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 is given a new denotational characterisation encapsulated within the CSP denotational semantics. It informs, reinforces and extends the traditional denotational approach to abstraction.

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. The terminology of this paper and its underlying motivation is derived from this context.

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 into 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.

The main contribution is to generalise the process algebraic notion of abstraction in operational terms, and to provide an equivalent formulation within the failures model for CSP.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 738–757, 1999.

The structure of this paper is as follows: section 2 introduces the CSP notation; section 3 is concerned with the testing framework for equivalence of processes; section 4 is concerned with denotational characterisations of the testing viewpoint; sections 5 and 6 consider the application of this work to particular areas. The paper concludes with a discussion. Proofs of the theorems may be found in the full version of this paper [Sch99a] and are omitted here.

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, Sch99b].

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 subsequently to behave as P. The prefix choice process $x : A \rightarrow P(x)$ is initially prepared to engage in any event from the set A. If an event $x \in A$ is chosen by its environment then its subsequent behaviour is $P(x)$. The output $c!v \rightarrow P$ is able initially to perform only $c.v$, the output of v on channel c, after which it behaves as P. The input $c?x : T \rightarrow P(x)$ 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 $c?x \rightarrow P(x)$. 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 $a \rightarrow P$ can perform an a event and reach the process P. This is written $(a \rightarrow P) \xrightarrow{a} P$. The process $(a \rightarrow P) \setminus \{a\}$ can perform an internal event, denoted τ, and reach the process $P \setminus \{a\}$. This is written $(a \rightarrow P) \setminus \{a\} \xrightarrow{\tau} (P \setminus \{a\})$. For the full operational semantics of CSP, see [Ros97, Sch99b].

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 $\langle \mathit{in}.3, \mathit{out}.3, \mathit{in}.5\rangle$ which is a possible trace of the recursive process $COPY = \mathit{in}?x : T \rightarrow \mathit{out}!x \rightarrow COPY$. 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 $(\langle \mathit{in}.3, \mathit{out}.3, \mathit{in}.5\rangle, \{\mathit{out}.3, \mathit{out}.4\})$ which is a possible failure of $COPY$. The *divergences* of a process P, $\mathit{divergences}(P)$, are those traces which can lead to P performing 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, Sch99b]. In any such model a process P is refined by another process Q (written $P \sqsubseteq Q$) if the semantics of Q is contained in the semantics of P: anything that can be observed of Q can also be observed of P.

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:

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

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). We are concerned with understanding when processes are equivalent on such an interface. We will use the set H simply to 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 $L \cap H = \varnothing$. There is no reason in principle why L and H should cover Σ.

The set H is itself divided up into 'refusable' events HR and 'non-refusable' events HN, 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. A related view is to consider the set HN as those events whose occurrence is entirely under the control of the process P, whereas HR consists of those events which require cooperation from the environment. It should not be possible to deduce anything about a high level process interacting with P from the fact that a HN event must occur. Events that the *system* cannot refuse (such as inputs in an I/O automaton) can be incorporated into the description of P. If e is such an event, then it would be a requirement on P that e never appears in a refusal set.

L is also divided up into the disjoint sets LR and LN.

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

This partition[1] 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 subsets of Σ 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 function p and so will generally be subscripted with the p. In this paper, we follow the 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 function p.

We now define the notion of a LN_p test:

Definition 1. *a LN_p test T is a CSP process only able to perform events from $L_p \cup \{\omega\}$ which can never refuse any of the events in LN_p*[2].

[1] strictly speaking not a partition, since some of the sets may be empty

[2] 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

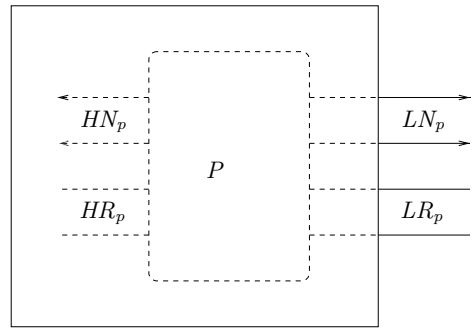

Fig. 1. Partitioning P's interface

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 LN_p test contains the special 'success' event ω 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

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

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 $P_i \in states(e)$ from which an ω event is possible:

Definition 2.

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

no difference in this paper since the part of an execution after success is irrelevant for our purposes.

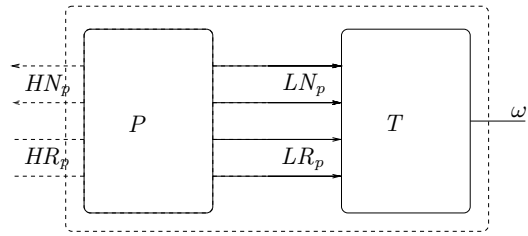

Fig. 2. Testing P

Different notions of testing are captured by different modalities of successful testing [dNH84]: 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. *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.

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

Definition 4. $P \equiv_{\mathsf{may_p}} Q$ *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 [dNH84] 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 $P\ \mathsf{may}_p\ T \Leftrightarrow P\ \mathsf{may}_p\ T'$ 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 $P \parallel T$.

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 1. *If* $P \equiv_{\mathsf{may_p}} Q$ *and* $L_{p'} \subseteq L_p$ *then* $P \equiv_{\mathsf{may_{p'}}} Q$

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.

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[3].

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 5. *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 6. *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 1. Let P_1 be the process:

$$P_1 = h_r \rightarrow h_n \rightarrow l_r \rightarrow l_n \rightarrow P_1$$
$$\Box\ l_r \rightarrow l_n \rightarrow P_1$$

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 firstly cannot perform l_n before l_r, and secondly is guaranteed to be able to accept l_r and then provide l_n.

Then P_1 must T_1. 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.

On the other hand, for the test $T_2 = l_n \rightarrow \omega \rightarrow Stop$, we have that $\neg(P_1$ 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:

[3] 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

Definition 7. $P \equiv_{\mathsf{must}_p} Q$ *if and only if* $\forall T : TEST_{LN_p} \bullet (P\ \mathsf{must}_p\ T \Leftrightarrow Q\ \mathsf{must}_p\ T)$

Example 2. If

$$P_2 = h_{r2} \rightarrow h_{n2} \rightarrow l_r \rightarrow l_n \rightarrow P_2 \\ \Box\ l_r \rightarrow l_n \rightarrow P_2$$

and

$$P_3 = h_{r2} \rightarrow h_{n2} \rightarrow l_r \rightarrow l_n \rightarrow P_3 \\ \sqcap\ h_n \rightarrow l_r \rightarrow l_n \rightarrow P_3 \\ \Box\ l_r \rightarrow l_n \rightarrow P_3$$

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) 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 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.

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 $T = l \rightarrow \omega \rightarrow RUN_l$.

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.

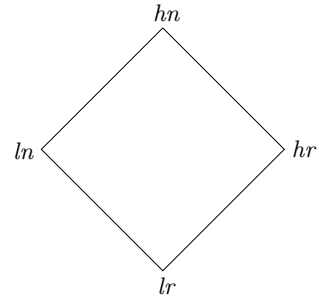

Fig. 3. The partial order $\leqslant$ on the interface partition

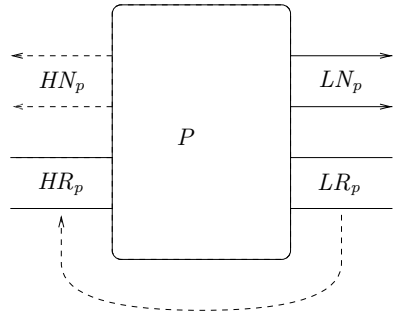

Fig. 4. Moving events from LR to HR

As the boundaries are shifted and the partition function p changes, the equivalence relation $\equiv_{\mathsf{must_p}}$ changes accordingly. The four possible locations of an event of P's interface may be ordered as in Figure 3, 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 3 is associated with a lemma which supports the claim that transferring events from the lower to the higher set preserves $\equiv_{\mathsf{must_p}}$. Each edge in general corresponds to a strict weakening of $\equiv_{\mathsf{must_p}}$: new equivalences are introduced in each case. The resulting Lemma 2 collects these results together: if p is pointwise weaker than p', then must_p equivalence implies $\mathsf{must}_{p'}$ equivalence.

For example, increasing HR at the expense of LR preserves must testing equivalence. This scenario is illustrated in Figure 4.

Lemma 2. *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,

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

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. *For any process* P*,* $P \equiv_{\mathsf{must}_p} (P \setminus HN_p)$

3.4 Examples

Example 3.

$$h_n \rightarrow l_r \rightarrow \mathit{Stop} \equiv_{\mathsf{must}_p} l_r \rightarrow \mathit{Stop}$$

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

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

$$h_r \rightarrow l_r \rightarrow \mathit{Stop} \equiv_{\mathsf{must}_p} (l_r \rightarrow \mathit{Stop}) \sqcap \mathit{Stop}$$

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 4.

$$\begin{array}{lll} h_{n1} \rightarrow l_{r1} \rightarrow \mathit{Stop} & & l_{r1} \rightarrow \mathit{Stop} \\ \Box & \equiv_{\mathsf{must}_p} & \sqcap \\ h_{n2} \rightarrow l_{r2} \rightarrow \mathit{Stop} & & l_{r2} \rightarrow \mathit{Stop} \\ & & h_{n2} \rightarrow l_{r1} \rightarrow \mathit{Stop} \\ & \equiv_{\mathsf{must}_p} & \Box \\ & & h_{n1} \rightarrow l_{r2} \rightarrow \mathit{Stop} \end{array}$$

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.

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 1.

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

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 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 2. *If* $LN = \varnothing$ *then*

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

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

This gives us a low-level view of a process: the low-level view of P is simply $(abs_p(P))$. 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.

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 an 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

$$ufailures_{LN}(P) = \{(tr, X) \in failures(P) \mid LN \subseteq X\}$$

Such a set does not meet the standard axioms for CSP, as Example 5 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 5. If $H = \varnothing$, $LN = \{ln\}$, and $LR = \{lr\}$, then

$$ufailures_{LN}(lr \rightarrow ln \rightarrow Stop) = \{(\langle\rangle, \{ln\}), (\langle lr, ln\rangle, \{ln\}), (\langle lr, ln\rangle, \{ln, lr\})\}$$

This set is neither prefix-closed on traces, subset closed on refusals, and on the behaviour with the empty trace the event lr 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 3. *If* $H = \varnothing$ *then*

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

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

A new CSP operator $\mathsf{sticky}_{LN}\ (P)$ can be defined which provides a context for characterising urgent failures: $FDI(\ \mathsf{sticky}_{LN}\ (P)) = FDI(\ \mathsf{sticky}_{LN}\ (Q))$ if and only P and Q have the same urgent failures, divergences, and traces.

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.

In the most general case, we arrive at the following characterisation of must equivalence:

Theorem 4.

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

Note that this theorem also covers the case where $LN = \varnothing$, since in that case sticky_{LN} simply has no effect.

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

Observe also that Theorem 3 given above can be characterised in this form:

Theorem 5. *If* $H = \varnothing$ *then*

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

More about sticky Different processes might map to the same result under sticky_{LN}. For example, $P_1 = ln_1 \rightarrow \mathit{Stop} \ \Box\ ln_2 \rightarrow \mathit{Stop}$ and $P_2 = ln_1 \rightarrow \mathit{Stop} \sqcap ln_2 \rightarrow \mathit{Stop}$ have different refusals, yet $\mathsf{sticky}_{LN}\ (P_1)$ and $\mathsf{sticky}_{LN}\ (P_2)$ have the same refusals. Hence from Theorem 3 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:

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

The operational semantics for sticky point the way to a definition in terms of the standard CSP operators. This can be achieved as follows: firstly, let f_{old} and g be event 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_{LN} as follows:

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

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:

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

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_{LN} 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.

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 6.

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

but

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

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

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

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

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

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 $\checkmark$ 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.

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 8. *A process P is* interference-free *with respect to p if*

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

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.

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

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:

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

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. (Generally in fact the system would be modelled so they are always available.)

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 primitive 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

$$\begin{aligned} FT = in?x \rightarrow\ & out!x \rightarrow Stop \\ & \Box\ fault \rightarrow recover \rightarrow out!x \rightarrow Stop \end{aligned}$$

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 of

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

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

Then $SPEC \sqsubseteq_{\mathsf{must}_p} FT$[4] .

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

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

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

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

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).

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 division p of the interface of the process, and the variation in the relations as 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.

[4] in other words, $\forall T : TEST_{LN_p} \bullet SPEC\ \mathsf{must}_p\ T \Rightarrow FT\ \mathsf{must}_p\ T$

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. Theorem 4 provides a complete characterisation of this equivalence.

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 a previous denotational characterisation on non-divergent processes[RWW94]. 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, Irfan Zakkiudin for discussion and comments on earlier forms of this work, to the anonymous referees for their detailed comments, 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.

[dNH84] R de Nicola and M Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1), 1984.

[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.

[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.

[Sch99a] S A Schneider. Abstraction and testing. Technical Report TR-99-02, Royal Holloway, 1999.

[Sch99b] S A Schneider. *Concurrent and real-time systems: the CSP approach*. John Wiley, 1999. to appear.