Abstraction and testing

Steve Schneider
Department of Computer Science
Royal Holloway, University of London
Egham, Surrey, TW20 0EX, UK

Abstract

Restricted views of process behaviour result in a form of abstraction which is useful in the construction of specifications involving fault-tolerance and atomicity. This paper presents an operational characterisation of abstraction for refusable and non-refusable events, in terms of testing. This view is related to standard notions of testing, and to the denotational characterisations, and it is encapsulated within the CSP denotational semantics. It informs, reinforces and extends the traditional denotational approach to abstraction.

Contents

1 Introduction

Abstraction arises in a system when some of its activity is not directly observable. In such cases, the nature of the internal activity may only be indirectly inferred from the visible behaviour of the process. A form of abstraction thus arises when an observer only has a restricted view of the system. This situation may arise for example in the context of fault-tolerant systems if an observer interacts with the system on particular activities, but is unconcerned with the fault-tolerance mechanisms, which may be considered as abstracted. It also arises in the context of security, where a 'low-level' user should be prevented from achieving any interaction with the more secret parts of the system.

Process algebraic approaches to abstraction generally provide some operators to provide abstraction. In the context of CSP, which is the concern of this paper, the hiding operator (often called the 'abstraction' operator) provides a mechanism for removing particular events from the process' interface and making them internal. Events internalised in this way become urgent (non-refusable). However, Roscoe [Ros97] has observed that events in a process' interface generally fall the two categories of refusable and non-refusable, and that both kinds of event can be abstracted. This has led to a more sophisticated understanding of abstraction, expressed within the CSP language and in terms of its denotational semantics.

The aim of this paper is to provide an alternative understanding of abstraction, by taking an operational view in terms of testing. This will provide a more direct definition of abstraction, independently of any particular process algebra, and will thereby provide a more explicit and intuitive characterisation. This may inform and throw new light on the denotational approach.

2 Notation

CSP is an abstract language designed to describe the communication patterns of processes in terms of events that they may engage in. For a fuller introduction to the language and the semantic models, the reader is referred to [Ros97].

In CSP, systems are modelled in terms of the events that they can perform. The set of all possible events (fixed at the beginning of the analysis) is denoted Σ. Events may be atomic in structure or may consist of a number of distinct components or fields. Examples of events used in this paper are l_n and h_r, which are atomic events, and $in.3$ which is a compound event modelling the occurrence of the message 3 along channel in.

Processes are the entities that are described by CSP expressions, and they are described in terms of the possible events that they may engage in. The process RUN_A is repeatedly willing to engage in any event from the set A. The process $CHAOS_A$ is able to repeatedly engage in events from A, but might at any time nondeterministically refuse to perform any. The process $STOP$ is unable to perform any events.

The prefixed process $a \rightarrow P$ is able initially to perform only a, and subse-

quently to behave as P. The prefix choice process $x : A \to P(x)$ offers a choice of all the events in the set $A \subseteq \Sigma$, and its subsequent behaviour $P(x)$ is determined by the event chosen. The output $c!v \to P$ is able initially to perform only $c.v$, the output of v on channel c, after which it behaves as P. The input $c?x : T \to P(x)$ can accept any input x of type T along channel c, following which it behaves as $P(x)$. If the type T of the channel is clear from the context then it may be elided from the input, which becomes $c?x \to P(x)$.

Its first event will be any event of the form $c.t$ where $t \in T$. The process $P \,\Box\, Q$ (pronounced 'P external choice Q') is initially willing to behave either as P or as Q, with the choice resolved (by the process' environment) on performance of the first event. The process $P \sqcap Q$ (pronounced 'P internal choice Q') can behave either as P or as Q, and the environment of the process has no control over which. The process $P \setminus A$ behaves as P, but with all of the events in A performed internally where they were previously external events.

Processes may also be composed in parallel. If D is a set of events then the process $P \,|[\, D \,]|\, Q$ behaves as P and Q acting concurrently, with the requirement that they have to synchronise on any event in the synchronisation set D; events not in D may be performed by either process independently of the other. Interleaving is a special form of parallel operator in which the two components do not interact on any events: it is written $P \,|||\, Q$, and is equivalent to $P \,|[\, \{\} \,]|\, Q$. There is also an indexed form $\Big|||_{i \in I} P_i$.

Processes may also be recursively defined by means of equational definitions.

The operational semantics of CSP processes defines, for any particular process, which events the process can perform next, and the possible processes that can be subsequently . For example, the process $a \to P$ can perform an a event and reach the process P. This is written $(a \to P) \xrightarrow{a} P$. The process $(a \to P) \setminus \{a\}$ can perform an internal event, denoted τ, and reach the process $P \setminus \{a\}$. This is written $(a \to P) \setminus \{a\} \xrightarrow{\tau} (P \setminus \{a\})$. For the full operational semantics of CSP, see [Sch98].

The *traces* of a process P, $\mathit{traces}(P)$, is defined to be the set of finite sequences of events from Σ that P may possibly perform. Examples of traces include the empty trace $\langle\rangle$, and $\langle in.3, out.3, in.5\rangle$ which is a possible trace of the recursive process $COPY = in?x : T \to out!x \to COPY$. The set of infinite traces, $\mathit{infinites}(P)$, are the infinite sequences of events from Σ that P might possibly perform.

The *failures* of a process P, $\mathit{failures}(P)$, is defined to be the set of trace/refusal pairs (tr, X) that P can exhibit, where tr is a trace and X is a set of events that P can *refuse* to participate in after some execution of the sequence of events tr. Examples of failures include the empty failure $(\langle\rangle, \varnothing)$, and the trace/refusal pair $(\langle in.3, out.3, in.5\rangle, \{out.3, out.4\})$ which is a possible failure of $COPY$. The *divergences* of a process P, $\mathit{divergences}(P)$, are those sequences of events during whose occurrence P might perform an infinite sequence of internal events. The *FDI* semantics of a process P consists of the three sets of failures, divergences, and infinite traces of P: $\mathit{FDI}(P) = \langle \mathit{failures}(P), \mathit{divergences}(P), \mathit{infinites}(P)\rangle$. All the denotational models for CSP are covered in detail in [Ros97].

The 'after' operator which gives the behaviour of a process P after a trace tr is written as $P \,/\, tr$. It is a partial operator (since tr might not be a trace of P), giving those failures, divergences, and infinite traces of P which are subsequent to the performance of the events in tr.

A process P is *deterministic* if it is unable to refuse events that it can do:

$$(tr, X) \in \mathit{failures}(P) \wedge (tr \frown \langle a \rangle, \varnothing) \in \mathit{failures}(P) \quad \Rightarrow \quad (tr, X \cup \{a\}) \notin \mathit{failures}(P)$$

If P is deterministic, then it must be deterministic at all times: $P \,/\, tr$ is also deterministic, for any trace tr of P.

3 Testing

Abstraction will be characterised in terms of the interfaces through which processes (and tests) can interact, and in terms of distinguishing refusable from non-refusable events.

We are interested in testing processes P by means of test processes T, which can interact with P only through an interface L (where L stands for low-level events). The set H will denote all the events that are not in the set L. These are the events that P might engage in but which the test T has no access to. Thus the set of all events $\Sigma = L \cup H$, where $L \cap H = \varnothing$.

The set L is itself divided up into 'refusable' events LR and 'non-refusable' events LN, where 'refusable' is from the point of view of the environment of the process P under test (i.e., whether the environment is in a position to refuse them). For example, output events of P are generally non-refusable, whereas input events are generally refusable, e.g. the environment can refuse to press a key on the keyboard, but it cannot refuse to allow an event to appear on the screen. An related view is to consider the set LN as those events whose occurrence is entirely under the control of the process P, whereas LR consists of those events which require cooperation from the environment.

H is also divided up into the disjoint sets HR and HN.

Four subsets of Σ are thus identified: HN, HR, LN and LR. These sets are pairwise disjoint, and their union is Σ.

This partition of Σ will be characterised by a function p which indicates for each event which set it is in:

$$p : \Sigma \to \{hn, hr, ln, lr\}$$

The relationship between p and the sets of the partition is that

$$\begin{aligned} HN_p &= \{a \mid p(a) = hn\} \\ HR_p &= \{a \mid p(a) = hr\} \\ LN_p &= \{a \mid p(a) = ln\} \\ LR_p &= \{a \mid p(a) = lr\} \end{aligned}$$

The sets H, L, HN, HR, LN and LR are dependent on the partition function p and so will generally be subscripted with the p. In this paper, we follow the

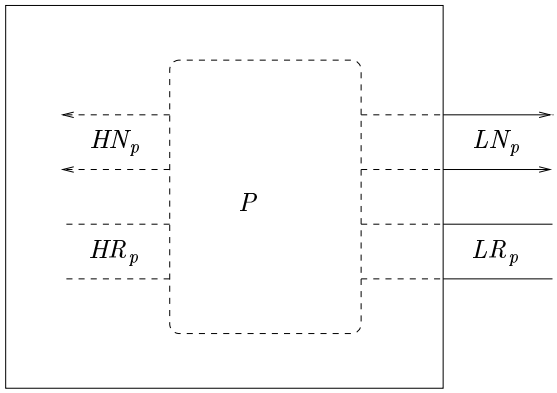

Figure 1: Partitioning P's interface

convention that $l_n \in LN_p$, $l_r \in LR_p$, $h_n \in HN_p$, $h_r \in HR_p$. This partition of an interface is illustrated in Figure 1.

The testing relations will be defined with respect to a partition function p.

We now define the notion of a LN_p test:

DEFINITION 3.1 a LN_p test T is a CSP process with alphabet $L_p \cup \{\omega\}$ which can never refuse any of the events in LN_p[1]. □

This means that at any stage T should either have an internal transition (indicating that it is not stable at that point), or else it should have a transition for every event in LN_p. The alphabet of a LN_p test contains the event ω, a special 'success' event not appearing in either H_p or L_p. The set of all LN_p tests is denoted $TEST_{LN_p}$.

An execution of a process P is a finite or infinite sequence of transitions e as follows:

$$e = P \xrightarrow{\mu_1} P_1 \xrightarrow{\mu_2} P_2 \ldots$$

where each step $P_i \xrightarrow{\mu_{i+1}} P_{i+1}$ is a step given by the operational semantics of CSP. The first process expression is P. The states appearing in this execution, $states(e)$, are P_1, P_2, etc.

For example,

$$(a \rightarrow STOP \sqcap b \rightarrow STOP) \xrightarrow{\tau} (b \rightarrow STOP) \xrightarrow{b} STOP$$

is a finite execution. If $P = a \rightarrow P$ is a recursively defined process, then

$$P \xrightarrow{\tau} (a \rightarrow P) \xrightarrow{a} P \xrightarrow{\tau} (a \rightarrow P) \xrightarrow{a} \ldots$$

[1] After a success state (see later) has been reached, we will sometimes elide the requirement to be unable to refuse LN_p in tests given in some examples. This makes no difference in this paper since the part of an execution after success is irrelevant for our purposes.

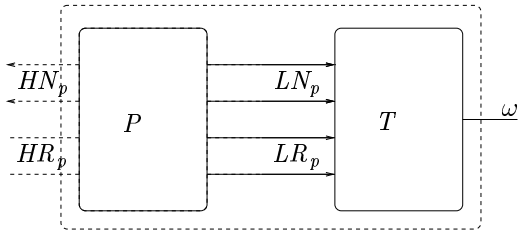

Figure 2: Testing P

is an infinite execution.

To test a process P with a test T and interface L_p, executions e of the process $(P \,|[\, L_p \,]|\, T) \setminus L_p$ are considered. This arrangement is illustrated in Figure 2.

An execution e is successful if and only if there is some process expression $P_i \in \mathit{states}(e)$ from which an ω event is possible:

DEFINITION 3.2

$$e \text{ is successful} \quad \Leftrightarrow \quad \exists\, P \in \mathit{states}(e) \bullet P \xrightarrow{\omega}$$

□

Different notions of testing are captured by different modalities of successful testing: may testing is concerned with the possibility of successful execution, and must testing is concerned with the guarantee of successful execution.

It is a well-known result [Hen88] that standard may and must testing correspond directly to the denotational semantics: two processes are may testing equivalent precisely when they have the same trace semantics; and two processes are must testing equivalent if and only if they have the same *FDI* semantics.

3.1 May testing

The notion of may_p testing can now be defined. For generality and consistency of approach with must_p testing introduced later, it is parametrised by the partition function p. However, observe that it is independent of the way H_p and L_p are themselves partitioned into refusable and non-refusable events. Thus the refusability or otherwise of high-level events is irrelevant to may testing.

DEFINITION 3.3 If P is a process, and T is a LN_p test, then P may_p T if and only if there is some successful execution e of $(P \,|[\, L_p \,]|\, T) \setminus L_p$ □

Observe that there will be some successful execution of $(P \,|[\, L_p \,]|\, T) \setminus L_p$ if and only if there is some successful execution of $P \,|[\, L_p \,]|\, T$. The events in L_p are hidden to make the definition similar to the definition for must_p testing which will come later.

A test T has access only to the events L_p that P can perform, and not the high-level events H_p. In other words, T can interact with P only on the events in L_p, so the behaviour of T in the test will be independent of the events in H_p that P performs. Furthermore, the non-refusable events LN_p that P might perform cannot be blocked by T (at least, not before an ω success event), though the subsequent behaviour of T might depend on which such events were performed. The test thus allows its result to depend on the observation of events from LN_p even though T has no control over their occurrence.

EXAMPLE 3.4 Let $LN_p = \{a_l, b_l\}$. Define

$$\begin{aligned} P_1 &= a_l \rightarrow STOP \\ P_2 &= b_l \rightarrow STOP \end{aligned}$$

Any LN_p test cannot block a_l or b_l. However, their possible success can depend on which of these is performed. The LN_p test

$$\begin{aligned} T &= a_l \rightarrow (\omega \rightarrow RUN_{LN_p} \,\Box\, RUN_{LN_p}) \\ &\quad \Box\ b_l \rightarrow RUN_{LN_p} \end{aligned}$$

only allows the success event after a single a_l. Thus it may succeed with P_1, but cannot with P_2: P_1 may$_p$ T but $\neg(P_2$ may$_p$ $T)$.

Note that this test, with the requirement to be live on a_l and b_l elided after success, would become

$$(a_l \rightarrow \omega \rightarrow STOP) \,\Box\, (b_l \rightarrow RUN_{LN_p})$$

□

Two processes will be defined to be may$_p$ testing equivalent if they may pass exactly the same LN_p tests:

DEFINITION 3.5 $P \equiv_{\mathsf{may_p}} Q$ if and only if $\forall\, T : TEST_{LN_p} \bullet (P$ may$_p$ $T \Leftrightarrow Q$ may$_p$ $T)$ □

This means that if only events in L can be observed, then P and Q should be considered equivalent if any test which has access only to the events in L (a particular view of the process), and does not block the events in LN_p, cannot tell the difference between P and Q.

The following relationship between this form of may$_p$ testing and the standard de Nicola/Hennessy form of may testing [dNH87] makes it clear that this is a generalisation, in that may testing is simply may_{p_0} testing for a particular partition p_0.

If p_0 is the partition that considers all events as low-level refusable events, then all events will be visible to the testing process, which has no constraints on being required to accept any of them. In this case $LR_{p_0} = \Sigma$, and H_{p_0} and LN_{p_0} are all $\varnothing$. It then turns out that may_{p_0} testing is equivalent to the standard notion of may testing. In this case, the testing process has access to all the

events that the processes under test can perform; and is able to block any of them, which is exactly the situation in may testing. This is the most powerful kind of test within this framework: the one which allows the most distinctions to be made.

In fact, given any set of non-refusable events LN_p, any arbitrary CSP test T can be converted to a LN_p test T' such that P may $T \Leftrightarrow P$ may T' for any process P. This is achieved by introducing the extra possibility $\ldots \Box RUN_{LN_p}$ to every state. No new successful executions are introduced, since whenever any of these choices are taken then there is no possibility of reaching ω; so any successful execution of $P \,|[\, L \,]|\, T'$ must correspond to a successful execution of $P \parallel T$.

EXAMPLE 3.6 If T is the test

$$a_l \rightarrow b_l \rightarrow \omega \rightarrow STOP$$

then for $LN_p = \{a_l, b_l\}$, the corresponding test T' which is never able to refuse either a_l of b_l will be given as

$$\begin{array}{lcl} T' & = & RUN_{LN_p} \\ & & \Box\ a_l \rightarrow \ RUN_{LN_p} \\ & & \qquad \Box\ b_l \rightarrow \ RUN_{LN_p} \\ & & \qquad\qquad \Box\ \omega \rightarrow RUN_{LN_p} \end{array}$$

This is illustrated in Figure 3 □

This means that if two processes are equivalent under may testing for a particular set L, then the precise nature of LN_p and LR_p are not relevant—they will remain equivalent whatever the sets LN_p and LR_p happen to be, subject to their union remaining L.

Hence may testing need only be parametrised by the set L; since the finer distinctions made by LN_p and LR_p make no difference at the level of may testing.

A straightforward consequence of the definitions is as follows:

LEMMA 3.7 If $P \equiv_{\mathsf{may_p}} Q$ and $L_{p'} \subseteq L_p$ then $P \equiv_{\mathsf{may_{p'}}} Q$ □

Proof If T is a test with respect to p', (so the alphabet of T is $L'_p \cup \{\omega\}$), then observe that $P \,|[\, L_{p'} \,]|\, T$ has a successful execution if and only if $P \,|[\, L_p \,]|\, (T \,|||\, RUN_{L_p \setminus L_{p'}})$ has a successful execution. Hence if all tests with respect to p identify P and Q, then all tests of the form $(T \,|||\, RUN_{L_p \setminus L_{p'}})$ do so, so P and Q are identified under $\text{may}_{p'}$ testing equivalence. □

This lemma states that if some low-level events are promoted to become high-level events, then any processes that were previously equivalent will remain so: testing processes have access to even less information. If two processes cannot be distinguished by any tests which have access to L_p, then they will never be distinguished by any tests which have access to a smaller set of events $L_{p'} \subseteq L_p$; in fact this smaller set of tests is subsumed within the previous set of tests.

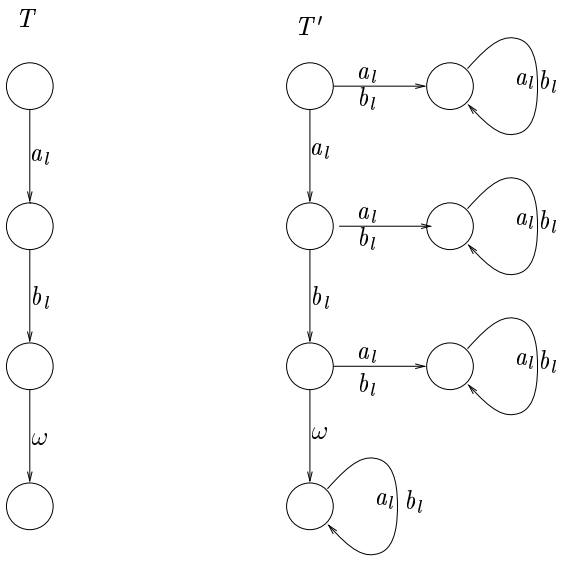

Figure 3: Converting Example 3.6's test

Refinement

The operational characterisation of may testing equivalence generalises in the usual way to may_p testing refinement. Essentially, Q will be a refinement of P if any test T that Q may succeed can also allow P to succeed:

DEFINITION 3.8

$$P \sqsubseteq_{\mathsf{may}_p} Q \quad \Leftrightarrow \quad \forall\, T : TEST_{LN_p} \bullet Q \text{ may}_p\ T \Rightarrow P \text{ may}_p\ T$$

□

The traditional view of refinement considers Q as 'an implementation of P, where P encapsulates or describes the permissible behaviour. From the point of view of may testing, this means that if Q is able to interact with its environment in some way, then this should also be allowed by the specification P: in other words, P should also be able to interact with the same environment to achieve the same result. The process Q should never be able to achieve anything not permitted by P.

For example,

$$l1 \rightarrow STOP \,\Box\, l2 \rightarrow STOP \quad \sqsubseteq_{\mathsf{may}_p} \quad l1 \rightarrow STOP$$

Any test which the right hand process may pass may also be passed by the left hand process.

3.2 Must testing

Must testing can be considered as the dual of may testing. In may testing, a process may pass a test if there is some successful execution. In must testing, a process must pass a test if *every* execution is successful. Since partial executions might not reach a success state because the execution has not run for long enough, attention is focused onto *complete* executions[2].

An execution e of a process P will be considered to be complete if P could be unable to extend it. This will certainly be the case if e is infinite, but it will also arise if e finishes in a stable state where only refusable events are possible—since the process might be prevented from continuing its execution. However, if there are any non-refusable events available, then the execution is not complete since it is entirely within the process' control to continue the execution.

DEFINITION 3.9 An execution e is *complete* with respect to (a set of non-refusable events) N if

1. e is infinite; or
2. e is finite, and the last state Q of e can perform neither internal transitions nor transitions from N.

□

The point is to consider the execution as complete even if refusable events are possible. Such executions could be complete executions if the process is placed in a high-level environment (which the test cannot know about) in which such events are blocked.

Must testing can now be defined:

DEFINITION 3.10 If P is a process, and T is an LN_p test, then P must_p T if and only if every complete execution (with respect to $LN_p \cup HN_p$) of $(P \,|[\, L_p \,]|\, T) \setminus L_p$ is successful. □

EXAMPLE 3.11 Let P_1 be the process:

$$\begin{aligned} P_1 \quad = \quad & h_r \rightarrow h_n \rightarrow l_r \rightarrow l_n \rightarrow P_1 \\ & \Box\ l_r \rightarrow l_n \rightarrow P_1 \end{aligned}$$

Define the test $T_1 = l_n \rightarrow RUN_{l_n} \ \Box\ l_r \rightarrow l_n \rightarrow \omega \rightarrow STOP$. This will succeed as long as the process under test (1) cannot perform l_n before l_r, and (2) is guaranteed to be able to accept l_r and then provide l_n.

Then P_1 must T. If its choice is in favour of l_r, then the complete execution is successful. If its choice is in favour of the high-level event, then the complete execution must include the second high-level event since this is not refusable, and so progress to the low-level events and hence to the success state.

[2] In the standard approach to testing, these executions are the *maximal* ones since they cannot be extended at all. In our setting they are not necessarily maximal, since finite complete executions might be extendable with refusable events

On the other hand, for the test $T_2 = l_n \rightarrow \omega \rightarrow STOP$, we have that $\neg(P_1 \ \mathsf{must} \ T_2)$. The test does not provide l_r, and so no low-level interaction between the process and the test is possible; the success state will not be reached. □

Two processes will be must_p testing equivalent if they must pass exactly the same tests:

DEFINITION 3.12 $P \equiv_{\mathsf{must_p}} Q$ if and only if $\forall\, T : TEST_{LN_p} \bullet (P \ \mathsf{must}_p \ T \Leftrightarrow Q \ \mathsf{must}_p \ T)$ □

EXAMPLE 3.13 If

$$\begin{aligned} P_2 \quad = \quad & h_{r2} \rightarrow h_{n2} \rightarrow l_r \rightarrow l_n \rightarrow P_2 \\ & \Box \ l_r \rightarrow l_n \rightarrow P_2 \end{aligned}$$

and

$$\begin{aligned} P_3 \quad = \quad & h_{r2} \rightarrow \quad h_{n2} \rightarrow l_r \rightarrow l_n \rightarrow P_3 \\ & \qquad\qquad \sqcap \ h_n \rightarrow l_r \rightarrow l_n \rightarrow P_3 \\ & \Box \ l_r \rightarrow l_n \rightarrow P_3 \end{aligned}$$

then P_1, P_2, and P_3 are all must_p equivalent. A process which interacts with them only on l_r and l_n will be unable to distinguish between them. □

This means that two processes P and Q should be considered equivalent (from the point of view of the interface information S) if any test which has access only to the events in L (a particular view of the process), and does not block the events in LN_p, cannot tell the difference between P and Q.

If p_0 is such that $LR_{p_0} = \Sigma$, and H_{p_0} and LN_{p_0} are all $\varnothing$, then $\mathsf{must}_{\ p_0}$ testing is equivalent to the standard notion of must testing. In this case, the testing process has access to all the events that the processes under test can perform; and is able to block any of them. This is the most powerful kind of test within this framework: it allows the most distinctions to be made.

In other words, must testing is equivalent to $\mathsf{must}_{\ p_0}$ testing.

From the point of view of must testing, it makes a difference whether the high-level events are refusable or not, since we are concerned with liveness and progress. For example, $h \rightarrow l \rightarrow STOP$ is must_p equivalent to $l \rightarrow STOP$ if h is not refusable, but the two processes are distinguishable if h is refusable. In that case the first process might be blocked from performing the event h, and never reach the stage where it offers the event l. This cannot happen for the second process. One test which distinguishes these two processes is $T = l \rightarrow \omega \rightarrow RUN_l$.

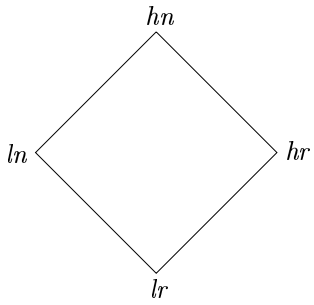

Figure 4: The partial order $\leqslant$ on the interface partition

3.3 Changing views

The notion of abstraction is bound up in the available views of a process as given by the sets L and H, and also by the distinction between refusable and non-refusable events within those sets. Varying the views on processes gives different degrees of abstraction and varies the capability of an observer to tell processes apart.

Views on a process might be varied by shifting the boundary between refusable and non-refusable events, at high and low-levels; and by shifting the boundary between levels for refusable and for non-refusable events.

As the boundaries are shifted and the partition function p changes, the equivalence relation $\equiv_{\mathsf{must}_\mathsf{p}}$ changes accordingly. The four possible locations of an event of P's interface may be ordered as in Figure 4, where higher positions for events result in *weaker* equivalence relations: those more abstract relations which identify more processes. Thus as previously observed, if all events are in LR_p then the equivalence is strongest, and in fact is equivalent to standard must testing; and if all events are in HN_p (or in fact a combination of HN_p and HR_p) then the equivalence is weakest, able only to identify the possibility of divergence in a process.

Each order relation in Figure 4 is associated with a lemma which supports the claim that transferring events from the lower to the higher set preserves $\equiv_{\mathsf{must}_\mathsf{p}}$. Each edge in general corresponds to a strict weakening of $\equiv_{\mathsf{must}_\mathsf{p}}$: new equivalences are introduced in each case. The resulting Corollary 3.18 collects these results together: if p is pointwise weaker than p', then must_p equivalence implies $\mathsf{must}_{p'}$ equivalence.

The first lemma states that increasing HN at the expense of HR preserves must testing equivalence.

LEMMA 3.14 If $LN_{p'} = LN_p$, $LR_{p'} = LR_p$, and $HN_{p'} \supseteq HN_p$, then $\equiv_{\mathsf{must}_{\mathsf{p}'}}$ is weaker than $\equiv_{\mathsf{must}_\mathsf{p}}$. □

Proof If $P \equiv_{\mathsf{must}_\mathsf{p}} Q$ then no test can distinguish them. In the context of p' there are fewer complete test executions (since fewer satisfy clause 2 of

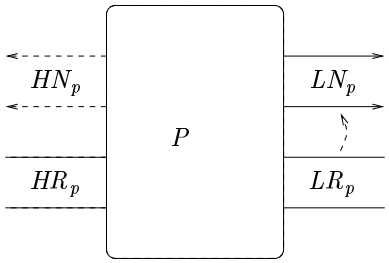

Figure 5: Low-level refusable events become non-refusable

Definition 3.9 for complete executions), but the set of possible tests remain as before, so no test will be able to distinguish P from Q within p'. □

New equivalences are introduced in general: for example, if $h \in HR$ then $h \rightarrow l \rightarrow STOP$ and $l \rightarrow STOP$ are distinguished; but if $h \in HN$, then they become equal.

However, if $HR \cup HN = \Sigma$, then no new equivalences are introduced, since $\equiv_{\mathsf{must_p}}$ will be the weakest it can be, only able to distinguish non-divergent processes from possibly-divergent ones.

The next lemma states that increasing LN at the expense of LR preserves must testing equivalence. This scenario is illustrated in Figure 5.

LEMMA 3.15 If $HN_p = HN_{p'}$, $HR_p = HR_{p'}$, and $LN_{p'} \supseteq LN_p$, then $\equiv_{\mathsf{must_{p'}}}$ is weaker than $\equiv_{\mathsf{must_p}}$. □

Proof Any test T which distinguishes P from Q in the context of p' will also serve as a test to distinguish P from Q in the context of p. Thus if $P \equiv_{\mathsf{must_p}} Q$ then $P \equiv_{\mathsf{must_{p'}}} Q$. □

Increasing LN introduces new equivalences in general. For example, if l_1 and l_2 are in LR, then the processes

$$\begin{aligned} P_1 &= l_1 \rightarrow STOP \,\Box\, l_2 \rightarrow STOP \\ P_2 &= (l_1 \rightarrow STOP \,\Box\, l_2 \rightarrow STOP) \sqcap l_1 \rightarrow STOP \end{aligned}$$

are distinguished by the test $T = l_2 \rightarrow \omega \rightarrow STOP$.

However, if l_1 and l_2 are contained in LN, then there is no test that can distinguish P_1 from P_2, and they are therefore must equivalent[3].

He The next lemma states that increasing HR at the expense of LR preserves must testing equivalence. This scenario is illustrated in Figure 6.

LEMMA 3.16 If $HN_p = HN_{p'}$, $LN_p = LN_{p'}$, and $HR_{p'} \supseteq HR_p$, then $\equiv_{\mathsf{must_{p'}}}$ is weaker than $\equiv_{\mathsf{must_p}}$. □

[3] T will not do, since it is not always willing to accept l_1 and l_2

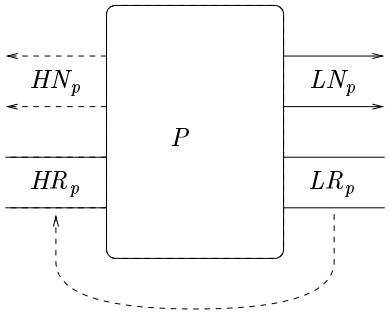

Figure 6: Abstracting refusable events

Proof If T is a test which distinguishes P from Q in the context of p', then $T \;|||\; CHAOS_{HR_{p'} \setminus HR_p}$ will serve as a test to distinguish P from Q in the context of S. Thus if $P \equiv_{\mathsf{must}_p} Q$ then $P \equiv_{\mathsf{must}_{p'}} Q$. □

New equivalences can be introduced by this step. The two processes $l_1 \rightarrow STOP$ and $l_2 \rightarrow STOP$ are easily distinguished when l_1 and l_2 are in LR, but if l_1 and l_2 are contained in HR then the processes are indistinguishable: there is no activity at the low-level interface at all.

The final lemma states that increasing HN at the expense of LN preserves must testing equivalence.

LEMMA 3.17 If $HR_p = HR_{p'}$, $LR_p = LR_{p'}$, and $HN_{p'} \supseteq HN_p$, then $\equiv_{\mathsf{must}_{p'}}$ is weaker than $\equiv_{\mathsf{must}_p}$. □

Proof If T is a test which distinguishes P from Q in the context of p', then $T \;|||\; RUN_{HR_{p'}} x \setminus HR_p$ will serve as a test to distinguish P from Q in the context of p. Thus if $P \equiv_{\mathsf{must}_p} Q$ then $P \equiv_{\mathsf{must}_{p'}} Q$. □

This step also introduces new equivalences. The example used with the previous lemma will also serve here: if l_1 and l_2 are contained in LN then $l_1 \rightarrow STOP$ and $l_2 \rightarrow STOP$ are easily distinguished, but if l_1 and l_2 become part of H_p then the processes are indistinguishable with regard to any test.

COROLLARY 3.18 If $(\forall\, a : \Sigma . p(a) \leqslant p'(a))$ then for any P and Q, if $P \equiv_{\mathsf{must}_p} Q$ then $P \equiv_{\mathsf{must}_{p'}} Q$. □

If partition p is pointwise less than partition p', then $\equiv_{\mathsf{must}_p}$ is stronger than $\equiv_{\mathsf{must}_{p'}}$.

The relation $p \leqslant p'$ (defined pointwise) can also be characterised in terms of the corresponding sets. In this case,

$$\begin{aligned} p \leqslant p' \Leftrightarrow \quad & LR_p \subseteq LR_{p'} \\ & LR_p \cup LN_p \subseteq LR_{p'} \cup LN_{p'} \\ & LR_p \cup HR_p \subseteq LR_{p'} \cup HR_{p'} \end{aligned}$$

high-level non-refusable events are hidden from any interacting process, and they are urgent, so they behave as hidden events. This is made explicit in the following lemma:

LEMMA 3.19 For any process P, $P \equiv_{\mathsf{must_p}} (P \setminus HN_p)$ □

Proof Suppose there is an unsuccessful complete execution of P in parallel with some test. Then $P \setminus HN_p$ can engage in the same execution, interacting on the same low-level events and performing the same high-level events (though they are now labelled with τ actions). This execution will also be complete for $P \setminus HN_p$. Conversely, suppose there is an unsuccessful complete execution of $P \setminus HN_p$ in parallel with a test T. Then P can step through the same execution, and it will be complete for P as well. Hence P and $P \setminus HN_p$ may fail exactly the same tests, so they must pass exactly the same tests. □

3.4 Examples

EXAMPLE 3.20

$$h_n \rightarrow l_r \rightarrow STOP \quad \equiv_{\mathsf{must}} \quad l_r \rightarrow STOP$$

The non-refusability of the high-level event means that at the low level the l_n event is guaranteed to be offered.

In contrast, a refusable high-level event yields the following:

$$h_r \rightarrow l_r \rightarrow STOP \quad \equiv_{\mathsf{must}} \quad l_r \rightarrow STOP \sqcap STOP$$

At the low level, the event l_r might never be offered, since the high-level event could be refused in a complete execution. In this case, the low-level behaviour is described by $STOP$. However, the high-level event could also be performed, and so the possibility of the low-level event is also present. □

EXAMPLE 3.21

$$\begin{array}{lll} h_{n1} \rightarrow l_{r1} \rightarrow STOP & & l_{r1} \rightarrow STOP \\ \Box & \equiv_{\mathsf{must_p}} & \sqcap \\ h_{n2} \rightarrow l_{r2} \rightarrow STOP & & l_{r2} \rightarrow STOP \\ & & h_{n2} \rightarrow l_{r1} \rightarrow STOP \\ & \equiv_{\mathsf{must_p}} & \Box \\ & & h_{n1} \rightarrow l_{r2} \rightarrow STOP \end{array}$$

In this example, the high-level events are non-refusable, and so one of them is guaranteed to occur in a complete execution. However, the testing process has no control over which will occur, and so the process is equivalent to one which offers a nondeterministic choice between the two low-level events. An observer who can engage only in low-level events cannot distinguish these three processes.

Observe that there is information flow from high to low in the two processes that have high-level events: the identity of the low-level event that occurs allows the identity of the preceding high-level event to be deduced. □

EXAMPLE 3.22 The 'refusal' of a low-level non-refusable event can still be detected by some tests, even if the traces are exactly the same. For example,

$$l_n \rightarrow STOP \quad \not\equiv_{\mathsf{must}} \quad STOP \sqcap l_n \rightarrow STOP$$

since the test $T = l_n \rightarrow \omega \rightarrow RUN_{l_n}$ is guaranteed to succeed on the left-hand process, but not on the right-hand one.

However, if some other non-refusable event is possible, then it masks the distinction:

$$\begin{array}{rcl} l_n \rightarrow STOP & \equiv_{\mathsf{must}} & (l_n \rightarrow STOP \sqcap STOP) \\ \Box\ l_{n2} \rightarrow STOP & & \Box\ l_{n2} \rightarrow STOP) \end{array}$$

There is no LN-test that can distinguish these two processes, as any such test must initially offer l_{n2} alongside l_n, and so every complete execution must contain some low-level event.

On the other hand, if only refusable event are possible alongside l_n, then the distinction can still be made:

$$\begin{array}{rcl} l_n \rightarrow STOP & \not\equiv_{\mathsf{must}} & (l_n \rightarrow STOP \sqcap STOP) \\ \Box\ l_r \rightarrow STOP & & \Box\ l_r \rightarrow STOP) \end{array}$$

The same test T as above will distinguish these processes. □

EXAMPLE 3.23 Two processes which are equivalent through low-level views $LR1$ and $LR2$ individually need not be equivalent through those views together—$LR1 \cup LR2$. For example, define the processes

$$\begin{array}{rcl} P_1 & = & (STOP \sqcap a \rightarrow STOP) \,|||\, (STOP \sqcap b \rightarrow STOP) \\ P_2 & = & STOP \sqcap (a \rightarrow STOP \,|||\, b \rightarrow STOP) \end{array}$$

If $p_1(a) = lr, p_1(b) = hr$ and $p_2(a) = hr, p_2(b) = lr$, then

$$\begin{array}{rcl} P_1 & \equiv_{\mathsf{must}\ p_1} & P_2 \\ P_1 & \equiv_{\mathsf{must}\ p_2} & P_2 \end{array}$$

The two processes are indistinguishable when only a, or only b, is available to be tested.

However, if both a and b are low-level events, then there is a test that distinguishes P_1 from P_2:

$$T \quad = \quad ((a \rightarrow b \rightarrow \omega \rightarrow STOP) \,\Box\, c \rightarrow \omega \rightarrow STOP) \setminus c$$

It is possible for P_1 to fail this test, by performing a but then failing to engage in b; whereas P_2 cannot do this, since if it can perform a then it can also perform b.

In this example, the occurrences of a and b in P_1 are completely independent of each other, and the performance of one provides no information about the performance of the other. On the other hand, in P_2 the a and the b are either both available or both unavailable, and so the occurrence of one does provide information about the availability of the other. □

3.5 A note on refusal testing

If the tests had the ability to carry out *refusal testing* (which a timed framework, for example, would give them) then the distinction between LN_p and LR_p is again significant. Refusal testing would be possible only on those events in LR_p, and so moving them to LN_p would lose some distinctions that could be made and introduce some new equivalences.

The traditional refusal testing example illustrates the point:

$$\begin{aligned} V1 &= coin \rightarrow STOP \sqcap (coffee \rightarrow STOP \,\Box\, tea \rightarrow STOP) \\ V2 &= (coin \rightarrow STOP \sqcap coffee \rightarrow STOP) \,\Box\, (coin \rightarrow STOP \sqcap tea \rightarrow STOP) \end{aligned}$$

Two vending machines can either return a coin, or else provide tea or coffee. These are indistinguishable under must testing. However, $V2$ can refuse *tea* and then provide *coffee*, a behaviour that is not possible for $V1$. In our framework there is no test that can make this distinction.

If *tea* and *coffee* are in LN_p, then these will be considered equivalent in a refusal testing framework because the tests will be required to be available on both of these events and so unable to test them for refusal. However, if either of these events is in LR_p then it could be tested for refusal, and thus distinguish the processes. For example, if *tea* is in LR_p then the test that fails if *tea* is refused and *coffee* subsequently performed, but succeeds otherwise, will distinguish the two processes.

Refinement

The operational characterisation of must testing equivalence also generalises in the usual way to must_p testing refinement. Essentially, Q will be a refinement of P if any test T that P must succeed must also allow Q to succeed:

DEFINITION 3.24

$$P \sqsubseteq_{\mathsf{must}_p} Q \quad \Leftrightarrow \quad \forall\, T : TEST_{LN_p} \bullet P \;\mathsf{must}_p\; T \Rightarrow Q \;\mathsf{must}_p\; T$$

□

In this case, if there is any context in which P is guaranteed to behave appropriately, then Q must also be guaranteed to behave appropriately. In the case of must testing, the process description P encapsulates the behaviour that should be guaranteed, in contrast to the world of may testing where P simply describes what is permitted.

4 Denotational characterisations

It is useful to have a characterisation in terms of the denotational semantics for when two processes are must_p equivalent, and when they are may_p equivalent. This allows model-checkers such as FDR to be deployed in analysing processes for such equivalences.

4.1 May equivalence

With regard to may testing, two processes will be considered equivalent with regard to a particular low-level view if their trace sets are identical when projected onto that view.

THEOREM 4.1

$$P \equiv_{\mathsf{may_p}} Q \quad \Leftrightarrow \quad traces(P \setminus H_p) = traces(Q \setminus H_p)$$

□

Proof (sketch) If there is a test that distinguishes P from Q, then this should lead to a trace of $P \setminus H_p$ that is not in $Q \setminus H_p$ (or vice versa). And if there is a trace of one but not the other, then this should lead to a test which can make the distinction. □

4.2 Must equivalence

Encapsulating must testing equivalence in the most general case is not straightforward. There are two independent issues to be resolved: one concerning the appropriate way to handle the high-level events; and the other concerning how best to treat the low-level non-refusable events.

Lemma 3.19 indicates that non-refusable high-level events can simply be hidden. high-level refusable events should also be removed from view, but in a way which does not make them urgent (since they are not required to occur). This can be achieved by running the process in parallel with a process which might block such high-level events at any stage; and then hiding all the high-level events. This is the approach taken in [GH97], by means of a 'regulator' process.

Case 1: $LN = \varnothing$

We will begin by considering the set of low-level non-refusable events to be empty. As before, high-level urgent events can simply be hidden

THEOREM 4.2 If $LN = \varnothing$ then

$$\begin{aligned} P \equiv_{\mathsf{must_p}} Q \quad \Leftrightarrow \quad & FDI((P \,|[\, HR_p \,]|\, CHAOS_{HR_p}) \setminus H_p) \\ & = FDI((Q \,|[\, HR_p \,]|\, CHAOS_{HR_p}) \setminus H_p) \end{aligned}$$

□

The construction $(P \,|[\, HR_p \,]|\, CHAOS_{HR_p}) \setminus H_p$ will be abbreviated $abs_p(P)$: an abstracted view of P.

Proof If $P \not\equiv_{\mathsf{must_p}} Q$ then there is some test T which distinguishes P from Q. In this case, there is (without loss of generality) an unsuccessful complete

execution e of P in parallel with T, whereas all complete executions of Q in parallel with T are successful. The execution e consists of a sequence of high and low-level transition steps, where the low-level transitions are in synchronisation with T and the high-level steps are independent. If the execution contains an infinite number of low-level events, then Q cannot not exhibit the same sequence of low-level events or else Q could take T through the same sequence of unsuccessful states. Hence there is an infinite trace of P that is not an infinite trace of Q, so their FDI semantics are different. Otherwise, the execution contains a finite number of low-level events. If it contains an infinite number of high-level events, then it will correspond to a divergence of $abs_p(P)$. If $abs_p(Q)$ has the same divergence, then it can also take T through an unsuccessful complete execution (the same one or one which diverges earlier), which contradicts the assumption that Q must_p T. Hence $abs_p(Q)$ has a different FDI semantics.

Finally, if the complete unsuccessful execution has a finite number of both high-level and low-level events, then it must end in a state where no internal events of events from HR are possible. This corresponds to an execution of $P \,|[\, HR \,]|\, CHAOS_{HR}$ which ends in a state where no internal events, or internal events from HR or HN are possible, which in turn corresponds to a failure of $abs_p(P)$ (since the final state is stable). If $abs_p(Q)$ exhibits the same failure, then it can engage in a similar complete execution which takes T through the same states, and hence is also unsuccessful. But this contradicts the assumption that Q must_p T. Hence there is a failure which distinguishes $abs_p(P)$ from $abs_p(Q)$.

Hence in all cases, if $P \not\equiv_{\mathsf{must}_p} Q$ then $FDI(abs_p(P)) \neq FDI(abs_p(Q))$.

Conversely, suppose that $FDI(abs_p(P)) \neq FDI(abs_p(Q))$. Then there is some behaviour that distinguishes them. In this case, a test can be constructed which distinguishes P from Q.

If $abs_p(P)$ and $abs_p(Q)$ differ on a divergence $tr = \langle l_1 \ldots l_n \rangle$ (assume without loss of generality that $tr \in divergences(P) \setminus divergences(Q)$) then the test $T = T_1$ will make the distinction, where

$$T_i \quad = \quad \begin{array}{ll} (l_i \rightarrow T_{i+1}) \,\Box\, c \rightarrow \omega \rightarrow STOP) \setminus c & \text{if } i < n+1 \\ (c \rightarrow \omega \rightarrow STOP) \setminus c & \text{if } i \geqslant n+1 \end{array}$$

An execution of a process P in parallel with this test can fail only if there is an execution of P whose projection onto L is tr, and which is either divergent itself or else able to perform infinitely many events from HN—precisely the conditions for tr to be a divergence of $abs_p(P)$. Thus Q must_p T but $\neg(P$ must_p $T)$.

If $abs_p(P)$ and $abs_p(Q)$ have identical divergences, but differ on an infinite trace $u = \langle l_1, l_2, \ldots \rangle$ (assume without loss of generality that $u \in infinites(abs_p(P)) \setminus infinites(abs_p(Q))$, then the test $T = T_1$ using the definition above (with $n = \infty$) will make the distinction.

Finally $abs_p(P)$ and $abs_p(Q)$ agree on their divergences and infinite traces, but differ on some failure (tr, X) where $tr = \langle l_1, \ldots, l_n \rangle$ (assume without loss of generality that $(tr, X) \in failures(abs_p(P)) \setminus failures(abs_p(Q))$, then the test

$T = T_1$ as defined below will make the distinction.

$$\begin{array}{lll} T_i & = & (l_i \rightarrow T_{i+1}) \,\Box\, c \rightarrow \omega \rightarrow STOP) \setminus c \quad \text{if } i < n+1 \\ & & (x : X \rightarrow \omega \rightarrow STOP) \qquad\qquad\quad \text{if } i \geqslant n+1 \end{array}$$

P can fail this test only if there is some execution whose events tr_0 projected onto L provides tr, such that after executing tr_0 none of the events in $X \cup HN$ are possible. But this is possible precisely when $(tr, X) \in \mathit{failures}(\mathit{abs}_p(P))$. Hence this test distinguishes P from Q. □

This gives us a low-level view of a process: the low-level view of P is simply $(\mathit{abs}_p(P))$. This only has low-level events. Any two processes which exhibit this low-level behaviour are indistinguishable through that view of the process.

Case 2: $LN \neq \varnothing$

If the set of low-level non-refusable events is not empty, then there are some constraints on the low-level behaviour of the tests.

Subcase 1: $H = \varnothing$

To begin with, we will consider the situation where there are no high-level events at all: every event is low-level, some are refusable and some are not.

Any *finite* complete execution must end up in a state in which all LN events are impossible for the process. This must correspond to a failure (tr, X) of the process for which $LN \subseteq X$. The events in LN which were performed should appear in the trace, since they were accessible to the tests. This corresponds to treating the events as urgent but visible, which is not as aspect of standard CSP, but which has been analysed in the context of timed CSP [DJS92]. The failures of such a process will be given as the urgent failures (with respect to LN), defined as

$$\mathit{ufailures}_{LN}(P) \quad = \quad \{(tr, X) \in \mathit{failures}(P) \mid LN \subseteq X\}$$

Such a set does not meet the standard axioms for CSP, as Example 4.3 illustrates. In fact, it turns out that it need not meet *any* of the axioms pertaining to failures: it is not necessarily prefix-closed on traces, subset closed on refusals, or even non-empty, and events need not extend either a trace or a refusal set.

EXAMPLE 4.3 If $H = \varnothing$, $LN = \{ln\}$, and $LR = \{lr\}$, then

$$\mathit{ufailures}_{LN}(lr \rightarrow ln \rightarrow STOP) \quad = \quad \{(\langle\rangle, \{ln\}), (\langle ln\rangle, \{ln\}), (\langle ln\rangle, \{ln, lr\})\}$$

This set is neither prefix-closed on traces, subset closed on refusals, and on the behaviour with the empty trace the event nr can extend neither the trace (by itself) or the refusal.

A recursive process that is always willing to perform events from LN, such as $P = ln \rightarrow P$, has an empty urgent failure set. However, it is guaranteed to have infinite traces corresponding to the sequences of events from LN that it must be able to perform. □

Along with the divergences and infinite traces, the urgent failures set does characterise must testing equivalence with regard to non-refusable low-level events:

THEOREM 4.4 If $H = \varnothing$ then

$$\begin{aligned} P \equiv_{\mathsf{must}_p} Q \quad \Leftrightarrow \quad & \mathit{ufailures}_{LN}(P) = \mathit{ufailures}_{LN}(Q) \\ & \wedge\ \mathit{divergences}(P) = \mathit{divergences}(Q) \\ & \wedge\ \mathit{infinites}(P) = \mathit{infinites}(Q) \end{aligned}$$

□

We will refer to this triple of semantic sets of P (with respect to the partition p) as $UDI_p(P)$.

Proof If P and Q are different under testing, then the test that distinguishes them must expose some difference in one of these sets. An infinite complete execution exposes a difference in the divergences or the failures, as in the proof of Theorem 4.2. A finite complete execution exposes a difference in the failures, but a failure in which LN is refused; so the *ufailures* provide sufficient information.

If P and Q are different on one of these sets, then there is some test that can be constructed which distinguishes them by failing only on that particular behaviour. If they differ on divergences or infinite traces, then a test can be constructed as in the proof of Theorem 4.2. If they differ on *ufailures* then a test can also be constructed as before, introducing at every state an extra choice, of every event in LN to the success state. □

A new CSP operator $\mathsf{sticky}_{LN}\ (P)$ can be defined which provides a context for characterising urgent failures: $\mathit{FDI}(\ \mathsf{sticky}_{(}\ P)) = \mathit{FDI}(\ \mathsf{sticky}_{(}\ Q))$ if and only P and Q have the same urgent failures, divergences, and traces. As we shall see, this can be defined in terms of standard operators, enabling model-checking within FDR.

The sticky operator masks non-urgent failures on a set LN by introducing as many refusals as possible whenever an event from LN is possible. In particular, whenever an event l from LN is possible, then it introduces the possibility that all other events should be refused, and that l is the only possible event. This has a similar effect to making the events in LN urgent, since it is the process P itself (and not its environment) that chooses the event to be performed. In order to be consistent with the axioms of the FDI model, such events cannot be forced to occur (since the process might be in an uncooperative environment), but once P has selected an event, it will then refuse all other events.

It is defined denotationally as follows:

$$\begin{aligned} \mathit{divergences}(\ \mathsf{sticky}_{LN}\ (P)) &= \mathit{divergences}(P) \\ \mathit{infinites}(\ \mathsf{sticky}_{LN}\ (P)) &= \mathit{infinites}(P) \\ \mathit{failures}(\ \mathsf{sticky}_{LN}\ (P)) &= \mathit{failures}(P) \\ & \quad \cup \{(tr, X) \mid (tr \frown l, \varnothing) \in \mathit{failures}(P) \\ & \qquad \wedge\ l \in LN \wedge l \notin X\} \end{aligned}$$

Observe that $\mathsf{sticky}_{LN}\ (P)$ has the same traces as P. It is only additional refusals that are introduced into the failure set.

Theorem 4.4 given above can be characterised in this form:

THEOREM 4.5 If $H = \varnothing$ then

$$P \equiv_{\mathsf{must_p}} Q \quad \Leftrightarrow \quad \mathsf{sticky}_{LN}\ (P) = \mathsf{sticky}_{LN}\ (Q)$$

□

Different processes might map to the same result under sticky_{LN} . For example, $P_1 = ln_1 \rightarrow STOP \ \Box\ ln_2 \rightarrow STOP$ and $P_2 = ln_1 \rightarrow STOP \sqcap ln_2 \rightarrow STOP$ have different refusals, yet $\mathsf{sticky}_{LN}\ (P_1)$ and $\mathsf{sticky}_{LN}\ (P_2)$ have the same refusals. Hence from Theorem 4.4 they are equivalent under must testing.

This new operator can also be given an operational semantics, which may provide an alternative understanding of its behaviour. The process $\mathsf{sticky}_{LN}\ (P)$ will have all the transitions that P has together with a few extra ones introduced to allow events from LN to be 'selected'. Two rules define its operational semantics:

$$\frac{P \xrightarrow{\mu} P'}{\mathsf{sticky}_{LN}\ (P) \xrightarrow{\mu} \mathsf{sticky}_{LN}\ (P')}$$

$$\frac{P \xrightarrow{a} P'}{\mathsf{sticky}_{LN}\ (P) \xrightarrow{\tau} (a \rightarrow \mathsf{sticky}_{LN}\ (P'))} \quad [\ a \in LN\ ]$$

The transitions that are introduced by means of the second rule correspond to the additional failures that are introduced to $\mathsf{sticky}_{LN}\ (P)$.

If the events in LN are hidden, it makes no difference whether they are abstracted by means of the sticky operator first:

$$P \setminus LN \quad = \quad (\ \mathsf{sticky}_{LN}\ (P)) \setminus LN$$

EXAMPLE 4.6 Let $LN = \{l1, l2\}$. The operator $\mathsf{sticky}_{LN}\ (P)$ has the effect of allowing the process to choose to perform events in LN, or at least to block further progress until the chosen event is performed. Thus an external choice becomes internal:

$$\mathsf{sticky}_{LN}\ (l1 \rightarrow STOP \ \Box\ l2 \rightarrow STOP) \quad = \quad l1 \rightarrow STOP \sqcap l2 \rightarrow STOP$$

A choice between events in LN and other events allows the process to resolve the choice in favour of the LN events, but not against them:

$$\begin{aligned} \mathsf{sticky}_{LN}\ (l1 \rightarrow STOP \ \Box\ h1 \rightarrow STOP) \quad &= \quad l1 \rightarrow STOP \\ &\quad \sqcap\ (l1 \rightarrow STOP \ \Box\ h1 \rightarrow STOP) \end{aligned}$$

This is sometimes (see [Ros97]) written as $(h1 \rightarrow STOP) \triangleright (l1 \rightarrow STOP)$ □

The operational semantics for sticky point the way to a definition in terms of the standard CSP operators[4]. This can be achieved as follows: firstly, let f_{old} and g be alphabet renaming functions such that $f_{old}(a) = (old, a)$ for all $a \in \Sigma$, and $g(old, a) = g(new, a) = a$ for all $a \in \Sigma$, with g leaving events not of the form (a, new) or (a, old) unchanged. Define the process R as follows:

$$\begin{aligned} R_{LN} \quad = \quad & (old, a) : f(LN) \rightarrow (new, a) \rightarrow R_{LN} \\ & \Box\ (old, a) : (f(\Sigma) \setminus f(LN)) \rightarrow R_{LN} \end{aligned}$$

This process allows all events of the form (old, a), but whenever the event a is in LN, then it must perform the event (new, a) before any further events. sticky can then be defined as follows:

$$\mathsf{sticky}_{LN}\ (P) \quad = \quad g((f(P)\,|[\,f(\Sigma)\,]|\,R) \setminus f(LN))$$

Any sticky event $a \in LN$ of P is performed internally in this process (as (old, a)), but P is prevented from any further progress by R until (new, a) occurs (which appears in the overall process as the original sticky event a because of the renaming g). Non-sticky events are performed as expected.

Subcase 2 : $H \neq \varnothing$

Finally, in the most general case, we arrive at the following theorem:

THEOREM 4.7

$$P \ \equiv_{\mathsf{must_p}}\ Q \quad \Leftrightarrow \quad \mathsf{sticky}_{LN}\ (abs_p(P)) = \ \mathsf{sticky}_{LN}\ (abs_p(Q))$$

□

Observe that $\mathsf{sticky}_{LN}\ (abs_p(P)) = abs_p(\ \mathsf{sticky}_{LN}\ (P))$. The order in which the abstractions are performed is irrelevant.

4.3 Congruence

The equivalences considered in this paper are not congruences in general, which is perhaps why they are not generally considered in the literature on testing. This fact is unsurprising, since operators can influence the behaviour of a process through its abstracted interface, and if processes differ there then they may be affected differently.

EXAMPLE 4.8

$$h_{r1} \rightarrow l \rightarrow STOP \quad \equiv_{\mathsf{must_p}} \quad h_{r2} \rightarrow l \rightarrow STOP$$

but

$$STOP\,|[\,h_{r1}\,]|\,h_{r1} \rightarrow l \rightarrow STOP \quad \not\equiv_{\mathsf{must_p}} \quad STOP\,|[\,h_{r1}\,]|\,h_{r2} \rightarrow l \rightarrow STOP$$

[4] I am grateful to Bill Roscoe for this observation

since the processes behave differently on the abstracted event hr_1, they can behave differently when placed in parallel with a process that interacts with them on that event.

Another example concerns the event renaming where $f(h_{r1}) = l_1$, $f(h_{r2}) = h_{r2}$, and $f(l) = l$. In this case

$$f(h_{r1} \rightarrow l \rightarrow STOP) \quad \not\equiv_{\mathsf{must_p}} \quad f(h_{r2} \rightarrow l \rightarrow STOP)$$

Finally, an event renaming that renames a high-level refusable event to a high-level non-refusable event, but does not map high to low or low to high: $f(h_{r1}) = h_{n1}$, $f(h_{r2}) = h_{r2}$, and $f(l) = l$. In this case

$$f(h_{r1} \rightarrow l \rightarrow STOP) \quad \not\equiv_{\mathsf{must_p}} \quad f(h_{r2} \rightarrow l \rightarrow STOP)$$

The left hand process cannot refuse l at the low level, whereas the right hand process can.

The first two examples also illustrate may equivalences that are not preserved; the last example does preserve may equivalence. □

However, many of the operators do preserve both equivalences. Prefixing, sequential composition (provided ✓ is low-level), all forms of choice, and hiding certainly do so. Parallel composition does so, provided all synchronisations are at the low level: thus interleaving preserves equivalence, as does the operator $|[A]|$ provided $A \subseteq L$. Event renaming $f(P)$ preserves $\equiv_{\mathsf{must_p}}$ provided the partitions are respected: so $p(f(a)) = p(a)$ for all events a is required for $\equiv_{\mathsf{must_p}}$. With $\equiv_{\mathsf{may_p}}$, we require simply that high-level events do not become low-level, so $a \in H \Rightarrow f(a) \in H$ is sufficient to guarantee preservation of equality.

4.4 Refinement

Both may and must refinement are characterised in the denotational framework in the expected way:

$$P \sqsubseteq_{\mathsf{may}_p} Q \quad \Leftrightarrow \quad traces(Q \setminus H_p) \subseteq traces(P \setminus H_p)$$

$$P \sqsubseteq_{\mathsf{must}_p} Q \quad \Leftrightarrow \quad FDI(\ \mathsf{sticky}_{LN}\ (abs_p(Q))) \subseteq FDI(\ \mathsf{sticky}_{LN}\ (abs_p(P)))$$

where $\subseteq$ on FDI semantics is defined pointwise on each of the three components: failures, divergences, and infinite traces.

The operational characterisation of refinement is thus equivalent to the denotational characterisations.

5 Non-interference

Non-interference properties are generally considered in the context of a given system. The requirement is that even if an agent knows exactly how the system

works, there is no information flow across particular boundaries concerned with particular activity on that system. In other words, the options available to the low-level user should not divulge any information about the high-level users' activity. High-level users' activity is concerned with events in HR; the set HN consists of those high-level events entirely within the control of the system. Knowing that the system will perform an event of HN does not leak information about high-level activity, since the high-level user is unable to prevent it.

Information flow from high to low will be prevented if P's low-level possibilities at any stage are dependent entirely on P's previous low-level behaviour, and not in terms of any high-level behaviour. This would seem to indicate that if two sequences of events have been performed, which appear the same on the low level, then the resulting processes should be equivalent.

This characterisation can be made in various ways. It has traditionally been made denotationally, and this has led to some difficulties to its relationship with refinement. If e is an execution of a process P, then $Ltrace_p(e)$ is the sequence of low-level (in the sense of the partition p) events in e.

Operationally, lack of information flow in P from high to low level might be characterised as follows:

DEFINITION 5.1 A process P is *interference-free* with respect to p if

$$P \stackrel{e'}{\Longrightarrow} P' \wedge P \stackrel{e''}{\Longrightarrow} P'' \wedge Ltrace_p(e') = Ltrace_p(e'') \Rightarrow P' \equiv_{\mathsf{must}_\mathsf{p}} P''$$

□

where $Ltrace_p(e)$ is the sequence of low-level (in the sense of the partition p) events in e. If the low-level views of two executions are the same, then the resulting processes must be indistinguishable.

In this case, we can say that P is *interference-free on* p. This is a very strong definition: it excludes nondeterministic processes, even those that can perform no high-level events, such as $P = l_{r1} \rightarrow STOP \sqcap l_{r2} \rightarrow STOP$: the distinguishable processes $l_{r1} \rightarrow STOP$ and $l_{r2} \rightarrow STOP$ are both reachable via the empty trace. In fact in such cases, a refinement of this system (with respect to the low-level view) might introduce high-level events and lose interference-freedom as a result. For example, the process $Q = h_{r1} \rightarrow l_{r1} \rightarrow STOP \,\Box\, l_{r2} \rightarrow STOP$ is actually equivalent to P described above, with respect to the low-level interface, and yet Q leaks information from high to low.

This operational definition is attractive in one sense, since it considers individually all possible processes resulting from the execution, rather than considering them all together (where the acceptable behaviour of some can mask the unacceptable behaviour of others), as is the case with the denotational 'after' operator.

This definition has the difficulty that it is dependent on the precise nature of the operational semantics for a process, and two processes that are equivalent (under must testing for example) might be treated differently by the definition. This means that it cannot be characterised denotationally, and that in general its truth or falsity is not determined from the denotational semantics.

Divergence

To take an extreme example, the process which has one state and is only able to perform internal actions to that one state may be defined recursively as follows: $\bot = \bot$. It is must equivalent to $\bot \sqcap LEAK$, where $LEAK$ is a process which takes in messages on a high-level channel, and immediately communicates them on a low-level channel: $LEAK = in_H?x \rightarrow out_L!x \rightarrow LEAK$. Yet $\bot$ meets the definition, whereas $\bot \sqcap LEAK$ does not.

The desire to have no information flow preserved by refinement has led to some difficulties with regard to this example. If $\bot$ is seen as a process which does not provide information flow, but it can be refined by $LEAK$, then it is patently clear that any definition of security with respect to information flow is either going to fail on $\bot$ or else will not be preserved by refinement.

Divergence-free processes

If the process is divergence-free, then the situation is rather better. In this case, the definition will hold of precisely those processes whose low-level behaviour is deterministic: that is, those processes P for which $abs_p(P)$ is deterministic. This coincides with Roscoe's characterisation of non-interference.

THEOREM 5.2 If P is divergence-free, then P is interference-free on p if and only if $abs_p(P)$ is deterministic. □

Proof (sketch) If $abs_p(P)$ is deterministic, then given any execution $P \stackrel{e}{\Longrightarrow} P'$, then the process P' reached after the execution must be a refinement (in the denotational sense) of the process $P \,/\, trace(e)$. This means that $abs_p(P')$ is a refinement of $abs_p(P \,/\, trace(e))$, which in turn is a refinement of $abs_p(P) \,/\, Ltrace_p(e)$. But this process is deterministic, since $abs_p(P)$ is, and so $abs_p(P')$ must be denotationally equivalent to $abs_p(P \,/\, tr)$ (since they are both refinements of the same deterministic process, and hence equivalent to it and thus to each other). This must be the case for any process P' reached after any execution corresponding on the low level to tr, and so all such processes are denotationally equivalent, and thus $\equiv_{\mathsf{must}_p}$ equivalent. Hence the definition will always apply to processes whose low level views are deterministic.

Conversely, if $abs_p(P)$ is not deterministic, then there is some low-level trace tr such that $tr \frown \langle l \rangle$ is a possible low-level trace, and $(tr, \{l\})$ is a possible low-level refusal set. Since P is not divergent, these behaviours correspond to particular executions of P. This means that there are executions e_1 and e_2 corresponding to tr such that $P \stackrel{e_1}{\Longrightarrow} P_1 \stackrel{l}{\longrightarrow}$ and $P \stackrel{e_2}{\Longrightarrow} P_2$ such that $P_2 \not\stackrel{\tau}{\longrightarrow}$ and $P_2 \not\stackrel{l}{\longrightarrow}$. In other words, P can reach a state P_1 from which l is possible, and can also reach a stable state P_2 from which l is not possible. Then the test

$$T \quad = \quad (x : LN \rightarrow \omega \rightarrow STOP \,\Box\, l \rightarrow RUN_{LN}) \,\Box\, (a \rightarrow \omega \rightarrow STOP) \setminus a$$

can fail only if l is initially possible. Thus $P_2 \mathsf{must}_p\, T$ but $\neg(P_1 \mathsf{must}_p\, T)$. Hence P can reach two states that are distinguishable by some low-level test. □

Observe that this theorem holds even if $LN \neq \varnothing$, despite the fact that tests cannot directly detect refusals of events in LN. This means that two processes might be indistinguishable under testing, yet the definition of non-interference applies differently to them. For example, consider the two processes:

$$\begin{aligned} P_1 &= h_1 \rightarrow l_{n1} \rightarrow P_1 \,\Box\, h_2 \rightarrow l_{n2} \rightarrow P_1 \\ P_2 &= h : \{h_1, h_2\} \rightarrow l : \{l_{n1}, l_{n2}\} \rightarrow P_2 \end{aligned}$$

If $LN = \{l_{n1}, l_{n2}\}$, then these two processes are indistinguishable at the low level. Any low-level test must always be ready to accept all events in LN, and so is able to make distinctions only on the basis of traces—and P_1 and P_2 have the same low-level traces.

On the other hand, it is clear that P_1 allows interference whereas P_2 does not, and in fact P_1 fails the definition given above whereas P_2 meets it: $abs_p(P_1)$ is non-deterministic whereas $abs_p(P_2)$ is deterministic.

This example also illustrates the point that interference-freedom is concerned with particular processes that are given. The aim in interference-freedom is not to distinguish P_1 from P_2, but rather to make deductions about high-level activity from the visible low-level activity in a given process.

6 Atomicity and fault-tolerance

Atomicity is a feature of particular kinds of specification, where the desired behaviour is characterised in terms of the occurrence or availability of a single event.

Typically in analysing fault tolerance, faults are modelled by the occurrence of special fault events. These might appear at certain points of a process' description, indicating that the fault can occur at that stage during the process' execution. They will then be modelled as refusable events—the environment might perform them when the process makes them 'available', but is not obliged to do so.

The low-level activity need not be deterministic, so this is different from consideration of information flow properties.

For example, a one-place buffer that can lose its contents on the occurrence of a particular fault might be described as follows:

$$\begin{aligned} FBUFF = in?x \rightarrow\ & out!x \rightarrow FBUFF \\ & \Box\ power\_blip \rightarrow FBUFF \\ \Box\ & power\_blip \rightarrow FBUFF \end{aligned}$$

On the other hand, fault recovery will generally be modelled as non-refusable events: the fault recovery mechanism is under the control of the process itself, and should not be blocked by the environment.

As a crude example, a system might undergo a fault between input and output, from which it must recover before performing output. This could be

modelled, extremely crudely, as

$$\begin{array}{lcl} FT & = & in?x \rightarrow \ out!x \rightarrow STOP \\ & & \quad \Box \ fault \rightarrow recover \rightarrow out!x \rightarrow STOP \end{array}$$

The requirement on the system might be that, when the fault recovery mechanisms are out of the view of the user, then this system, should look like a simple buffer taking an input to an output.

In other words, the *requirement* on the system is that it is a refinement under must_p testing to

$$SPEC = in?x \rightarrow out!x \rightarrow STOP$$

Here we have $p(fault) = hr$ and $p(recover) = hn$.

Then $SPEC \sqsubseteq_{\mathsf{must}_p} FT$.

If $p(fault) = hn$, then any complete execution would not be able to finish if a fault was possible—this is tantamount to *relying* on the fault to occur. In this case, $SPEC \equiv_{\mathsf{must}} FT$ again holds, but so too does the equivalence

$$in?x \rightarrow fault \rightarrow recover \rightarrow out!x \rightarrow STOP \quad \equiv_{\mathsf{must_p}} \quad FT$$

which relies on *fault* to occur in order to guarantee output.

Conversely, if $p(recover) = hr$ then recovery can be blocked. In this case correct behaviour cannot be guaranteed, and in fact

$$FT \quad \equiv_{\mathsf{must_p}} \quad in?x \rightarrow (STOP \sqcap out!x \rightarrow STOP)$$

The approach to fault-tolerant modelling suggested by this example is to treat fault events as high-level refusable events HR, to treat system recovery mechanisms as high-level non-refusable events HN, and to treat the normal part of the system, which the user interacts with, in terms of low-level events (either refusable or non-refusable as appropriate).

Bullet-proof write

For example, a memory cell may be specified by the following process description:

$$\begin{array}{lcl} MEM(v) & = & write?w \rightarrow MEM(w) \\ & & \Box \\ & & read!v \rightarrow MEM(v) \end{array}$$

Any user of this cell may either write a fresh value w to it, or else read its current contents v. A protocol which ensures that a write is successful by means of a sequence of internal events would be required to exhibit the same behaviour as this specification. The specification is concerned with atomicity in the sense that the internal behaviour should implement a single *write* event.

An implementation $MEMIMP(v)$ would be required to refine $MEM(v)$ when observed through the interface $\{write, read\}$:

$$MEM(v) \quad \sqsubseteq_{\mathsf{must}_p} \quad MEMIMP(v)$$

In this case, all the 'high'-level events would be internal system events and would therefore be urgent. The low-level events on *read* and *write* are under the control of the environment and are therefore in LR—the environment can refuse them. The sets LN and HR are empty. Since $MEM(v)$ is deterministic, any implementation must actually be equivalent to it.

In fault-tolerant systems, the nature and impact of the faults to be tolerated can have an effect on the specification. For example, an implementation may be required to cope with the possibility of power failures on volatile memory. Such a fault-tolerant design cannot hope to guarantee success in a write operation, since an update might become lost at the moment it is provided. In this case, the requirement will be that the memory cell itself should not be corrupted, and should remain with its previous value.

This weaker specification of a fault-tolerant memory cell $FTMEM$ is expressed as follows:

$$\begin{aligned} FTMEM(v) \quad = \quad & write?w \rightarrow (FTMEM(w) \sqcap FTMEM(v)) \\ & \Box \\ & read!v \rightarrow FTMEM(v) \end{aligned}$$

An implementation of this, such as the Bullet-Proof-Write of [GH97], will be required to satisfy

$$FTMEM(v) \quad \sqsubseteq_{\mathsf{must}_p} \quad BPW(v)$$

Much of the fault-tolerance will not be expressed directly as a CSP specification, but will rather be contained within the modelling of the system components which comprise the system BPW. The faults that are possible will be modelled explicitly as events which are possible for their components, and their impact will also be contained within the component description. The set of all fault events can be given as the set $FAULTS$. All such fault events will be considered as high-level refusable events, and so $FAULTS \subseteq HR$.

A memory cell implementation will be fault-tolerant in a particular sense if it can always recover from corruption in the presence of fault events which can cause corruption. On the other hand, it will not be fault-tolerant to the extent that it will sometimes fail to do the update. Fault-tolerance is generally with respect to particular faults, and coping with the most serious faults might result in a degraded service.

A liveness or correctness expectation would be that the memory should be updated when no fault events occur:

$$MEM(v) \quad \sqsubseteq_{\mathsf{must}_p} \quad STOP \,|[\, FAULTS \,]|\, BPW(v)$$

This will prevent a trivial implementation whereby no update is ever carried out.

Database transaction

We may wish to think of obtaining an answer from a database as an atomic transaction. But if we put in a request for an answer, and the database has to do some recovery first (e.g. if it becomes clear that there is some corruption of data) then this should be invisible to the user, and contained as part of the 'atomic' transaction of receiving an answer.

Here the specification would be $SPEC = query?x \rightarrow answer!f(x) \rightarrow SPEC$, and the requirement on any implementation IMP is that $SPEC \sqsubseteq_{\mathsf{must}_p} IMP$. In order to make this specification precise, it must be decided where each event appears in the interface. In this case, the *query* is under the control of the environment and hence will appear in LR, the *answer* can be under the control of the system and hence appear in LN, and all other events appear in H, with the fault events appearing in HR.

The user does not want or need to see all the internal activity, so from his point of view the system does behave in an atomic way. From the point of view of the 'tests' that the user can attempt, the operation *answer* is atomic.

7 Summary

This paper has been concerned with providing a more explicit approach to the kind of abstraction that is achieved when processes are viewed through particular interfaces, and where their events can be considered as refusable or not refusable by the environment of the process. The results have reinforced the denotational approach, provided a more explicit explanation and justification, and indeed have extended that approach by considering a more general categorisation of events.

This form of abstraction has been analysed for both may and must testing with respect to a partition p of the interface of the process, and the variation in the relations as the partition p varies has also been analysed: as less control over events is provided to the testing environment (either through removal from the interface, or through non-refusability), the equivalence relations become weaker.

The may testing equivalence and refinement relations turn out to be relatively straightforward to characterise in denotational terms, and are indeed equivalent to the denotational approaches which have been traditionally taken.

Must testing equivalence and refinement have also resulted in the expected equivalences as far as high-level, and refusable low-level events are concerned; but the relations when low-level non-refusable events are permitted have not been previously considered, and were tricky to characterise. At present it is still an open question whether the equivalence relation in this case can be characterised using the existing CSP language, or whether an extension would be required.

This approach to abstraction has been used in what appears to be an extremely strong operational characterisation of non-interference (or interference-freedom) which turns out to be equivalent to Roscoe's [RWW94] characterisation

on non-divergent processes. We have also considered its place in the specification of fault-tolerant systems, and in the characterisation of specifications that make use of atomicity.

Acknowledgements

I am grateful to Peter Ryan and Irfan Zakkiudin for discussion and comments on earlier forms of this work, and to Bill Roscoe for pointing out the CSP characterisation of sticky .

Support for this work was provided by DERA.

References

[DJS92] J Davies, D Jackson, and S Schneider. Making things happen in Timed CSP. In *Formal Techniques for Real-Time and Fault-Tolerant Systems*, volume 573 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[dNH87] R de Nicola and M Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1), 1987.

[GH97] M Goldsmith and J Hulance. Application of CSP and FDR to safety-critical systems: Investigation of refinement properties of fault tolerance and prototype implementation of analysis techniques. DERA project report, 1997.

[Hen88] M Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[Hoa85] C A R Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Ros97] A W Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[RWW94] A W Roscoe, J C P Woodcock, and L Wulf. Non-interference through determinism. In *European Symposium on Research in Computer Security*, volume 875 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[Sch98] S A Schneider. Concurrency and time, in preparation, 1998.