,\{i \mid 0 \leqslant i \leqslant n\}}
\end{aligned}
$$

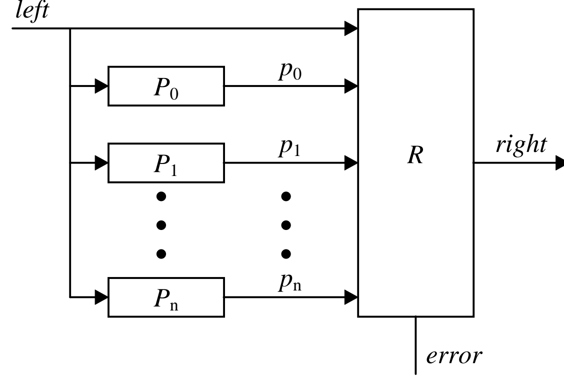

Figure 2. Distributed recovery block

$$
\begin{array}{rcl}
R_{x,\{i\}} & \widehat{=} & p_i?y \to \left(\begin{array}{l} \mathit{right}!y \to R \\ \not< T(x,y) \not> \\ \mathit{error} \to \mathit{right}!y \to R \end{array} \right) \quad \text{for } 0 \leqslant i \leqslant n \\
R_{x,A} & \widehat{=} & \Box_{i \in A}\, p_i?y \to \left(\begin{array}{l} \mathit{right}!y \to R'_{A\setminus\{i\}} \\ \not< T(x,y) \not> \\ R_{x,A\setminus\{i\}} \end{array} \right) \text{for } \begin{array}{l} A \subseteq \{i \mid 0 \leqslant i \leqslant n\} \\ \wedge\ \#A > 1 \end{array} \\
R'_{\emptyset} & \widehat{=} & R \\
R'_B & \widehat{=} & \Box_{j \in B}\, p_j?z \to R'_{B\setminus\{j\}} \quad \text{for } B \subset \{i \mid 0 \leqslant i \leqslant n\} \wedge B \neq \emptyset \\
\mathit{Blocks} & \widehat{=} & \|[\mathit{left}]_{0 \leqslant i \leqslant n} P_i \\
\mathit{System} & \widehat{=} & \mathit{Blocks} \underset{\mathit{left},p_0,p_i,\ldots,p_n}{\|} R \\
\mathit{DRB}_n & \widehat{=} & \mathit{System} \setminus \{p_0, p_i, \ldots, p_n\}
\end{array}
$$

Each P_i simply inputs a data item and outputs the result via the corresponding p_i channel. R receives all the results via $p_0, p_1, \ldots, p_n$ on a first-come-first-serve basis. The first result that arrives and passes the acceptance test T is the one to be output via *right*; the rest are simply discarded, without being tested at all. In cases where none of the results passes the acceptance test, the *error* signal is raised and the last result received is output (as defined in $R_{x,\{i\}}$). R is defined by a set of five mutually recursive equations. The first three describe R's behaviour until an acceptable result is received, using a pair of index variables, x and A, that correspond to the input value and the set of outstanding results. The last two equations describe R's behaviour after an acceptable result is received (and output); their definition requires only one index variable corresponding to the set of outstanding results.

Note that the design as such does not yet tolerate any hardware faults. We shall later on make adjustments to this design once we have made the assumption about

the nature of hardware faults that are to be tolerated. For now, all we can say is that the design possesses the suitable "structure" in which, after some adjustments at the component level, the system can continue to operate even after partial hardware failures (though, at a lower degree of diversity).

6. Verification

The verification of the design DRB_n is done in two different ways, one for each style of specification as discussed in Section 4.

6.1. Algebraic transformation and model checking

The verification of DRB_n against its process-oriented specification $SpecDRB_n$ amounts to showing that the behaviour of the former is a refinement of the behaviour of the latter, as stated in the following lemma:

LEMMA 1 $SpecDRB_n \sqsubseteq DRB_n$

Proof: One straightforward way to show the refinement relationship is to apply semantics-preserving transformation upon the process definition based on the algebraic rules associated with the CSP operators [15]. This approach amounts to exploring or enumerating manually all possible states of a process defined by parallel combination. Also, since here we are concerned with the behavioural properties rather than the functional (input-output) properties of the process, we can afford to consider an abstract version of DRB_n which ignores the functional aspect involving values, variables, functions, and boolean expressions. The boolean choice "$\not< b \not>$" becomes an internal (nondeterministic) choice and the interface sets (alphabets) in the parallel combination can be made implicit to facilitate algebraic manipulation. This results in a process $AbsDRB_n$ that is refined by DRB_n. Hence any specification that is satisfied by $AbsDRB_n$ will also be satisfied by DRB_n.

The resulting process $AbsDRB_n$ is *data independent* [23, 33] with respect to the messages passed along the channels, and so it will satisfy $SpecDRB_n$ if an equivalent system DRB'_n which ignores the messages passed along its channels satisfies $SpecDRB_n$. Such a system is defined as follows:

$$
\begin{array}{rcll}
P_i & \widehat{=} & left \rightarrow p_i \rightarrow P_i & \text{for } 0 \leqslant i \leqslant n \\
R & \widehat{=} & left \rightarrow R_{\{i \mid 0 \leqslant i \leqslant n\}} & \\
R_{\{i\}} & \widehat{=} & p_i \rightarrow \left(\begin{array}{l} right \rightarrow R \\ \sqcap \\ error \rightarrow right \rightarrow R \end{array} \right) & \text{for } 0 \leqslant i \leqslant n \\
R_A & \widehat{=} & \Box_{i \in A}\, p_i \rightarrow \left(\begin{array}{l} right \rightarrow R'_{A-\{i\}} \\ \sqcap \\ R_{A-\{i\}} \end{array} \right) & \text{for } \begin{array}{l} A \subseteq \{i \mid 0 \leqslant i \leqslant n\} \\ \wedge\ \#A > 1 \end{array} \\
R'_{\emptyset} & \widehat{=} & R &
\end{array}
$$

$$\begin{aligned} R'_B &\mathrel{\widehat=} \Box_{j\in B}\, p_j \to R'_{B-\{j\}} \quad \text{for } B \subset \{i \mid 0 \leqslant i \leqslant n\} \wedge B \neq \emptyset \\ DRB'_n &\mathrel{\widehat=} (P_0 \parallel P_1 \parallel \ldots \parallel P_n \parallel R) \setminus \{p_0, p_i, \ldots, p_n\} \end{aligned}$$

Consider the simplest case where $n = 1$,

$$\begin{aligned} & DRB'_1 \\ = \; & (P_0 \parallel P_1 \parallel R) \setminus \{p_0, p_1\} \\ = \; & \left(\begin{array}{l} \mathit{left} \to p_0 \to P_0 \\ \parallel \\ \mathit{left} \to p_1 \to P_1 \\ \parallel \\ \mathit{left} \to \left(\begin{array}{l} p_0 \to \left(\begin{array}{l} \mathit{right} \to p_1 \to R \\ \sqcap \\ p_1 \to \left(\begin{array}{l} \mathit{right} \to R \\ \sqcap \\ \mathit{error} \to \mathit{right} \to R \end{array} \right) \end{array} \right) \\ \Box \\ p_1 \to \left(\begin{array}{l} \mathit{right} \to p_0 \to R \\ \sqcap \\ p_0 \to \left(\begin{array}{l} \mathit{right} \to R \\ \sqcap \\ \mathit{error} \to \mathit{right} \to R \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{p_0, p_1\} \\ = \; & \mathit{left} \to \left(\begin{array}{l} p_0 \to \left(\begin{array}{l} (\mathit{right} \to q \to (P_0 \parallel P_1 \parallel R)) \\ \sqcap \\ p_1 \to \left(\begin{array}{l} \mathit{right} \to (P_0 \parallel P_1 \parallel R) \\ \sqcap \\ \mathit{error} \to \mathit{right} \to (P_0 \parallel P_1 \parallel R) \end{array} \right) \end{array} \right) \\ \Box \\ p_1 \to \left(\begin{array}{l} (\mathit{right} \to p_0 \to (P_0 \parallel P_1 \parallel R)) \\ \sqcap \\ p_0 \to \left(\begin{array}{l} \mathit{right} \to (P_0 \parallel P_1 \parallel R) \\ \sqcap \\ \mathit{error} \to \mathit{right} \to (P_0 \parallel P_1 \parallel R) \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{p_0, p_1\} \\ = \; & \mathit{left} \to \left(\begin{array}{l} \mathit{right} \to (P_0 \parallel P_1 \parallel R) \setminus \{p_0, p_1\} \\ \sqcap \\ \mathit{error} \to \mathit{right} \to (P_0 \parallel P_1 \parallel R) \setminus \{p_0, p_1\} \end{array} \right) \\ = \; & \mathit{left} \to \left(\begin{array}{l} \mathit{right} \to DRB'_1 \\ \sqcap \\ \mathit{error} \to \mathit{right} \to DRB'_1 \end{array} \right) \end{aligned}$$

The simplified behaviour of DRB'_1 is equivalent to $SpecDRB_n$ if we ignore all channel messages. In fact, as we increase n, we obtain the same simplified behaviour of DRB'_1 and hence we conclude that $SpecDRB_n \sqsubseteq DRB_n$. ■

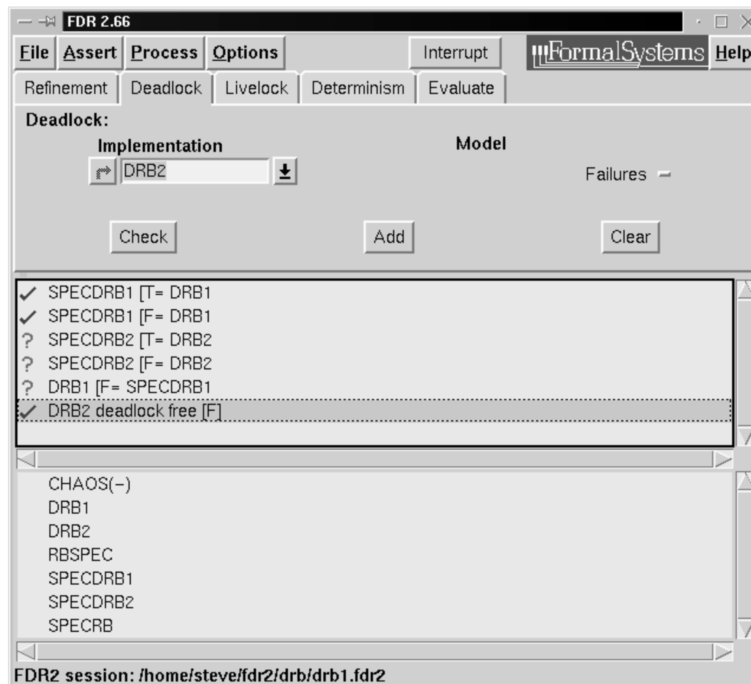

Figure 3. A snapshot of the FDR tool

One may find the algebraic transformation too laborious for practical application. Fortunately, the FDR model checking tool [32, 33] can be used to automatically verify the above lemma for particular values of n. Figure 3 shows a screen shot of using FDR to verify Lemma 1.

6.2. Reasoning with traces and failures

Having proved that DRB_n is a refinement of $SpecDRB_n$, it remains to prove that DRB_n possess the properties S_1, S_2, and L, ie.

LEMMA 2 DRB_n **sat** $S_1 \wedge S_2 \wedge L$

The proof of this lemma is based on the standard proof rules and techniques found in [15, 33, 39] and is given in full in the complete version of this paper [40].

7. Introducing time

The DRB design discussed so far does not address any real-time requirements. Using the language of real-time CSP [31, 38, 39], we are able to incorporate timing

information into the design. This allows for the analysis of real-time behaviour, but it is important to take care that the results already obtained from the untimed analysis are preserved where possible.

The theory of timewise refinement for real-time CSP [38] gives conditions under which timing information can be introduced into an untimed CSP system while preserving the untimed properties. This paper is concerned with two kinds of timewise refinement: with regard to safety properties expressed in terms of traces, and more generally with regard to liveness properties expressed in terms of failures. If P is an untimed CSP process, and Q is a real-time CSP process, then the trace timewise refinement $P \sqsubseteq_t Q$ means that given any timed trace s of Q, the same trace with the times removed—$strip(s)$—is a trace of P. The failures timewise refinement $P \sqsubseteq_f Q$ means that whenever $(s, [t, \infty) \times ref)$ is a timed failure of Q, then $(strip(s), ref)$ is a failure of P: Q's eventual refusal of ref forever corresponds in the untimed world to the refusal set ref.

These characterisations allow timewise refinements to preserve safety and liveness properties. Mathematically, this means that if P **sat** $S(tr)$ and $P \sqsubseteq_t Q$, then Q **sat** $S(strip(s))$; and that if P **sat** $S(tr, ref)$ and $P \sqsubseteq_f Q$, then Q **sat** $(\aleph = [t, \infty) \times ref) \Rightarrow S(strip(s), ref)$. The specifications for Q are the same specifications S translated to the timed setting. For example, untimed deadlock-freedom translates to timed deadlock-freedom, and untimed liveness on a channel translates to timed liveness on a channel: that the channel must always eventually respond.

There are a number of results concerning the introduction of time into process descriptions in ways that yield a timewise refinement of the original untimed process, and thus preserve its properties. We present only the two that are used in this paper, which are derived from more general results of [38]. The first lemma states that introducing time delays preserves both traces and failures timewise refinement:

LEMMA 3 *If Q is obtained from P by replacing some of its event prefix operations '$\rightarrow$' with a timed event prefix '$\xrightarrow{t}$' then $P \sqsubseteq_t Q$ and $P \sqsubseteq_f Q$*

The other lemma is concerned in addition with the replacement of external choice by timeout choice.

LEMMA 4 *If Q is obtained from P by replacing some of its event prefix operations '$\rightarrow$' with timed event prefix, and by replacing some of its external choices '$\Box$' with timeouts '$\triangleright\{t\}$', then $P \sqsubseteq_t Q$.*

This lemma does not hold for failures timewise refinement.

7.1. Real-time requirement

An important timing requirement of our design is

> "Upon receipt of an input, the system must be ready to output a result *or* raise the error signal within δ seconds".

We can specify this requirement based on the timed failure model as follows:

$$RT(s,\aleph) \mathrel{\widehat{=}} \mathit{left} \text{ at } t \Rightarrow$$
$$\quad \mathit{right} \text{ live within } \delta \text{ of } t \text{ until } \mathit{right} \lor$$
$$\quad \mathit{error} \text{ live within } \delta \text{ of } t \text{ until } \mathit{error}$$

Timing information can now be incorporated into the design as follows,

$$\begin{array}{rcl}
P_i & \widehat{=} & \mathit{left}?x \stackrel{\epsilon_i}{\to} p_i!f(x) \to P_i \qquad \text{for } 0 \leqslant i \leqslant n \\
R & \widehat{=} & \mathit{left}?x \to R_{x,\{i|0\leqslant i\leqslant n\}} \\
R_{x,\{i\}} & \widehat{=} & p_i?y \stackrel{\epsilon_T}{\to} \left(\begin{array}{l} \mathit{right}!y \to R \\ \not< T(x,y) \not> \\ \mathit{error} \to \mathit{right}!y \to R \end{array} \right) \quad \text{for } 0 \leqslant i \leqslant n \\
R_{x,A} & \widehat{=} & \Box_{i\in A}\, p_i?y \stackrel{\epsilon_T}{\to} \left(\begin{array}{l} \mathit{right}!y \to R'_{A-\{i\}} \\ \not< T(x,y) \not> \\ R_{x,A-\{i\}} \end{array} \right) \text{for } \begin{array}{l} A \subseteq \{i \mid 0 \leqslant i \leqslant n\} \\ \land \#A > 1 \end{array} \\
R'_{\emptyset} & \widehat{=} & R \\
R'_B & \widehat{=} & \Box_{j\in B}\, p_j?z \to R'_{B-\{j\}} \quad \text{for } B \subset \{i \mid 0 \leqslant i \leqslant n\} \land B \neq \emptyset \\
\mathit{Blocks} & \widehat{=} & |||[\mathit{left}]_{0\leqslant i\leqslant n} P_i \\
\mathit{System} & \widehat{=} & \mathit{Blocks} \underset{\mathit{left},p_0,p_i,\ldots,p_n}{\|} R \\
\mathit{TDRB}_n & \widehat{=} & \mathit{System} \setminus \{p_0, p_i, \ldots, p_n\}
\end{array}$$

We have assigned time values, $\epsilon_0, \epsilon_1, \ldots, \epsilon_n$ and ϵ_T to the computation modules $f_0, f_1, \ldots, f_n$, and acceptance test T, respectively. Note that these values may represent the maximum, minimum, or average amount of time for the modules to compute for each input. Here, since the real-time requirement RT is concerned with maximum response time, we shall consider the time values as maximum time values, i.e. the worst case scenario. On the other hand, we have not specified timing information for communications to take place; their timing is considered negligible here.

Observe that as a result of Lemma 3 the system $DRB_n \sqsubseteq_t TDRB_n$ and $DRB_n \sqsubseteq_f TDRB_n$, so the timed system $TDRB_n$ satisfies its process-oriented specification $SpecDRB_n$ (ie. $SpecDRB_n \sqsubseteq_t TDRB_n$ and $SpecDRB_n \sqsubseteq_f TDRB_n$) and meets the property-oriented specification $S_1 \land S_2 \land L$.

Due to the limited amount of space, the following verification only treats the case of $n = 1$. To show that $TDRB_1$ **sat** RT, we first state some behavioural properties about the component processes.

LEMMA 5

$$P_0 \ \textbf{sat} \ \mathit{left} \text{ at } t \Rightarrow p_0 \text{ live from } t + \epsilon_0 \text{ until } p_0$$

LEMMA 6

$$P_1 \;\; \textbf{sat} \;\; \textit{left} \text{ at } t \Rightarrow p_1 \text{ live from } t + \epsilon_1 \text{ until } p_1$$

LEMMA 7

$$\begin{array}{l} R \;\; \textbf{sat} \;\; \textit{left} \text{ at } t \Rightarrow \\ \qquad p_0 \text{ live from } t \text{ until } \{p_0, p_1\} \wedge \\ \qquad p_1 \text{ live from } t \text{ until } \{p_0, p_1\} \\ R \;\; \textbf{sat} \;\; p_0 \text{ at } t \wedge \text{name of last } \Sigma \text{ before } t = \textit{left} \Rightarrow \\ \qquad \textit{right} \text{ live from } t + \epsilon_T \text{ until } \textit{right} \vee \\ \qquad p_1 \text{ live from } t + \epsilon_T \text{ until } p_1 \\ R \;\; \textbf{sat} \;\; p_1 \text{ at } t \wedge \text{name of last } \Sigma \text{ before } t = \textit{left} \Rightarrow \\ \qquad \textit{right} \text{ live from } t + \epsilon_T \text{ until } \textit{right} \vee \\ \qquad p_0 \text{ live from } t + \epsilon_T \text{ until } p_0 \\ R \;\; \textbf{sat} \;\; p_1 \text{ at } t \wedge \text{name of last } \Sigma \text{ before } t = p_0 \Rightarrow \\ \qquad \textit{right} \text{ live from } t + \epsilon_T \text{ until } \textit{right} \vee \\ \qquad \textit{error} \text{ live from } t + \epsilon_T \text{ until } \textit{error} \\ R \;\; \textbf{sat} \;\; p_0 \text{ at } t \wedge \text{name of last } \Sigma \text{ before } t = p_1 \Rightarrow \\ \qquad \textit{right} \text{ live from } t + \epsilon_T \text{ until } \textit{right} \vee \\ \qquad \textit{error} \text{ live from } t + \epsilon_T \text{ until } \textit{error} \end{array}$$

With the help of the proof rules given in the Appendix, we shall prove that the combined behaviour of P_0 and P_1 should satisfy the following specification:

LEMMA 8

$$\begin{array}{l} \textit{Blocks} \;\; \textbf{sat} \;\; \textit{left} \text{ at } t \Rightarrow \\ \qquad p_0 \text{ live from } t + \epsilon_0 \text{ until } p_0 \wedge \\ \qquad p_1 \text{ live from } t + \epsilon_1 \text{ until } p_1 \end{array}$$

Proof: We outline our proofs following the style adopted in [10].

$$\begin{array}{l} \textit{left} \text{ at } t\{\textit{Blocks}\} \\ \;\vdash\; \textit{left} \text{ at } t\{P_0\} \qquad \textbf{Rule 3} \\ \;\vdash\; p_0 \text{ live from } t + \epsilon_0 \text{ until } p_0\{P_0\} \qquad \textbf{Lemma 5} \\ \;\vdash\; p_0 \text{ live from } t + \epsilon_0 \text{ until } p_0\{\textit{Blocks}\} \qquad \textbf{Rule 2} \end{array}$$

Each proof step above takes the form:

$$\textit{property}\{\textit{Process}\} \qquad \textbf{Justification}$$

We have omitted the observation variables from the predicates, replacing them with the name of the process being observed.

Similarly, we can prove that

$$\begin{array}{l} \mathit{left}\ \mathsf{at}\ t\{\mathit{Blocks}\} \\ \quad \vdash\ p_1\ \mathsf{live\ from}\ t+\epsilon_1\ \mathsf{until}\ p_1\{\mathit{Blocks}\} \end{array}$$

■

Now, we are ready to prove the combined behaviour of *Blocks* and R satisfies RT.

LEMMA 9

$$\begin{array}{lll} \mathit{TDRB}_1 & \mathbf{sat} & \mathit{left}\ \mathsf{at}\ t \Rightarrow \\ & & \mathit{right}\ \mathsf{live\ within}\ \delta\ \mathsf{of}\ t\ \mathsf{until}\ \mathit{right} \vee \\ & & \mathit{error}\ \mathsf{live\ within}\ \delta\ \mathsf{of}\ t\ \mathsf{until}\ \mathit{error} \end{array}$$

Proof: Assume that an input occurs at time t,

$$\begin{array}{l} \mathit{left}\ \mathsf{at}\ t\{\mathit{TDRB}_1\} \\ \quad \vdash\ \mathit{left}\ \mathsf{at}\ t\{\mathit{System}\} \\ \quad \vdash\ \mathit{left}\ \mathsf{at}\ t\{\mathit{Blocks}\} \qquad \textbf{Rule 3} \\ \quad \vdash\ p_0\ \mathsf{live\ from}\ t+\epsilon_0\ \mathsf{until}\ p_0 \wedge \\ \qquad\quad p_1\ \mathsf{live\ from}\ t+\epsilon_1\ \mathsf{until}\ p_1\{\mathit{Blocks}\} \qquad \textbf{Lemma 8} \\ \quad \vdash\ p_0\ \mathsf{live\ within}\ \epsilon_0\ \mathsf{of}\ t\ \mathsf{until}\ p_0 \wedge \\ \qquad\quad p_1\ \mathsf{live\ within}\ \epsilon_1\ \mathsf{of}\ t\ \mathsf{until}\ p_1\{\mathit{Blocks}\} \qquad \textbf{Rule 5} \end{array}$$

On the other hand, we have

$$\begin{array}{l} \quad \vdash\ \mathit{left}\ \mathsf{at}\ t\{R\} \qquad \textbf{Rule 3} \\ \quad \vdash\ p_0\ \mathsf{live\ from}\ t\ \mathsf{until}\ \{p_0,p_1\} \wedge \\ \qquad\quad p_1\ \mathsf{live\ from}\ t\ \mathsf{until}\ \{p_0,p_1\}\{R\} \qquad \textbf{Lemma 7} \\ \quad \vdash\ p_0\ \mathsf{live\ within}\ \epsilon_0\ \mathsf{of}\ t\ \mathsf{until}\ \{p_0,p_1\} \wedge \\ \qquad\quad p_1\ \mathsf{live\ within}\ \epsilon_1\ \mathsf{of}\ t\ \mathsf{until}\ \{p_0,p_1\}\{R\} \qquad \textbf{Rule 5} \\ \quad \vdash\ p_0\ \mathsf{live\ within}\ \epsilon_0\ \mathsf{of}\ t\ \mathsf{until}\ \{p_0,p_1\} \wedge \\ \qquad\quad p_1\ \mathsf{live\ within}\ \epsilon_1\ \mathsf{of}\ t\ \mathsf{until}\ \{p_0,p_1\}\{\mathit{System}\} \qquad \textbf{Rule 4} \end{array}$$

Since p_0 and p_1 are supposed to be internal events not available to, or controllable by, the environment, they occur as soon as all parties participating in them are ready to do so. Assume that $\epsilon_0 < \epsilon_1$ which means p_0 is offered before p_1, we have

$$\begin{array}{l} \quad \vdash\ p_0\ \mathsf{at}\ t+\epsilon_0\{\mathit{System}\} \\ \quad \vdash\ p_0\ \mathsf{at}\ t+\epsilon_0\{R\} \qquad \textbf{Rule 3} \\ \quad \vdash\ p_0\ \mathsf{at}\ t+\epsilon_0 \wedge \mathsf{name\ of\ last}\ \Sigma\ \mathsf{before}\ t+\epsilon_0 = \mathit{left}\{R\} \end{array}$$

$\vdash$ *right* live from $t + \epsilon_0 + \epsilon_T$ until *right* $\vee$
 p_1 live from $t + \epsilon_0 + \epsilon_T$ until $p_1\{R\}$ **Lemma 7**
$\vdash$ *right* live within $\epsilon_0 + \epsilon_T$ of t until *right* $\vee$
 p_1 live within $\epsilon_0 + \epsilon_T$ of t until $p_1\{R\}$ **Rule 5**

We have shown that as from time $t + \epsilon_0 + \epsilon_T$, R is ready to either perform *right*, in which case the timing requirement is met provided that $\epsilon_0 + \epsilon_T < \delta$, or perform p_1. The occurrence of p_1 requires the participation of *Blocks* and this can only happen at the time when both *Blocks* and R are ready to perform p_1, i.e. $\max(t+\epsilon_1, t+\epsilon_0+\epsilon_T)$ where $\max(x,y) = \begin{cases} x \text{ if } x > y \\ y \text{ otherwise} \end{cases}$.

Let us now consider the case where p_1 is performed.

$\vdash$ p_1 live within $\max(\epsilon_1, \epsilon_0 + \epsilon_T)$ of t until $p_1\{R\}$ **Rule 5**
$\vdash$ p_1 live within $\max(\epsilon_1, \epsilon_0 + \epsilon_T)$ of t until $p_1\{System\}$ **Rule 4**
$\vdash$ p_1 at $\max(t+\epsilon_1, t+\epsilon_0+\epsilon_T)\{System\}$ p_1 **is internal**
$\vdash$ p_1 at $\max(t+\epsilon_1, t+\epsilon_0+\epsilon_T) \wedge$
 name of last Σ before $\max(t+\epsilon_1, t+\epsilon_0+\epsilon_T) = p_0\{R\}$
$\vdash$ *right* live from $\max(t+\epsilon_1, t+\epsilon_0+\epsilon_T) + \epsilon_T$ until *right* $\vee$
 error live from $\max(t+\epsilon_1, t+\epsilon_0+\epsilon_T) + \epsilon_T$ until *error*$\{R\}$
 Lemma 7
$\vdash$ *right* live from $t + \epsilon_0 + \epsilon_T$ until *right* $\vee$
 right live from $\max(t+\epsilon_1, t+\epsilon_0+\epsilon_T) + \epsilon_T$ until *right* $\vee$
 error live from $\max(t+\epsilon_1, t+\epsilon_0+\epsilon_T) + \epsilon_T$ until *error*$\{TDRB_1\}$
 Rule 2
$\vdash$ *right* live within $\epsilon_0 + \epsilon_T$ of t until *right* $\vee$
 right live within $\max(\epsilon_1 + \epsilon_T, \epsilon_0 + 2\epsilon_T)$ of t until *right* $\vee$
 error live within $\max(\epsilon_1 + \epsilon_T, \epsilon_0 + 2\epsilon_T)$ of t until *error*$\{TDRB_1\}$
 Rule 5

On the other hand, if we assume that p_1 is offered before p_0, i.e. $\epsilon_1 < \epsilon_0$, we can arrive at

$\vdash$ *right* live within $\epsilon_1 + \epsilon_T$ of t until *right* $\vee$
 right live within $\max(\epsilon_0 + \epsilon_T, \epsilon_1 + 2\epsilon_T)$ of t until *right* $\vee$
 error live within $\max(\epsilon_0 + \epsilon_T, t + \epsilon_1 + 2\epsilon_T)$ of t until *error*$\{TDRB_1\}$

Hence, provided that $\max(\epsilon_0 + 2\epsilon_T, \epsilon_1 + 2\epsilon_T) < \delta$, we have $TDRB_1$ **sat** RT.

■

8. Tolerating hardware faults

The DRB design discussed so far has not assumed any hardware faults. In this section, we are going to make some adjustments to the design at the component level so that the system will continue to operate in the event of hardware failures. But first, we need to make explicit assumptions about the kind of hardware faults that are to be tolerated. Our hardware fault assumption is that any one of the processors whose behaviours are described by $P_0, P_1, \ldots, P_n$ could fail as a result of some hardware faults. When any one of them fails, it behaves like a "sink" on channel *left*, i.e. it continues accepting inputs on channel *left*, but ceases to produce any output on channel p_i. Furthermore, we assume that at most n processors could fail. This can be modelled by requiring that P_0 does not fail, whereas the others can do: P_0 represents the component which does not fail during the execution. With such hardware fault assumption, we can describe the behaviour of the faulty processors as follows:

$$
\begin{aligned}
P_0 &\mathrel{\widehat{=}} \mathit{left}?x \stackrel{\epsilon_i}{\rightarrow} p_i!f_i(x) \rightarrow P_0 \\
P_i &\mathrel{\widehat{=}} \begin{pmatrix} \mathit{left}?x \stackrel{\epsilon_i}{\rightarrow} p_i!f_i(x) \rightarrow (P_i \sqcap P_i') \\ \sqcap \\ P_i' \end{pmatrix} \quad \text{for } 1 \leqslant i \leqslant n \\
P_i' &\mathrel{\widehat{=}} \mathit{left}?x \stackrel{\epsilon}{\rightarrow} P_i' \quad \text{for } 1 \leqslant i \leqslant n
\end{aligned}
$$

Note that we have assumed that a faulty processor could fail right from the start. Our strategy to deal with the possibility of such failure is to set a timeout on each of the channels $p_0, p_1, \ldots, p_n$ in R (since R should treat all the processors as equally likely to fail).

$$
\begin{aligned}
R &\mathrel{\widehat{=}} \mathit{left}?x \rightarrow R_{x,\{i \mid 0 \leqslant i \leqslant n\}} \\
R_{x,\{i\}} &\mathrel{\widehat{=}} p_i?y \stackrel{\epsilon_T}{\rightarrow} \begin{pmatrix} \mathit{right}!y \rightarrow R \\ \nleqslant T(x,y) \ngtr \\ \mathit{error} \rightarrow \mathit{right}!y \rightarrow R \end{pmatrix} \quad \text{for } 0 \leqslant i \leqslant n \\
R_{x,A} &\mathrel{\widehat{=}} \Box_{i \in A}\, p_i?y \stackrel{\epsilon_T}{\rightarrow} \begin{pmatrix} \mathit{right}!y \rightarrow R'_{A-\{i\}} \\ \nleqslant T(x,y) \ngtr \\ R_{x,A-\{i\}} \triangleright\{\varepsilon\} \\ \mathit{error} \rightarrow \mathit{right}!y \rightarrow R \end{pmatrix} \text{for } \begin{array}{l} A \subseteq \{0,1,\ldots,n\} \\ \wedge\ \#A > 1 \end{array} \\
R'_{\emptyset} &\mathrel{\widehat{=}} R \\
R'_B &\mathrel{\widehat{=}} \left(\Box_{j \in B}\, p_j?z \rightarrow R'_{B-\{j\}}\right) \triangleright\{\varepsilon\}\ R \quad \text{for } \begin{array}{c} B \subset \{0,1,\ldots,n\} \\ \wedge \\ B \neq \emptyset \end{array} \\
\mathit{Blocks} &\mathrel{\widehat{=}} \Big|\Big| [\mathit{left}]_{0 \leqslant i \leqslant n} P_i \\
\mathit{System} &\mathrel{\widehat{=}} \mathit{Blocks} \underset{\mathit{left}, p_0, p_i, \ldots, p_n}{||} R \\
\mathit{FTDRB}_n &\mathrel{\widehat{=}} \mathit{System} \setminus \{p_0, p_i, \ldots, p_n\}
\end{aligned}
$$

Since $FTDRB_n$ is obtained from the description of DRB_n through the addition of timeouts and delays, Lemma 4 proves that $DRB_n \sqsubseteq_t FTDRB_n$ which means that $FTDRB_n$ satisfies the (timed versions of the) safety properties S_1 and S_2. However, the theory of timewise refinement cannot be used to establish L, which is a property based on the untimed failure model, since $DRB_n \not\sqsubseteq_f FTDRB_n$. In fact, if we abstract away the timing information in $FTDRB_n$ and apply algebraic transformation as in section 6.1, we obtain (for $n = 1$),

$$
\begin{aligned}
FTDRB_1' &= left \rightarrow \begin{pmatrix} right \rightarrow (FTDRB_1' \sqcap D \sqcap Stop) \\ \sqcap \\ error \rightarrow right \rightarrow (FTDRB_1' \sqcap D \sqcap Stop) \end{pmatrix} \\
D &= left \rightarrow \begin{pmatrix} right \rightarrow D \\ \sqcap \\ error \rightarrow right \rightarrow D \end{pmatrix}
\end{aligned}
$$

which is clearly not a refinement of the process-oriented specification $SpecDRB_n$. The same conclusion is obtained by using the FDR tool. The reason is that, without timing information, the resolution of timeouts becomes nondeterministic and this analysis shows that the untimed version of $FTDRB_n$ *could* deadlock as a result. This happens when a timeout occurs prematurely, when in fact the timeouted P_i is still functioning and eventually becomes ready to output.

Finally, for RT, the proof which can be found in [40] is similar to that given in Section 7 for Lemma 9 but is longer due to the possibility that P_1 could fail.

9. Conclusion and discussion

This paper has presented the design and verification of a simple distributed recovery block (DRB) design in CSP. The DRB scheme handles software design faults based on design diversity. While it is beyond the scope of this paper to discuss about the merit of design diversity, the interested reader is referred to [19, 11] which give empirical and theoretical results on the reliability of fault-tolerant software based on design diversity. It should also be pointed out that there are other approaches to handling software design faults, such as exception handling [7] and data diversity [1]. On the other hand, the DRB design can be considered as an example of a class of system architectures which tolerate both hardware and software faults in a unified way [22]. Typically, such an architecture involves multiple processors executing alternative software modules (developed based on a common specification) and a "decider" which monitors the results of variant execution. The behaviour of each processor can be described by a process in CSP and the safety, liveness, and real-time properties of the whole system can be specified and verified in the same way as in our case study. In particular, we can abstract away the functional aspect of the behaviour in the verification since the design itself is concerned with fault-tolerance capabilities.

CSP has been used in the design and verification of fault-tolerant systems elsewhere [28, 14, 33], but without addressing the real-time dimension. Our treatment

also differs in the modelling of hardware faults — we do not model hardware faults as explicit events. Modelling hardware faults as events allows a finer control over assumptions about their occurrence, but at the cost of additional complexity in proof. In this paper we model the nondeterministic behaviour of faulty processes, and abstract away the hardware faults themselves.

The functional notation f_i to represent the implemented software function in the component is appropriate to represent the systematic nature of software faults. The fact that the design is verified for arbitrary instantiations of the f_i establishes that it is tolerant to such kind of software fault. An alternative approach to modelling the behaviour of faulty software modules would be simply to allow a process P_i to nondeterministically provide any output after an input:

$$P_i = \mathit{left}?x \rightarrow \bigsqcap_y \mathit{right}!y \rightarrow P_i$$

The acceptance test T will still be used to judge whether or not y is an acceptable output for the input x. This approach could be used to model 'glitches' (transient hardware faults) in the computation of the function, where the output may vary on different occurrences of the same input. However, the liveness property L becomes inapplicable, so only weaker results may be obtained for the design. In particular, the property that any output from the system must be the result of one of the computations is lost.

The use of two different but compatible semantic models in capturing and verifying different kinds of requirements has shown practical benefits: safety and liveness properties that do not address timing can be handled by the failures model which is relatively simple and allows algebraic manipulation and model checking, whereas real-time properties are addressed using the more sophisticated timed failures model. The preservation of safety and liveness properties from one model to another is taken care of by the theory of timewise refinement. The introduction of the timed model could also allow other kinds of faults, such as intermittent hardware faults, to be modelled, and the behaviour of the resulting system analysed.

The failure and timed failure semantics of CSP allow us to treat the behaviour of faulty hardware components with non-determinism. In this approach, the normal and faulty behaviours of a (non-deterministic) process have no fundamental difference — they are both (timed) failures of the same process. Coenen and Hooman [5, 6] have proposed a compositional semantics for fault-tolerant real-time systems in which faulty behaviours are distinguished from normal behaviours. Their approach is intended to facilitate the specification and reasoning of fault-tolerant systems by separating normal behaviours from faulty behaviours. However, it does not provide any way to define such faulty behaviours, leaving them totally arbitrary.

Hansson [13] has extended CCS [25] with real-time and probabilities and applied the formalism to fault-tolerant real-time systems. The extensions allow reliability and performance to be modelled since reliability requirements can be stated in terms of probabilities of desirable or undesirable behaviours, e.g. the reliability of P_i in our design can be formulated in terms of a probability of failing to produce an output. A timed probabilistic version of CSP has also been developed [24] but has not been applied to real-time fault-tolerance.

Acknowledgments

The authors are grateful to Francis Tam for his contribution on the initial version of this paper and to Simon Gay and Tomasz Janowski for their constructive comments and suggestions on later versions of this paper. Comments and suggestions from the anonymous referees are also gratefully acknowledged. The authors are also grateful to the British Council for a travel grant under the UK/Hong Kong Joint Research Scheme which has supported this work. The second author is grateful for DERA for supporting this work.

References

1. P.E. Ammann and J.C. Knight. Data Diversity: An Approach to Software Fault Tolerance. In *17th International Symposium on Fault Tolerant Computing Systems*, pages 122–126, 1987.
2. S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31:560–599, 1984.
3. A. Cau and W.-P. de Roever. Specifying fault-tolerance within stark's formalism. In *Proc. 23rd Symp. on Fault-Tolerant Comp.*, pages 392–401. IEEE Computer Society Press, 1993.
4. G.H. Chisholm and A.S. Wojcik. An Application of Formal Analysis to Software in a Fault-Tolerant Environment. *IEEE Transactions on Computers*, 48(10):1053–1063, October 1999.
5. J. Coenen and J. Hooman. A compositional semantics for fault-tolerant real-time systems. In J. Vytopil, editor, *Proc. Second International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 33–51, Nijmegen, The Netherlands, January 1992. Springer-Verlag.
6. J. Coenen and J. Hooman. Parameterized semantics for fault tolerant real-time systems. In J. Vytopil, editor, *Formal Techniques in Real-Time Fault-Tolerant Systems*, pages 51–78. Kluwer Academic Publishers, 1993.
7. F. Cristian. Exception Handling and Software Fault Tolerance. *IEEE Transactions on Computers*, C-31(6):531–540, June 1982.
8. F. Cristian. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering*, SE-11(1):23–31, 1985.
9. J.W. Davies. *Specification and Proof in Real-Time Systems.* Cambridge University Press, 1993.
10. J.W. Davies and S.A. Schneider. Real-time CSP. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-time System Development*, volume 2. World Scientific, 1995.
11. D.E. Eckhardt and L.D. Lee. A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincidental Errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, December 1985.
12. Tom R. Halfhill. The Truth Behind the Pentium Bug. *Byte*, March 1995.
13. H. A. Hansson. Modeling real-time and reliability. In J. Vytopil, editor, *Formal Techniques in Real-Time Fault-Tolerant Systems*, pages 79–105. Kluwer Academic Publishers, 1993.
14. Jifeng He and C.A.R. Hoare. Algebraic specification and proof of a distributed recovery algorithm. *Distributed Computing*, 2:1–12, 1987.
15. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
16. J. J. Horning et al. A Program Structure for Error Detection and Recovery. In E. Gelenbe and C. Kaiser, editors, *Lecture Notes in Computer Science, 16*, volume 16, pages 171–187. Springer Verlag, 1974.
17. M. Joseph, A. Moitra, and N. Soundararajan. Proof rules for fault-tolerant distributed programs. *Science of Computer Programming*, 8:43–67, 1987.
18. 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.

19. J.C. Knight and N.G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.
20. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 1(3):872–923, 1994.
21. L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In H. Langmaak, W.-P. de Roever, and J. Vytopil, editors, *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 42–76. Springer-Verlag, 1994.
22. Jean-Claude Laprie et al. Definition and analysis of hardware- and software-fault-tolerant architectures. *IEEE Computer*, 23(7):39–51, July 1990.
23. R. Lazic. *A semantic study of data-independence with applications to the mechanical verification of concurrent systems.* PhD thesis, Oxford University, 1997.
24. G. Lowe. *Probabilities and priorities in timed CSP.* D.Phil thesis, Oxford University, 1993.
25. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
26. A.W. Roscoe M.W. Mislove and S.A. Schneider. Fixed points without completeness. *Theoretical Computer Science*, 138, 1995.
27. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich Von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
28. Jan Peleska. Design and verification of fault tolerant systems with CSP. *Distributed Computing*, 5:95–106, 1991.
29. B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.
30. G.M. Reed. *A uniform mathematical theory for real-time distributed computing.* D.Phil thesis, Oxford University, 1988.
31. G.M. Reed and A.W Roscoe. A timed model for communicating sequential processes. In *13th ICALP*, volume 226 of *LNCS*, pages 314–323. Springer-Verlag, 1986.
32. A.W. Roscoe. Model checking CSP. In *A Classical Mind: Essays in Honour of C.A.R. Hoare.* Prentice Hall, 1994.
33. A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.
34. Henk Schepers. Real-time systems and fault-tolerance. In Mathai Joseph, editor, *Real-Time Systems: Specification, Verification and Analysis*, chapter 6, pages 229–257. Prentice Hall, 1996.
35. R.D. Schlichting and F.B. Schneider. Fail-stop processors: an approach to designing fault tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.
36. F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comp. Surveys*, 22(4):299–319, 1990.
37. S.A. Schneider. Unbounded nondeterminism for real-time processes. Technical Report 13-92, Oxford University, 1992.
38. S.A. Schneider. Timewise refinement for communicating processes. *Science of Computer Programming*, 28:43–90, 1997.
39. S.A. Schneider. *Concurrent and Real-time Systems: The CSP Approach.* John Wiley, 2000.
40. W.L. Yeung, S. A. Schneider, and F. Tam. Design and verification of distributed recovery blocks in CSP. Technical Report CSD-TR-98-08, Royal Holloway, University of London, 1998.

Appendix - Proof rules used in this paper

The following proof rules have been taken from [39].

RULE 1 *Parallel combination*

$$\frac{\begin{array}{l} P \textbf{ sat } S(s,\aleph) \\ Q \textbf{ sat } T(s,\aleph) \end{array}}{P \underset{A}{\|} Q \textbf{ sat } \exists\, s_P, s_Q, \aleph_P, \aleph_Q \bullet \begin{array}{l} S(s_P,\aleph_P) \wedge T(s_Q,\aleph_Q) \wedge \\ (s,\aleph) \in (s_P,\aleph_P) \underset{A}{\|} (s_Q,\aleph_Q) \end{array}}$$

$(s_P,\aleph_P) \underset{A}{\|} (s_Q,\aleph_Q)$ is the set of all compound behaviours of $P \underset{A}{\|} Q$ that can arise from those concurrent behaviour of P and Q.

RULE 2

$$\frac{\begin{array}{l} a \textsf{ live from } t \textsf{ until } a(s_P,\aleph_P) \\ (s,\aleph) \in (s_P,\aleph_P) \underset{A}{\|} (s_Q,\aleph_Q) \end{array}}{a \textsf{ live from } t \textsf{ until } a(s,\aleph)} [a \notin A]$$

In this rule, if the processes do not synchronise on event a and one of them is live on a (until a occurs), then so is the combination. Note that this rule is symmetric for the component processes P and Q with respect to the parallel composition.

RULE 3

$$\frac{\begin{array}{l} a \textsf{ at } t(s,\aleph) \\ (s,\aleph) \in (s_P,\aleph_P) \underset{A}{\|} (s_Q,\aleph_Q) \end{array}}{a \textsf{ at } t(s_P,\aleph_P)} [a \in A]$$

If the processes do synchronise on event a, when a occurs, both processes must perform the event at the same time. The same remark on Rule 2 about symmetry also applies here.

RULE 4

$$\frac{\begin{array}{l} a \textsf{ live within } \delta \textsf{ of } t \textsf{ until } A_1(s_P,\aleph_P) \\ a \textsf{ live within } \delta \textsf{ of } t \textsf{ until } A_2(s_Q,\aleph_Q) \\ (s,\aleph) \in (s_P,\aleph_P) \underset{A}{\|} (s_Q,\aleph_Q) \end{array}}{a \textsf{ live within } \delta \textsf{ of } t \textsf{ until } A_1 \cup A_2(s,\aleph)} \quad [a \in A]$$

The liveness of a common event in a parallel combination is governed by both of the component processes.

Rule 5

$$\frac{a \ \textsf{live from} \ t+\varepsilon \ \textsf{until} \ A(s,\aleph)}{a \ \textsf{live within} \ \delta \ \textsf{of} \ t \ \textsf{until} \ A(s,\aleph)} \quad [\varepsilon \leqslant \delta]$$

This rule allows us to weaken a “live from” specification into a corresponding “live within” specification.