Abstraction and Testing in CSP

Steve Schneider

Department of Computer Science, Royal Holloway University of London, Egham, 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 a generalisation of 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.

Keywords: Abstraction; Process algebra; Failures semantics; Operational semantics; Testing; CSP

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 has only 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 are derived from this context.

Process algebraic approaches generally provide some operators to achieve 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’s interface and making them internal. Events internalised in this way become non-refusable in the sense that the system containing them will not be in a stable state if any of them are on offer; no other party will be able to block the process from performing them. However, Roscoe [Ros97] has observed that events in a process’s 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 enables specifications to be understood expressed in a natural way. The results in terms of standard CSP models then allows verification using the standard theory and tools.

The main contribution is to generalise the process algebraic notion of abstraction in operational terms, and to provide an equivalent formulation within the FDI (failures, divergences, infinite traces) model for CSP. The proofs for all of the results presented here can be found in the full version of this paper [Sch99a], and are omitted here for reasons of space.

The structure of this paper is as follows: Section 2 provides some brief motivation for this work; Section 3 introduces the CSP notation; Section 4 is concerned with the testing framework for equivalence of processes; Section 5 is concerned with denotational characterisations of the testing viewpoint; Sections 6 and 7 consider the application of this work to particular areas. The paper concludes with a discussion.

2. Motivation

When specifying the required behaviour of a system, it is often useful to focus on its activity with respect to a particular point of view. For example, part of the specification of a network of automated teller machines for banking will be concerned with the interactions that customers can have. Such specifications will need to abstract away the detail about the rest of the system and be expressed only in terms of the activities that can take place at the cashpoint.

There are a number of situations where it is appropriate to provide specifications in terms of the expected behaviour with respect to some viewpoint. These will be discussed in more detail in Sections 6 and 7, but will be introduced briefly here. As a first example, this approach allows fault-tolerant systems to be specified without requiring any particular fault-recovery mechanisms to be described at the specification stage. It is appropriate to give the required behaviour of the system from the viewpoint of a user who is not concerned with the fault-tolerance mechanisms. Another motivating example is concerned with atomicity of updates to the state in distributed systems [LMW94]. In such systems complicated algorithms are used to ensure consistency in the face of possible interference by different agents. From the point of view of any of the users, the behaviour of the system is simple to specify: it should appear as if the update occurs as a single atomic event, precluding interference.

The work in this paper was originally motivated by a different kind of problem: non-interference as a security property of systems. In this case, the concern is to ensure that the behaviour of the system on a particular interface does not reveal any information about activity in the rest of the system. Non-interference was originally formulated in terms of transition systems [GoM82, McL94]. Lack of information flow from a ‘high level’ within the system to a ‘low level’ is generally characterised in terms of activity at the high level making no difference to the visible behaviour at the low level. More recently [Rya91, RWW94, FoG97, RyS99], process algebraic characterisations have been proposed, concerning the behaviour of the system from a particular (low-level) viewpoint, with the rest of the system (that the low-level agent should not be able to access) abstracted. The process-algebraic approaches apply well-understood ideas from concurrency theory, in particular notions of equivalence and abstraction, to provide an alternative understanding of non-interference.

In the case of non-interference, the abstracted events are events that the observer should not know about, rather than simply internal details that the observer does not need to know about. The process of abstraction is the same, but the way it is used is more delicate and needs to be carried out with greater care. This paper contributes to this task by providing an understanding of the nature and varieties of abstraction and how they can be characterised in a way which makes them useful for analysis and verification.

3. 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 $P \square Q$ (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 $P \parallel [D] Q$ 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. Processes may also be recursively defined by means of equational definitions. For example, $COPY = in?x : T \rightarrow out!x \rightarrow COPY$ repeatedly accepts a value on channel in and then outputs it on channel out .

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 reached. 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 , $traces(P)$, 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 process $COPY$. The set of infinite traces, $infinities(P)$, are the infinite sequences of events from Σ that P might possibly perform.

The *failures* of a process P , $failures(P)$, 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 $(\langle \rangle, \{\})$, 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 , $divergences(P)$, 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 : $FDI(P) = \langle failures(P), divergences(P), infinities(P) \rangle$. 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 a subset of the semantics of P ; if the semantics models processes by tuples of sets, then $P \sqsubseteq Q$ if each set in the semantics of Q is a subset of the corresponding set in the semantics of P . Intuitively semantic sets correspond to the sets of observations that can be made of the process, so refinement may be understood as relating two processes when 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 could possibly do:

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

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

4. 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. Events will therefore be classified along two dimensions: whether they are to be abstracted, and whether or not they are refusable.

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 $L \cap H = \{\}$.

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 should be able 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. Events that the system P 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 along entirely similar lines.

Choices between high-level refusable events correspond to a kind of internal choice in the sense that the low-level environment has no control over how they are resolved. Conversely, choices over low-level refusable events correspond to external choice, in that the low-level environment participates in their occurrence.

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

This partition¹ of Σ will be characterised by a function p which indicates for each event which set it is in:

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

The relationship between p and the subsets of Σ is that

$$HN_p = \{a \mid p(a) = hn\}$$

$$HR_p = \{a \mid p(a) = hr\}$$

$$LN_p = \{a \mid p(a) = ln\}$$

$$LR_p = \{a \mid p(a) = lr\}$$

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 Fig. 1.

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

We now define the notion of an LN_p test. To do this, a special success event $\omega \notin \Sigma$ is introduced.

Definition 4.1. An 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. $\square$

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 set of all LN_p tests is denoted $TEST_{LN_p}$.

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

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

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 $P = P_0$. The states appearing in this execution, $states(e)$, are P_1 , P_2 , etc.

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

Fig. 1. Partitioning P 's interface. Fig. 2. Testing P .

For example,

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

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

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

is an infinite execution.

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

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

Definition 4.2.

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

□

Different notions of testing are captured by different modalities of successful testing [dNH87]: 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 denotational semantic models: 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.

4.1. 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 . The subscript will be dropped when it is clear from the context. Observe, however, that may_p testing 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 4.3. If P is a process, and T is an LN_p test, then $P \text{ may}_p T$ if and only if there is some successful execution e of $(P \parallel [L_p] \parallel T) \setminus L_p$. $\square$

Observe that there will be some successful execution of $(P \parallel [L_p] \parallel T) \setminus L_p$ if and only if there is some successful execution of $P \parallel [L_p] \parallel T$. 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 pass exactly the same LN_p may_p tests:

Definition 4.4. $P \equiv_{\text{may}_p} Q$ if and only if $\forall T : \text{TEST}_{LN_p} \bullet (P \text{ may}_p T \Leftrightarrow Q \text{ may}_p T)$. $\square$

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 $\{\}$. 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 an LN_p test T' such that $P \text{ may}_p T \Leftrightarrow P \text{ may}_p T'$ for any process P . This is achieved by introducing the extra possibility $\dots \square \text{RUN}_{LN_p}$ to every state. No new successful executions are introduced, since whenever any of these new choices are taken then there is no possibility of reaching ω ; so any successful execution of $P \parallel [L] \parallel T'$ must correspond to a successful execution of $P \parallel [L] \parallel T$.

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 is not relevant – the processes 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 4.5. If $P \equiv_{\text{may}_p} Q$ and $L_{p'} \subseteq L_p$ then $P \equiv_{\text{may}_{p'}} Q$. $\square$

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: tests have access to even less information, and so testing is considering processes at a higher level of abstraction. 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 on $L_{p'}$ is subsumed within the previous set of tests on L_p .

4.2. Must Testing

Must testing can be considered as the dual of may testing: it is concerned with the *guarantees* that can be made about process behaviour, rather than the *possibilities*. In may testing, a process passes a test if there is some successful execution. In must testing, a process passes a test if *every* execution is successful – if any run is guaranteed to reach a success state. 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's control to continue the execution.

Definition 4.6. 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 no internal transitions, or 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 4.7. 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 $(P \parallel [L_p] \parallel T) \setminus L_p$ is successful. □

Example 4.8. Let P_1 be the process:

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

Define the test $T_1 = l_n \rightarrow \text{RUN}_{l_n} \square l_r \rightarrow l_n \rightarrow \omega \rightarrow \text{Stop}$. This is guaranteed to 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_p T_1 . If a complete execution of $(P_1 \parallel [L_p] \parallel T_1) \setminus L_p$ begins with an internal l_r , then it is successful. If it begins with h_r , then it must continue with the non-refusable h_n , and so progress to the low-level events and hence to the success state. □

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

Definition 4.9. $P \equiv_{\text{must}_p} Q$ if and only if $\forall T : \text{TEST}_{LN_p} \bullet (P \text{ must}_p T \Leftrightarrow Q \text{ must}_p T)$. □

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 .

Example 4.10. Consider processes P_2 and P_3 defined as follows:

$$P_2 = (h_{r2} \rightarrow h_{n2} \rightarrow l_r \rightarrow l_n \rightarrow P_2) \square (l_r \rightarrow l_n \rightarrow P_2) \\ P_3 = l_r \rightarrow l_n \rightarrow P_3$$

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 the different high-level events that can occur in P_1 and P_2 , or even if those high-level events cannot occur, as in P_3 . □

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 extendible with refusable events.

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

Example 4.11. The following two processes differ only on the nature of the choice they provide between the two events:

$$P_4 = (l_{n1} \rightarrow \text{Stop}) \sqcap (l_{n2} \rightarrow \text{Stop})$$

$$P_5 = (l_{n1} \rightarrow \text{Stop}) \sqcap (l_{n2} \rightarrow \text{Stop})$$

If l_{n1} and l_{n2} are in LN then these processes are indistinguishable, because neither event can be blocked by any test in $TEST_{LN_p}$. Observe that if l_{n1} and l_{n2} were in LR then the test $T_2 = l_{n1} \rightarrow \omega \rightarrow \text{Stop}$ makes the distinction: P_4 must T_2 , but P_5 can fail T_2 . $\square$

If p_0 is such that $LR_{p_0} = \Sigma$, and H_{p_0} and LN_{p_0} are all $\{\}$, 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, $h \rightarrow l \rightarrow \text{Stop}$ is must_p equivalent to $l \rightarrow \text{Stop}$ 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 $T = l \rightarrow \omega \rightarrow \text{RUN}_l$.

4.3. 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 high-level and low-level events.

As the boundaries are shifted and the partition function p changes, the equivalence relation $\equiv_{\text{must}_p}$ changes accordingly. The four possible locations of an event of P 's interface may be ordered as in Fig. 3, where higher positions for events result in *weaker* equivalence relations: relations at a higher level of abstraction, which identify more processes. Thus as previously observed, if all events are in LR_p then the equivalence is strongest, and in fact corresponds 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 – there are no low-level events for a test to interact on.

Each edge of the order relation in Fig. 3 is associated with a lemma which supports the claim that transferring events from the lower to the higher set preserves $\equiv_{\text{must}_p}$. Each edge in general corresponds to a strict weakening of $\equiv_{\text{must}_p}$: new equivalences are introduced in each case. The resulting Lemma 4.12 collects these results together: if p is pointwise weaker than p' , then must_p equivalence implies $\text{must}_{p'}$ equivalence: $\equiv_{\text{must}_p}$ is stronger than $\equiv_{\text{must}_{p'}}$.

Fig. 4. Moving events from LR to HR .

Lemma 4.12. If $(\forall a : \Sigma.p(a) \leq p'(a))$ then for any P and Q , if $P \equiv_{\text{must}_p} Q$ then $P \equiv_{\text{must}_{p'}} Q$. $\square$

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

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

Lemma 4.13. For any process P , $P \equiv_{\text{must}_p} (P \setminus HN_p)$ $\square$

Example 4.14.

$$h_n \rightarrow l_r \rightarrow \text{Stop} \equiv_{\text{must}_p} l_r \rightarrow \text{Stop}$$

The non-refusability of the high-level event in the left-hand process 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:

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

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. $\square$

Example 4.15.

$$\begin{array}{ccc} h_{n1} \rightarrow l_{r1} \rightarrow \text{Stop} & & l_{r1} \rightarrow \text{Stop} \\ \square & \equiv_{\text{must}_p} & \square \\ h_{n2} \rightarrow l_{r2} \rightarrow \text{Stop} & & l_{r2} \rightarrow \text{Stop} \end{array}$$

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 between these two processes. $\square$

5. 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.

5.1. May Equivalence

With regard to may testing, the characterisation is unsurprising. 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 5.1.

$$P \equiv_{\text{may}_p} Q \Leftrightarrow \text{traces}(P \setminus H_p) = \text{traces}(Q \setminus H_p) \quad \square$$

5.2. 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 4.13 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 non-refusable, since they are not required to occur. This can be achieved by running the process in parallel with the process Chaos_{HR_p} , which can block such high-level events at any stage; and then hiding all the high-level events.

We will begin with the special case in which the set of low-level non-refusable events is empty. Thus tests can have control over the occurrence of all low-level events. As before, high-level non-refusable events can simply be hidden.

Theorem 5.2. If $LN = \{\}$ then

$$P \equiv_{\text{must}_p} Q \Leftrightarrow \text{FDI}((P \parallel [HR_p] \parallel \text{Chaos}_{HR_p}) \setminus H_p) = \text{FDI}((Q \parallel [HR_p] \parallel \text{Chaos}_{HR_p}) \setminus H_p) \quad \square$$

Following Roscoe [Ros97], the construction $(P \parallel [HR_p] \parallel \text{Chaos}_{HR_p}) \setminus H_p$ will be abbreviated $\text{abs}_p(P)$: an abstracted view of P in which only the behaviour on the low-level events are visible. Any two processes which exhibit this low-level behaviour are indistinguishable through that view of the process.

In the general case, the set of low-level non-refusable events is not empty, so 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 non-refusable but visible, which is not an aspect of standard CSP, but which has been analysed in the context of urgent events in timed CSP [DJS92]. The failures of such a process will be given as the non-refusable failures (with respect to LN), defined as

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

Such a set does not meet the standard axioms for CSP, as Example 5.3 illustrates. In fact, it turns out that *any* of the axioms pertaining to failures may be violated: 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 5.3. If $H = \{\}$, $LN = \{l_n\}$, and $LR = \{l_r\}$, then

$$\begin{aligned} \text{nrfailures}_{LN}(l_r \rightarrow l_n \rightarrow \text{Stop}) \\ = \{(\langle \rangle, \{l_n\}), (\langle l_r, l_n \rangle, \{l_n\}), (\langle l_r, l_n \rangle, \{l_n, l_r\})\} \end{aligned}$$

This set is neither prefix-closed on traces, subset-closed on refusals, and on the behaviour with the empty trace the event l_r can extend neither the trace (by itself) nor the refusal.

A recursive process that is always willing to perform events from LN , such as $P = l_n \rightarrow P$, has an empty non-refusable 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. $\square$

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

Theorem 5.4. If $H = \{\}$ then

$$\begin{aligned} P \equiv_{\text{must}_p} Q &\Leftrightarrow \text{nrfailures}_{LN}(P) = \text{nrfailures}_{LN}(Q) \\ &\quad \wedge \text{divergences}(P) = \text{divergences}(Q) \\ &\quad \wedge \text{infinities}(P) = \text{infinities}(Q) \end{aligned}$$

□

A new CSP operator $\text{sticky}_{LN}(P)$ can be defined which provides a context within the standard FDI model for characterising equivalence of non-refusable failures: $\text{FDI}(\text{sticky}_{LN}(P)) = \text{FDI}(\text{sticky}_{LN}(Q))$ if and only if P and Q have the same non-refusable failures, divergences, and traces.

The sticky operator masks the appropriate 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 an effect similar to making the events in LN non-refusable, since it is the process P itself (and not its environment) that chooses the event to be performed. The difference is that 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.

The FDI semantics of sticky_{LN} is given as follows:

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

Observe that $\text{sticky}_{LN}(P)$ has the same traces as P . It is only additional refusals that are introduced into the failure set. Where extracting non-refusable failures loses information from the failures of a process by removing some failures, sticky loses information by the masking effect of adding failures. For example, P_4 and P_5 of Example 4.11 have different refusals, yet $\text{sticky}_{LN}(P_4)$ and $\text{sticky}_{LN}(P_5)$ have the same refusals.

Thus we obtain an alternative to Theorem 5.4, characterised in this form:

Theorem 5.5. If $H = \{\}$ then

$$P \equiv_{\text{must}_p} Q \Leftrightarrow \text{sticky}_{LN}(P) = \text{sticky}_{LN}(Q)$$

□

In the most general case where we have an arbitrary partition of the events into the four categories, we arrive at the following characterisation of must equivalence:

Theorem 5.6.

$$P \equiv_{\text{must}_p} Q \Leftrightarrow \text{sticky}_{LN}(\text{abs}_p(P)) = \text{sticky}_{LN}(\text{abs}_p(Q))$$

□

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

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

This new operator sticky can also be given an operational semantics, which provides an alternative understanding of its behaviour. The process $\text{sticky}_{LN}(P)$ 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:

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

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

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

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

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

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 and g be event renaming functions such that $f(a) = (old, a)$ 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:

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

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:

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

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.

5.3. 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 high-level events, and if processes differ there then they may be affected differently.

Example 5.7.

$$h_{r1} \rightarrow l \rightarrow \text{Stop} \equiv_{\text{must}_p} h_{r2} \rightarrow l \rightarrow \text{Stop}$$

but

$$\text{Stop} \parallel [\{h_{r1}\}] \parallel h_{r1} \rightarrow l \rightarrow \text{Stop} \not\equiv_{\text{must}_p} \text{Stop} \parallel [\{h_{r1}\}] \parallel h_{r2} \rightarrow l \rightarrow \text{Stop}$$

Since the processes behave differently on the abstracted event h_{r1} , 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

$$f(h_{r1} \rightarrow l \rightarrow \text{Stop}) \not\equiv_{\text{must}_p} f(h_{r2} \rightarrow l \rightarrow \text{Stop})$$

A final example uses an event renaming f 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

$$f(h_{r1} \rightarrow l \rightarrow \text{Stop}) \not\equiv_{\text{must}_p} f(h_{r2} \rightarrow l \rightarrow \text{Stop})$$

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; in the last example may equivalence is preserved. $\square$

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 $\parallel [A]$

provided $A \subseteq L$. Event renaming $f(P)$ preserves $\equiv_{\text{must}_p}$ provided f respects the partitions: $p(f(a)) = p(a)$ for all events a . With $\equiv_{\text{may}_p}$, 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.

6. Non-interference

The examples in this section and the next demonstrate the use of must_p equivalence and refinement in non-interference, atomicity, and fault-tolerance specifications. The results given in previous sections would enable verifications to be carried out (by hand or with model-checking tool support) for proposed implementations.

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-level process activity to the low-level process activity 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.

As an example of what happens if this does not occur, consider the process

$$LEAK = in_H ?x \rightarrow out_L !x \rightarrow LEAK$$

$LEAK$ is a process which takes in messages on a high-level channel, and immediately communicates them on a low-level channel. A low-level event (e.g. $out_L.3$) is possible only if $in_H.3$ has previously occurred, so the possibilities at the low level are dependent on previous high-level activity, and hence there is some information flow from the high level to the low level.

This characterisation can be made in various ways. It has traditionally been made denotationally, and this has led to some difficulties in 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 6.1. A process P is *interference-free* with respect to p if

$$P \xRightarrow{e'} P' \wedge P \xRightarrow{e''} P'' \wedge Ltrace_p(e') = Ltrace_p(e'') \Rightarrow P' \equiv_{\text{must}_p} P'' \quad \square$$

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: for example, 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.

6.1. 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: $\perp = \perp$. It is must equivalent to $\perp \sqcap LEAK$, yet $\perp$ meets the definition, whereas $\perp \sqcap LEAK$ does not.

The desire to have the property ‘no information flow’ preserved by refinement has led to some difficulties with regard to this kind of example. If $\perp$ 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 $\perp$ or else will not be preserved by refinement.

6.2. 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 the characterisation of non-interference given in [RWW94].

Theorem 6.2. If P is divergence-free, then P is interference-free on p if and only if $abs_p(P)$ is deterministic. $\square$

Observe that this theorem holds even if $LN \neq \{\}$, 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:

$$\begin{aligned} P_1 &= h_{n1} \rightarrow l_{n1} \rightarrow P_1 \sqcap h_{n2} \rightarrow l_{n2} \rightarrow P_1 \\ P_2 &= h : \{h_{n1}, h_{n2}\} \rightarrow l : \{l_{n1}, l_{n2}\} \rightarrow P_2 \end{aligned}$$

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.

This example illustrates how the characterisation of abstraction through testing allows definitions of non-interference such as that given in Definition 6.1 to be expressed in an intuitive way. Stating that there should be no information flow can be understood as requiring that low-level activity should not give any information about what has occurred at the high level. Hence it permits characterisations of non-interference, and provides an alternative operational insight into the definition of non-interference presented in [RWW94].

7. 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’s description, indicating that the fault can occur at that stage during the process’s execution. They will then be modelled with 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.

7.1. Communications

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

$$\begin{aligned} FBUFF &= in?x \rightarrow out!x \rightarrow FBUFF \\ &\quad \sqcap power_blip \rightarrow FBUFF \\ &\quad \sqcap power_blip \rightarrow FBUFF \end{aligned}$$

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 simple example, a system might develop a fault between input and output, from which it must recover before performing output. This could be modelled, extremely crudely, as

$$FT = in?x \rightarrow (out!x \rightarrow Stop \\ \square fault \rightarrow recover \rightarrow out!x \rightarrow Stop)$$

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

In other words, the *requirement* on the system FT is that it is equivalent to $SPEC$ under $must_p$ testing: $SPEC \equiv_{must_p} FT$. Here we have $p(fault) = hr$ and $p(recover) = hn$.

Having *fault* as refusable expresses the modelling decision that faults might, but need not, happen. The description of the process will contain the possibility of a fault event, but it might not occur. Events that are high-level non-refusable are expected to occur when they are possible for the process, so generally these will be events under the control of the process, such as recovery events.

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

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

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).

In general, a specification on the fault-tolerant system does not require equivalence between the specification and the implementation. If the implementation behaves better than the specification allows, then it will still be acceptable. Thus in general a refinement relation will be appropriate in the specification.

The appropriate refinement relation between $SPEC$ and FT would be written $SPEC \sqsubseteq_{must_p} FT$, which is defined by

$$SPEC \sqsubseteq_{must_p} FT \\ \Leftrightarrow \forall T : TEST_{LN_p} \bullet SPEC \text{ must}_p T \Rightarrow FT \text{ must}_p T$$

7.2. Memory Cell

A memory cell may be specified by the following process description:

$$MEM(v) = write?w \rightarrow MEM(w) \\ \square \\ read!v \rightarrow MEM(v)$$

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 $\{write, read\}$:

$$MEM(v) \sqsubseteq_{must_p} MEMIMP(v)$$

In this case, all the ‘high’-level events would be internal system events and would therefore be non-refusable, in HR . The low-level events on the channels *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:

$$FTMEM(v) = write?w \rightarrow (FTMEM(w) \sqcap FTMEM(v))$$

□

$$read!v \rightarrow FTMEM(v)$$

An implementation of this, *FTMEMIMP*, will be required to satisfy

$$FTMEM(v) \sqsubseteq_{\text{must}_p} FTMEMIMP(v)$$

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 *FTMEMIMP*. The faults that are possible will be modelled explicitly as events within the appropriate 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, some faults might result in a failure to achieve 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 further specification on *FTMEMIMP* is thus appropriate: a liveness or correctness expectation would be that the service is not degraded at all when no faults occur: that memory should be updated when all events in the set *FAULTS* are blocked:

$$MEM(v) \sqsubseteq_{\text{must}_p} Stop \mid [FAULTS] \mid FTMEMIMP(v)$$

7.3. 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 $SPEC \sqsubseteq_{\text{must}_p} IMP$. 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.

8. 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 correlate with the denotational characterisation [RWW94, Ros97] in terms of abs_p , have 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 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 has 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 5.6 provides a complete characterisation of this equivalence.

We have also seen that these testing equivalences are not full congruences – there are a number of operators which do not preserve equivalence, notably those which have some impact on high-level behaviour. Larsen has previously considered context-dependent equivalence between processes [Lar87], in the context of CCS. In this paper, we are concerned with equivalence within all of the particular contexts $C_P(X) = \text{abs}_p(X \mid [L] \mid P)$ (parametrised by process P). These contexts allow interaction with process P only on low-level events, and P must always be willing to perform non-refusable events.

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 and Irfan Zakkiudin for discussion and comments on earlier forms of this work, to the anonymous referees for their detailed comments, to Bill Roscoe for pointing out the CSP characterisation of sticky, and to Michael Goldsmith and Jason Hulance for the memory cell example. Support for this work was provided by DERA.

References

- [DJS92] Davies, J., Jackson, D. M. and Schneider, S. A.: Broadcast communication for real-time processes. In *Proceedings of the Symposium on Real-Time and Fault-Tolerant Systems*. Springer-Verlag, LNCS 571, 1992.
- [dNH87] de Nicola, R. and Hennessy, M.: Testing equivalences for processes. *Theoretical Computer Science*, 34(1): 83–134, 1987.
- [FoG97] Focardi, R. and Gorrieri, R.: The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9): 550–571, 1997.
- [GoM82] Goguen, J. A. and Meseguer, J.: Security policies and security models. In *IEEE Symposium on Research in Security and Privacy*, Oakland, California, IEEE Press 1982.
- [Hen88] Hennessy, M.: *Algebraic Theory of Processes*. MIT Press, 1988.
- [Lar87] Larsen, K. G.: A context dependent equivalence between processes. *Theoretical Computer Science*, 49(2): 185–216, 1987.
- [LMW94] Lynch, N., Merritt, M., Weihl, W. and Fekete, A.: *Atomic Transactions*. Morgan Kaufmann, 1994.
- [McL94] McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Research in Security and Privacy*, Oakland, California, IEEE Press 1994.
- [Ros97] Roscoe, A. W.: *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
- [RyS99] Ryan, P. Y. A. and Schneider, S. A.: Process algebra and non-interference. In *12th IEEE Computer Security Foundations Workshop*, Mordano, Italy, IEEE Press 1999.
- [RWW94] Roscoe, A. W., Woodcock, J. C. P. and Wulf, L.: Non-interference through determinism. In *European Symposium on Research in Computer Security*, volume 875 of LNCS. Springer-Verlag, 1994.
- [Rya91] Ryan, P. Y. A.: A CSP formulation of non-interference and unwinding. *Cipher Newsletter*, 19–27, IEEE Press 1991.
- [Sch99a] Schneider, S. A.: Abstraction and testing. Technical Report TR-99-02, Royal Holloway, 1999.
- [Sch99b] Schneider, S. A.: *Concurrent and Real-Time Systems: The CSP Approach*. John Wiley, 1999.

Received February 2000

Accepted in revised form April 2000