Journal of Universal Computer Science, vol. 11, no. 5 (2005), 771-802
submitted: 30/11/04, accepted: 31/1/05, appeared: 28/5/05 © J.UCS

Relaxing Atomicity and Verifying Correctness: Considering the Case of an Asynchronous Communication Mechanism

Jonathan Burton
(University of Newcastle upon Tyne, UK
j.i.burton@ncl.ac.uk)

Abstract: In an ideal world, where we could guarantee instantaneous, atomic data transfer — whatever the type of the data being transferred — shared memory communication between two concurrent processes could be implemented directly using single variables or registers, without any attendant access control policies or mechanisms. In practice, asynchronous communication mechanisms may be used to provide the illusion of atomic transfers of data while still allowing non-blocking reads and writes: that is, reads and writes may proceed concurrently without interfering with each other. In order to prove the correctness of such mechanisms, the natural approach would be to verify them against the specification provided by an idealised register with atomic, instantaneous — and so sequential — transfers of data. Yet such a verification is complicated by the fact that, in moving to the asynchronous communication mechanism from such a specification, additional concurrency has been introduced and so the (visible) behaviours of the mechanism are not directly comparable to those of the register. In this paper, we recall an extension of standard process algebraic refinement and show how it may be used to verify the correctness of a particular asynchronous communication mechanism, Simpson's 4-slot. In so doing, we look at a number of issues which seem significant in the consideration of correctness when the real atomicity of a specification has been relaxed in the move from specification to implementation.

Key Words: asynchronous communication, process algebra, relaxation of atomicity, verification
Category: F.3.1 Specifying and Verifying and Reasoning About Programs

1 Introduction

In an ideal world, where we could guarantee instantaneous, atomic[1] data transfer — whatever the type of the data being transferred — shared memory communication between two concurrent processes could be implemented directly using single variables or registers, without any attendant access control policies or mechanisms. However, such atomic data transfers are not possible and if, for example, a reading process and a writing process were allowed unconstrained access

[1] In this paper, we shall describe (sequences of) events as "atomic" (with respect to each other) if their occurrences do not overlap in time and so their respective executions cannot interfere with each other. This use of "atomic" is to be distinguished from the all-or-nothing property which is implied by the A of ACID in the database literature (see, for example, [24]). In fact, it is similar to the isolation described by the I of ACID.

to such a variable or register, interference would occur due to the overlapping of read and write events.

Usually, if communication is to take place between two concurrent processes via a shared memory area, some form of synchronization will be required in order to avoid interference. Such synchronization may take the form of a *critical section* or *handshake communication*. However, this may force one or both of the communicating processes to wait or block while the other completes a data transfer; this may be undesirable, particularly in a real-time environment. Even a buffer is not fully asynchronous: if it becomes full, further writes will be blocked until a read occurs and, if it becomes empty, further reads will be blocked until a write occurs.

It is to solve this problem that asynchronous communication mechanisms or ACMs have been introduced. Such mechanisms are characterised by the fact that, if used by a single reader and writer, neither the reader nor the writer will ever have to wait before it is allowed to interact with the mechanism. As a result, a writer may always write to an ACM and the reader may always read from it: that is, writes are destructive and may overwrite data already written, while reads are non-destructive and so re-reading is allowed. In order to allow such unconstrained access despite the reality of non-atomic data transfer, ACMs combine some sort of access logic with multiple data slots. The multiple data slots allow a read and a write to proceed concurrently without interfering with each other, while the access logic ensures the reader and writer never access the same slot at the same time. (The specific ACM we consider here is Simpson's 4-slot mechanism from [22].)

Intuitively, any particular ACM, M, is intended to mimic the behaviour of an idealised register, R. R stores a single value of the data type under consideration, and reads and writes in R occur instantaneously — i.e. with zero duration — and so atomically. Thus, accesses to R are *sequential*, since it is not possible for reads and writes to proceed concurrently. Since M does allow reads and writes to proceed concurrently, additional concurrency has been added in the move from the specification R to the implementation M and so the atomicity of the operations of R with respect to each other has been relaxed. We now consider some of the issues which this relaxation of atomicity may raise with respect to verification. (Note that we refer to the general process of introducing additional concurrency in the move from specification to implementation as *relaxation* or *refinement of atomicity*.)

In general, M will be endowed with a procedural interface via which it communicates with the wider process in which it is embedded and it is possible to give R the same interface: more specifically, a process may call the procedure to write a value and on completion of the write the procedure will return; similarly, a process may call the procedure to read a value from the mechanism and, on completion, the procedure will return the value read; moreover, all internal processing is hidden from the outside world. Thus, if we consider the sequences of *observable* events which may be performed by M or R, those sequences will consist of only four different types of event (where relevant, of course, a particular type of event may transmit a number of different data values): an event denoting a call to the write procedure, an event denoting a return from the write procedure, an event denoting a call to the read procedure and an event denoting a return from the read procedure. For the purposes of the discussion here, we will denote these CW (a call to write), RW (a return from the write procedure), CR (a call to read) and RR (a return from the read procedure).

In any sequence of events which may be performed by R, each call other than the last in the sequence must be immediately followed by a matching return. Similarly, a return from a particular procedure must be immediately preceded by the corresponding procedure call. Thus, sequences such as $\langle CW, RW, CR, RR, CR\rangle$ or $\langle CW, RW, CW, RW\rangle$ are possible for R. However, M may exhibit a wider range of behaviours, since it allows reads and writes to proceed concurrently: for example, $\langle CW, CR, RR, RW\rangle$ or $\langle CW, CR, RW, CW, RW, RR\rangle$ are possible executions of M which are not possible for R. Hence, the relaxation of atomicity which has occurred manifests itself in the *observable* behaviours of M and it is this fact which complicates the verification task with which we are faced.

The *traces model* (see Section 2 for further details) is one of the simplest — and least discriminating — models of (concurrent) process behaviour: it regards the meaning of a process to be the set of finite sequences of observable events which it can perform. An implementation process is regarded as correct if every sequence of visible events which it can perform can also be performed by the specification process under consideration. By virtue of the discussion above, it is immediately obvious that such a notion is unsuitable for relating the behaviours of M to those of R.

In order to address this problem, we use a variant of the traces model notion of correctness called *refinement-after-hiding* (further details are given in Section 4). This addresses the problem of a mismatch of visible behaviours after the occurrence of relaxation of atomicity by displacing the boundary at which we assume process behaviours can be observed. Rather than comparing directly the behaviours of M and R, we appeal to a different notion of correctness based on the less strict requirement that the behaviours of the two processes may not be distinguished when placed in appropriate process contexts (a context is simply a process with a “hole” into which another process — such as the ACM under consideration here — may be plugged). In order to check for correctness in practice, we define an interpretive mapping and verify that the interpreted behaviours of M are contained in those of R. Using this notion of correctness, we are able to show that the specific ACM which is considered here — the 4-slot mechanism — is a valid implementation of the idealised register, R. As well as being an exercise in the application of refinement-after-hiding, this result is significant in its own right: the asynchronous communication community tends not to consider correctness in terms of an explicit specification process such as the register, rather focusing on particular desirable properties of the behaviours of ACMs (see Section 9 for further details). Thus, it was not initially clear that the verification attempted here would actually succeed. Moreover, the verification has some wider significance in terms of considering relaxation of atomicity in general, which issue is discussed in Section 8. Note that, for reasons of space, the presentation in this paper focuses on showing correctness in the traces model. However, refinement-after-hiding is also defined in [2] in terms of more complicated models which deal with deadlock and liveness properties; moreover, the correctness of the ACM under consideration here has been shown in [2] in those models as well (further details are given in subsequent sections).

Before proceeding, we make an important point regarding related work. The verification problem described above is not unique in character and is faced in some form in many different computing domains, for example that of database or transaction-processing systems. In such systems, the main notion of correctness used is *serializability* (see, for example, [1, 24]), which allows us to relate the concurrent execution of a set of transactions performed by a system to some sequential execution of the same set of transactions. However, serializability is not suitable for the verification task considered here because it does not allow the use of an explicit specification process. Moreover, it has no facility to model explicitly any context into which we might place the ACM under consideration and, in any case, has no notion of process composition. Hence, we would not be able to consider the correctness of any wider network in which the ACM was deployed. The problem of the correctness of the ACM is, however, typical of those to which the correctness condition of *linearizability* ([10]) may be applied. Section 9 contrasts our approach with that of linearizability and also looks at other related work.

The remainder of the paper is organised as follows: Section 2 presents necessary preliminary information; Section 3 presents and explains Simpson's 4-slot mechanism; Section 4 introduces refinement-after-hiding; Section 5 describes the formal model of the 4-slot which we use in our verification; Section 6 details the verification itself; Section 7 considers the issue of verifying processes which may communicate with the 4-slot; Section 8 considers some issues raised by the success of the verification; finally, Section 9 looks at related work. The treatment in this paper is deliberately light in terms of technical details, so as to ease reading and also due to space restrictions. A full account of the work described here can be found in [2]. A general consideration of relaxation of atomicity can be found in [3] in this volume.

2 Preliminaries

We now give details of the process algebra CSP ([11, 19]), which is the formal model of concurrent systems used in this paper. Note, however, that we give only enough detail to support the material in the paper and the interested reader is referred to [19] for a full treatment.

A CSP process may be regarded as a black box — that is, we are interested only in externally observable behaviour — which can communicate with its external environment. Atomic instances of this communication are called *events* or *actions* and we say that a process may *engage* in a particular event when it is possible for it to communicate that event at some point during its lifetime. Events in CSP occur on communication *channels*: for example, channel c will be given an associated type and we may then use it to communicate any event $c.v$ such that v is a value of the appropriate type. It is also possible to define c as a simple channel without any associated data content and so, in certain cases, c may be used to denote the occurrence of an event. In CSP, an event occurs only when all of its participants are ready to execute it and, as soon as all of the participants are ready to execute an event, then it (or some other event) *must* occur. Moreover, event occurrences are instantaneous, as we abstract their duration into single moments, and they are assumed to be non-overlapping as we use an interleaving semantics.

CSP processes may be assigned a semantics in one of three different models, the traces model, the stable failures model or the failures divergences model. Due to limitations of space, we focus in this paper on the simplest of the three, the traces model. However, as indicated above, refinement-after-hiding is defined in [2] in terms of all three semantic models and the correctness of the ACM under consideration here has been shown in [2] in all three models. In the traces model, a process is denoted by a (possibly infinite) set of finite execution sequences of *visible* actions. For any process P, we denote the *traces* of P as τP. That Q is an implementation of (or *refines*) another process P in the traces model means that $\tau Q \subseteq \tau P$: i.e. every trace which Q may exhibit is also possible for P. This is denoted as $Q \sqsupseteq P$. Note that a specification in CSP is presented constructively as another process: both specifications and implementations are described using the same language and are given the same behavioural semantics. The traces model is suitable for considering issues of safety. Divergences — traces after which a process may engage in an infinite sequence of internal actions — are used to model the possibility of livelock. Trace/refusal pairs (failures) are used to model the possibility of deadlock, where a refusal is a set of events that a process may fail to offer after a given trace. The stable failures model uses traces and failures to model processes, while the failures divergences model uses divergences and failures. Refinement in both of these models equates to containment of behaviours, as it does in the traces model.

For any process P, αP — the *alphabet* of P — gives the set of events in which that process may engage. The way in which alphabets are calculated is described in [2]. The notion of alphabet is used both in the definition of refinement-after-hiding in Section 4 and in the definition of parallel composition which is given below (it is not used, however, when giving a CSP semantics to processes).

We now describe the subset of CSP operators which we shall need here. The operators are introduced informally and are given without a formal semantics as that is not needed here. A formal semantics of these operators may be found in [19]. $a \rightarrow P$ (referred to as the prefix operator) gives the process which first engages in the event a and then proceeds to behave like P. In a similar vein, $c?x \rightarrow P$, where c is a channel and x is a variable, denotes the process which is initially ready to engage in any event of the form $c.v$ and which then proceeds to behave as P but with the value v substituted for the variable x in P. We use $\Box$ to denote deterministic choice: $P \Box Q$ is a process where the initial events of P and Q are offered simultaneously. If P is a process and A is a set of events, then $P \setminus A$ is the process which behaves like P with the actions from A made invisible ($\setminus$ is the *hiding* operator and $\setminus A$ *hides* the events in A). Parallel composition $P \parallel Q$ models synchronous communication between P and Q in such a way that they have to engage simultaneously in all actions in $\alpha P \cap \alpha Q$ — i.e. all actions which the processes have in common — while each of them is free to engage independently in any action not in that set.[2] (We say that the parallel composition *synchronizes* on the set of shared events or that P and Q synchronize on those events.) Note that the interleaving operator, $|||$, may be used to denote parallel composition when $\alpha P \cap \alpha Q = \varnothing$ and so P and Q are allowed to proceed in parallel completely independently of each other. Recursion may be introduced using an equational definition of the form $N = P$, where P may contain one or more instances of the process name N. The notation $N(x) = P$ is used to represent the *family* of processes $N(v)$, where v is a data value and $N(v)$ denotes the process P with all occurrences of the parameter x replaced by v. We also use an operator which is not part of the standard CSP syntax and which is, in fact, shorthand for a particular combination of two operators already seen. The *network composition* operator, $\otimes$, is defined in the following way: $P \otimes Q \triangleq (P \parallel Q) \setminus (\alpha P \cap \alpha Q)$ (note that $X \triangleq Y$ means X is taken to be equal to Y by definition). In other words, it allows us to model the fact that two processes engage in communication between themselves but that no other process may directly observe or share in that communication. As with the parallel composition operator, $\otimes$ would be parameterized in practice with a set

[2] In reality, the parallel composition operator, $\parallel$, used in [2] and described in [19] is parameterized with a set of events, Y. Then, in the composition $P \parallel_Y Q$, P and Q have to engage simultaneously in all actions from Y while each is allowed to engage independently in any action which is not in Y. In practice, we usually use this parallel composition operator only in cases where $Y = \alpha P \cap \alpha Q$ and so are able to simplify its presentation for the purposes of this paper.

of events, Y; moreover, its definition would hide that set rather than $\alpha P \cap \alpha Q$.

The semantics of processes in CSP is compositional in the sense that the correctness of a particular combination of processes may be verified by considering in isolation the correctness of each of the processes to be combined. Thus, for any binary operator, $\oplus$, and processes Q, Q', P, P', that $Q \sqsupseteq P$ and $Q' \sqsupseteq P'$ allows us to infer that $(Q \oplus Q') \sqsupseteq (P \oplus P')$. (Note that this property holds in all three models of process behaviour.)

The following notation and concepts will be useful in the remainder of the paper, where $a_1, \ldots, a_n$ are events or actions, t, u are traces, $\mathcal{T}, \mathcal{T}'$ are sets of traces and $f : \mathcal{T} \to \mathcal{T}'$ is a mapping: $t = \langle a_1, \ldots, a_n \rangle$ is the trace whose i-th element is a_i; moreover, $\mathit{events}(t) \triangleq \{a_1, \ldots, a_n\}$ and, if $n = 0$, then t is the empty trace, denoted $\langle\,\rangle$; $t \circ u$ is the trace obtained by appending u to t; $\leq$ denotes the prefix relation on traces, and $t < u$ if $t \leq u$ and $t \neq u$; f is *monotonic* if $t, u \in \mathcal{T}$ and $t \leq u$ implies $f(t) \leq f(u)$, and *strict* if $\langle\,\rangle \in \mathcal{T}$ and $f(\langle\,\rangle) = \langle\,\rangle$. In this paper, the term "environment process" is used to mean a process with which the ACM under consideration might be composed. A context is simply a process with a "hole" into which another process, such as an ACM, may be plugged. Note that we may turn an environment process into a context by using the former as one of the arguments to a binary operator and leaving a space where the second argument should go: this space functions as the hole into which another process may be plugged.

3 Simpson's 4-slot mechanism

We now introduce Simpson's 4-slot mechanism, the verification of which is the subject of this paper. The software version of the mechanism from [22] is given in Figure 1. It is assumed to allow at most one read and one write to execute concurrently (see the rendering of the 4-slot in CSP in Section 5). It contains — as the name suggests — four data slots, arranged into two *pairs* of two *slots*. Each of these slots stores a value of type *datatype*[3] and together they constitute a 2-dimensional array, *data*, which is a global variable. The first dimension of the array represents the pair, the second the two slots within that pair. Intuitively, the writer tries to avoid the reader as it seeks to write into the mechanism, while the reader chases after the writer in order to read the last piece of data written (in the remainder of this paper, we use "writer" as shorthand for "write procedure" and "reader" as shorthand for "read procedure").

Three global variables — referred to as control variables — are used in order to manage the behaviour of the reader and writer respectively. These are *latest*, *reading* and *slot*. *latest* is a bit variable indicating the pair to which the writer

[3] In the general case, *datatype* will be a complex type whose reading and writing are not guaranteed to be atomic by the underlying system on which the 4-slot mechanism is implemented.

```
Global variables: $reading, latest : bit$
                  $slot : array\ of\ bit$
                  $data : array\ of\ (array\ of\ datatype)$

    procedure $write\ (item : datatype);$
        $var\ \ pair,\ index : bit;$
        $begin$
            $pair := not(reading);$
            $index := not(slot[pair]);$
            $data[pair, index] := item;$
            $slot[pair] := index;$
            $latest := pair;$
        $end;$

    procedure $read : datatype;$
        $var\ \ pair,\ index : bit;$
        $begin$
            $pair := latest;$
            $reading := pair;$
            $index := slot[pair];$
            $read := data[pair, index];$
        $end$
```

Figure 1: Simpson's 4-slot mechanism

last wrote, while *slot[i]* tells the reader which slot was last written to in pair *i*. The bit variable *reading* tells the writer the pair from which the reader is about to read or from which it has just read. Note also that *pair* and *index* are variables that are local to the read and write procedures.

The behaviour of the reader is relatively straightforward to understand. It ascertains the pair to which the writer last wrote and places this value in the local variable *pair*. It then indicates to the writer that it is going to read from this pair by storing the value in *reading*, before discovering the slot last written to in the pair by interrogating the variable *slot*. Finally, it reads the data item stored in *data* at the relevant pair and slot. Note that the data transfer from *data* which represents this read will *not* occur atomically in the general case.

As indicated above, the writer aims to avoid the slot and pair combination in which the reader finds itself. It first decides to write to the pair in which the reader has *not* indicated an interest via *reading* (we assume that $not(0) = 1$ and $not(1) = 0$). It then decides to write to the slot in that pair which contains

```
Global variable: $data$ : $datatype$

procedure $write$ ($item$ : $datatype$);
        $begin$
                $data := item$;
        $end$;

procedure $read$ : $datatype$;
        $begin$
                $read := data$;
        $end$
```

Figure 2: A register

the oldest value. This means that it is impossible to immediately overwrite the last data value written into the mechanism. It also means that the writer avoids the reader in the event that the latter is reading from this pair. (This may happen despite the efforts of the writer to choose the alternative pair due to the arbitrary interleaving of the commands contained in the respective read and write procedures.) The relevant data value is then written — non-atomically — into the correct pair and slot combination. *slot* is updated to indicate which slot was written to in the relevant pair before, finally, *latest* is updated to indicate to the reader the pair in which the last write occurred.

As indicated above, executions of the read procedure and of the write procedure may proceed concurrently and so the commands they contain can be arbitrarily interleaved. This is obviously necessary if we are to have non-blocking — and so asynchronous — communication. And it is this fact of arbitrary interleaving, along with the fact that data transfers are non-atomic, which leads to the need for verification to ensure that the mechanism does, indeed, behave as desired.

4 Introducing refinement-after-hiding

As indicated in Section 1, the 4-slot is intended to mimic the functionality of a register despite the fact that we cannot guarantee atomicity of read and write operations (see Figure 2 for a procedure-based representation of a register, where it is assumed that the read and write procedures *do* execute atomically and so only one procedure may be executing at any one time). We have already seen in Section 1 that, despite the conformance of the 4-slot interface with that of the register, we could not use standard CSP refinement to show that the 4-slot is a correct implementation of the register. This is an example of a more general problem with process algebraic notions of correctness. In this section, we describe this problem in more detail before going on to introduce one possible solution in the form of the notion of correctness from [2] known as *refinement-after-hiding*.

4.1 Problems with standard process algebraic refinement

Process algebras, such as those in [16, 19], are intended for the description and verification of concurrent systems. Since the primary focus with these formalisms is on the interactions which occur between concurrent processes, their semantics abstracts away from the internal behaviour of processes. In general, that one process implements another means that — in some sense — all visible behaviours of the first process must be possible for the second process (see definition of CSP trace refinement in Section 2). As a result, standard process algebraic notions of refinement can usually deal only with the case that the process of transforming a specification to an implementation has wrought only internal, *invisible* changes. This makes sense to the extent that notions of correctness should be concerned with *what* a process does and not *how* it does it. However, it has certain consequences for verification.

During the — software or hardware — development process, we take a specification and, through a process of *reification*, produce an implementation. It is possible to reify the data structures of a specification to make them less abstract and more suited to implementation; abstract specification behaviours may be represented in a more concrete and so more implementable form; and we may introduce additional concurrency not in evidence in the specification (such a change is used in the move from the register to the 4-slot). Since data reification is, by definition, internal, its effects do not manifest themselves directly in terms of visible behaviours. However, both the decomposition of behaviours and the relaxation of atomicity may manifest themselves externally and so in visible behaviours, meaning that specification and intuitively correct implementation cannot always be related using a process algebraic approach. For example, if an implementation allows additional, visible interleaving of events which is not possible for its specification, then process algebraic refinement cannot be used directly to show correctness even if this additional interleaving does not give rise to interference.

The usual approach taken to solve this problem is, before approaching verification, to compose component processes into networks such that all evidence of decomposition of behaviours and of relaxation of atomicity becomes internal. Thus, we are able to verify the composite implementation process against the corresponding composite specification process using a standard notion of refinement. However, such an approach has implications for the state explosion problem if we are to use model-checking in order to verify correctness. It also has implications in terms of verifying in isolation the correctness of a component process which is used in a number of different contexts. For example, if that component process contains bugs, some of them may only be discovered when it is deployed in a particular network. On a similar note, modifications to certain component processes within a particular network would necessitate the re-verification of the whole network and that could reveal bugs in processes other than those where the changes were made.

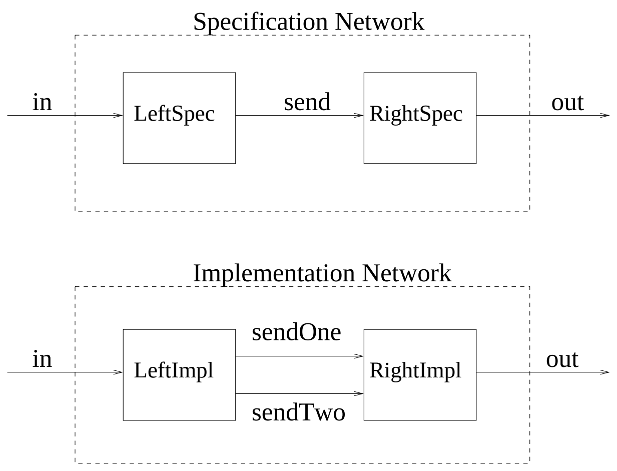

Figure 3: Relaxing atomicity for greater efficiency

We now sketch a small example which illustrates some of these issues. It constitutes an instance where standard CSP refinement fails to show correctness but we do have correctness when placed in context; it will also be useful to us as we introduce refinement-after-hiding. Figure 3 shows a specification network and a corresponding implementation network, each consisting of two component processes where the communication between those processes is hidden. The abstract communication between the two processes in the specification network has been rendered in the implementation network in a more concrete fashion using a particular communication protocol. More specifically, the specification network consists of *LeftSpec* and *RightSpec*, two single slot buffers which store and transmit messages, and which are connected by a channel *send*. The specification network, where communication on *send* is hidden, thus gives us a 2-slot buffer. In the implementation network, messages received by *LeftImpl* on channel *in* are split up into packets before being forwarded on either channel *sendOne* or channel *sendTwo*. *RightImpl* then reconstitutes the packets into messages and outputs them on channel *out*. Thus, both behaviour decomposition — the move from transmitting messages as monolithic entities to transmitting them as packets — and relaxation of atomicity — in the implementation, two packets from the same message may be transmitted simultaneously — have occurred in the move from specification to implementation component processes.

We can envisage full CSP definitions for *LeftSpec*, *RightSpec*, *LeftImpl* and *RightImpl* such that the implementation *network*, where behaviour on the channels *sendOne* and *sendTwo* is hidden, has the same behaviour in a CSP sense as the specification *network*, where behaviour on channel *send* is hidden. (The implementation network is given by $\mathit{LeftImpl} \otimes \mathit{RightImpl}$ and the specification network by $\mathit{LeftSpec} \otimes \mathit{RightSpec}$.) Obviously, this is something which could be shown using standard CSP refinement. However, we could not use standard CSP refinement to show that *LeftImpl* implements *LeftSpec*, nor that *RightImpl* implements *RightSpec* since we have used (visible) behaviour decomposition and (visible) relaxation of atomicity in the move from specification component process to corresponding implementation component process.

In order to address this problem, we note that the standard notions of correctness have to effectively displace the boundary at which behaviours are observed and must consider correctness only when a particular component process has been placed in a context which hides or makes internal all direct evidence of behaviour decomposition and relaxation of atomicity. For example, when showing the correctness of $\mathit{LeftImpl} \otimes \mathit{RightImpl}$ using standard CSP refinement, we cannot consider directly the behaviour of either *LeftImpl* or *RightImpl* in isolation: it is only when they have been composed — i.e. when they have been placed in context — that we can consider correctness. Thus, we do not allow our hypothetical observer — who compares implementation to specification processes on the basis of what can be observed of them — to look directly at component processes. Rather, the observer may consider directly only *networks* of processes. We take advantage of this intuition and place it on a more formal footing so that we *are* able to consider in isolation the correctness of component processes which, for example, have been derived through the (visible) relaxation of atomicity. (See [3] in this volume for a more detailed discussion of the issue of observability.)

4.2 Refinement-after-hiding

In [2], building on the work in [4, 12, 13], we presented a notion of refinement for CSP called refinement-after-hiding. This allows for a looser matching of implementation behaviours against specification behaviours by introducing a more discerning notion of visibility: events are regarded as visible only if they may be visible in any final network of which the implementation may be a component (it is in this sense that an observer may consider directly only networks of processes). Moreover, those events which are not regarded as visible in this weaker sense *must* be hidden during the construction of any such network. The former type of events are referred to as *finally visible* events and the latter as *finally invisible* events; in general, finally visible events are the same for both specification and implementation component processes. (In the example from Figure 3, events on channels *in* and *out* would be regarded as finally visible and those on channels *send*, *sendOne* and *sendTwo* as finally invisible.) The events of any implementation are partitioned according to these two categories; by the definition of refinement-after-hiding, finally visible events must be matched exactly in the corresponding specification, while finally invisible events may be matched much more loosely. For example, if $\langle in.m, a_1, \ldots, a_n \rangle$ is a trace of *LeftImpl*, where $a_1, \ldots, a_n$ are used to implement the sending of message m, then the corresponding trace of *LeftSpec* would be $\langle in.m, send.m \rangle$. In practice, we use an interpretive mapping in order to transform implementation behaviours and then check for containment of those interpreted behaviours in the behaviours of the specification. This use of an interpretive mapping and a looser matching of finally invisible implementation events reflects the idea that we are no longer able to observe directly the behaviours of our component implementation processes.

A full description of how refinement-after-hiding is defined for all three CSP semantic models may be found in [2]. Here, we simply give enough detail so that the verification of the 4-slot in the traces model which is described in Section 6 may be understood. Consider an implementation process, Q, and the corresponding specification process, P. In order to consider the correctness in the traces model of Q with respect to P, we define a strict, monotonic *extraction*[4] mapping, $extr_Q$, which is used to interpret the traces of Q as traces of P: when applied to $t \in \tau Q$, $extr_Q$ gives back a trace which engages in events from the specification process P. The fact of monotonicity means that receiving more information regarding a particular implementation trace cannot reduce our knowledge about the corresponding specification trace. We then verify that, for every $t \in \tau Q$, $extr_Q(t) \in \tau P$ and denote this as $Q \sqsupseteq^{rah} P$, i.e. that Q refines-after-hiding P. Note that $extr_Q$ may be defined in practice in terms of a number of component mappings and also that it behaves as the identity mapping when applied to behaviours consisting of finally visible events.

We are able to reclaim standard CSP (trace) refinement from refinement-after-hiding using the following two conditions (*Fvis* is used here to denote the set of finally visible events):

RAH1 If $\alpha Q \subseteq Fvis$ and $Q \sqsupseteq^{rah} P$, then $Q \sqsupseteq P$.

RAH2 If $Q_i \sqsupseteq^{rah} P_i$ for $i = 1, 2$, then $(Q_1 \otimes Q_2) \sqsupseteq^{rah} (P_1 \otimes P_2)$.

[4] The term "extraction mapping" is used to indicate the fact that we *extract* the behaviours of the specification from those of the implementation.

In other words, by RAH2 the notion of refinement-after-hiding is compositional: in order to infer the correctness of a network we need only verify the correctness of individual component processes. Moreover, by RAH1, if Q refines-after-hiding P and Q engages only in finally visible events, then Q is correct with respect to P according to standard CSP trace refinement. In terms of the example processes from Figure 3, we could verify that $\mathit{LeftImpl} \sqsupseteq^{rah} \mathit{LeftSpec}$ and $\mathit{RightImpl} \sqsupseteq^{rah} \mathit{RightSpec}$ using suitable extraction mappings. By RAH2, we would then be able to infer that

$$(\mathit{LeftImpl} \otimes \mathit{RightImpl}) \sqsupseteq^{rah} (\mathit{LeftSpec} \otimes \mathit{RightSpec})$$

and so, by RAH1, that

$$(\mathit{LeftImpl} \otimes \mathit{RightImpl}) \sqsupseteq (\mathit{LeftSpec} \otimes \mathit{RightSpec}).$$

(Recall that $\mathit{LeftImpl} \otimes \mathit{RightImpl}$ engages only in finally visible events, since all other events have been hidden.)

Implicit in condition RAH2 are the restrictions imposed by refinement-after-hiding on the implementation networks which we may build and verify. In particular, we may join component processes together using only the network composition operator and such compositions must synchronize on all shared events. The use of network composition means that, after synchronizing in parallel on the set of shared events, we hide those events and so make them internal. This use of hiding is what makes it possible for us to build a network which engages only in finally visible events: it is assumed that all processes which exhibit visible evidence of relaxation of atomicity or of behaviour decomposition will be composed with other processes in such a way that, as we build our final implementation network, that visible evidence will be hidden. Condition RAH1 encapsulates the idea that we may only observe directly the behaviour of processes which engage only in finally visible events. In general, the component implementation processes which we define will not obey this restriction and so may not be observed directly; however, final networks built using these component processes and the network composition operator will obey this restriction and so can be observed directly. Note also that counterparts to RAH1 and RAH2 hold for the notions of refinement-after-hiding defined in [2] in the stable failures and failures divergences models.

Using refinement-after-hiding, we *are* able to show that the 4-slot is a valid implementation of a register, and the extraction mapping used for this purpose is described in Section 6. In order to carry out the verification, we take advantage of a means described in [2] for automatic verification of refinement-after-hiding. This approach uses the industrial tool FDR2 ([6, 19]) and so a model-checking approach. Note that some small additional machinery is required when we verify the correctness of any component process which communicates with the 4-slot and this issue is discussed further in Section 7.

5 Modelling the 4-slot and register in CSP

In order to verify the 4-slot using our notion of refinement-after-hiding, we need to render both it and the register in CSP. In our modelling of the 4-slot, we make some simplifying assumptions. Firstly, we assume that reads and writes of the variables *latest*, *reading* and *slot* are atomic. In Simpson's view, such an assumption is justified and, once such a property has been assumed, a number of authors — including Simpson himself — have shown that the 4-slot behaves as if reads and writes to its data array were actually atomic (see Section 9.1 for further discussion of this issue). We also take advantage of this latter fact and so, in our CSP model of the 4-slot, reads and writes to the data array are atomic; this simplifies considerably the extraction mapping which is needed to show correctness. As a result of this latter simplification, our verification as presented here is essentially concerned with the *ordering* of the data values which may be transmitted by the 4-slot and with showing that they implement in some sense the data transmissions which are possible for the register.

As indicated at the end of Section 4, the means of automatic verification which we use in the verification of the 4-slot uses the model-checker FDR2. As a result, it is necessary to choose a concrete data type to be written to and read from (our CSP models of) both the register and the 4-slot. We have chosen a subset of the integers, namely $\{0..5\}$ — which we shall refer to as *dataint* — because they are a basic type. Since we are carrying out model-checking, it is also necessary to choose a finite type and the limits of the hardware on which FDR2 was run dictated the size of the type used.

Six basic processes are used in the CSP representation of the 4-slot, full details of which can be found in [2]. Here, we simply give enough detail so that the interpretive mapping defined in Section 6 may be understood. There is a process to represent each of the global variables *latest*, *reading* and *slot*, while another process represents the data array. Finally, *Reader* and *Writer* from Figure 4 are used to represent the read and write procedures respectively. In *Writer*, the channel *cw* is used to represent a call to the write procedure and it can communicate any value of type *dataint*: on the occurrence of $cw.v$, the value v is stored in *item* and is subsequently written into the data array. The channel *rw* is a simple channel with no explicit data content and is used to represent a return from the write procedure. Channel *cr* in *Reader* is used to represent a call to the read procedure. Channel *rr* is used to represent a return from the read procedure and can communicate any value of type *dataint*; the occurrence of $rr.v$ indicates that the value v has been returned to the calling process.

Note that, in each of *Writer* and *Reader*, the events used to represent the respective procedure bodies — i.e. all events other than the call and return events — are taken directly from the description of the 4-slot given in Figure 1. This is simply to make easier for the reader the task of relating the definitions

Defining the write procedure:

```
Writer = cw?item→
            pair := not(reading)→
            index := not(slot[pair])→
            data[pair,index] := item→
            slot[pair] := index→
            latest := pair→
            rw→ Writer
```

Defining the read procedure:

```
Reader = cr→
            pair := latest→
            reading := pair→
            index := slot[pair]→
            item := data[pair,index]→
            rr.item→ Reader
```

Figure 4: Defining the 4-slot procedures

of *Writer* and *Reader* to the definition given in Figure 1 and, in [2], *Writer* and *Reader* are defined using events given in standard CSP syntax. We use *IE* to denote the set of events used to represent the procedure bodies and note that they are essentially those events from the 4-slot which are *internal*. Finally, note that the description of the 4-slot in Figure 1 implies that there is a single action — i.e. *read := data[pair,index]* — which reads the relevant item from the data array and returns it to the calling program. In the definition of *Reader*, we use two separate actions to provide this functionality: this change makes cleaner the CSP definition of the relevant processes and, in any case, it is unlikely that *read := data[pair,index]* would execute atomically in practice. Moreover, since the verification of the 4-slot is successful then the additional interleaving which may have been introduced as a result of this change does not have any negative impact.

We finally define the CSP representation of the 4-slot. *FullFSlot*, defined as follows,

$$\mathit{FullFSlot} \triangleq ((\mathit{latest} \mid\mid\mid \mathit{reading} \mid\mid\mid \mathit{slot} \mid\mid\mid \mathit{data}) \parallel \mathit{Writer}) \parallel \mathit{Reader}$$

combines the processes to represent the global variables of the 4-slot mechanism with the processes which provide the functionality of the procedure calls.

- $\mathit{Register} \triangleq \mathit{Reg}(0)$
- $\mathit{Reg}(x) = \mathit{read}.x \rightarrow \mathit{Reg}(x) \;\square\; \mathit{write}?y \rightarrow \mathit{Reg}(y)$

Figure 5: A CSP version of the register

Moreover, all events from *IE* — i.e. all internal events used to implement the procedure bodies — are left visible. This representation of the 4-slot will be used for verification as the internal events must be left visible for this purpose: see Section 6 for further details. The process

$$\mathit{FSlot} \triangleq \mathit{FullFSlot} \setminus \mathit{IE}$$

then gives the CSP representation of the actual 4-slot. The only visible events in which it engages are call and return events.

The CSP version of the register is presented in Figure 5. Since read and write procedures may not occur concurrently in the register and because there is no longer any need to ensure that the 4-slot and register representations in CSP have the same interface, we represent reads and writes respectively as single events. (The channels *read* and *write* are assumed to communicate values from *dataint*.) The variable *data* from Figure 2 is represented as a parameter to the register process and, since individual CSP events occur instantaneously and cannot occur concurrently, we are guaranteed to have atomic transfers of data.

6 Verifying the 4-slot

We now move on to present the extraction mapping, extr_{FS}, which is used in the verification of the 4-slot. Before doing this, we first note that the verification itself is carried out with regard to *FullFSlot* — i.e. the 4-slot with all of its internal events left visible — rather than with regard to *FSlot*. This is simply because the traces of *FSlot* — consisting only of call and return events — do not give enough information to allow us to reclaim the traces of *Register* via an interpretive mapping. This is similar to the approach followed by *linearizability*, where the internal events used to implement method bodies in concurrent objects are made visible during verification (see [10] and Section 9.2). Section 7 then shows how we reclaim the correctness of *FSlot*. Since extraction mappings are strict and monotonic, we define extr_{FS} in the following way. By the strictness property, $\mathit{extr}_{FS}(\langle\,\rangle) = \langle\,\rangle$. For any trace $t \in \tau \mathit{FullFSlot}$ such that $t \neq \langle\,\rangle$, then $t = u \circ \langle a \rangle$ for some trace u and event a. By the monotonicity property, we know that $\mathit{extr}_{FS}(t) = \mathit{extr}_{FS}(u) \circ r$ for some trace r, where r may be the empty trace, $\langle\,\rangle$ (we say that we extract to r — or to the event/s which comprise r — on the occurrence of a). Thus, in order to define the mapping, we take $u \circ \langle a \rangle$ and must decide which high-level event or events from the register the occurrence of a after u represents. (Note we assume all events from *FullFSlot* are finally invisible.)

In order to do this, there are two main factors which must be taken into account. Firstly, we must extract to exactly one high-level write for each low-level write which occurs[5] and, similarly, we must extract to exactly one high-level read for each low-level read which occurs (moreover, the extracted event must transmit the same data value as the corresponding low-level read or write). For example, for each sequence of events used to represent a particular low-level write in $t \in \tau \mathit{FullFSlot}$, on the occurrence of exactly one of those events we will extract to the occurrence of a high-level write which transmits the same data value as the low-level write and on the occurrence of the rest of the events which comprise that low-level write we will not extract to anything. This restriction obviously makes sense intuitively; moreover, it is necessary when one comes to consider the correctness of environment processes with which *FSlot* might be composed (see [2] for further details). Note, however, that we never extract to a high-level event on the occurrence of a call or return event. This reflects the fact that such events do not provide any of the functionality of the procedures with which they are associated. The second factor to take into account when defining extr_{FS} is that we are mapping traces of *FullFSlot* to those of *Register*. This means that any high-level read event to which we extract must transmit the same data value as the last high-level write to which we extracted.

Mimicking the behaviour of the register after application of the mapping is complicated by two main factors (detail on how the relevant situations may arise can be found in Sections 6.2 and 6.3):

- A low-level read may actually read data written into *FullFSlot* by a low-level write that has not yet completed (that is, it has not yet updated both relevant control variables to fully indicate where it wrote the data).[6]

- The slot and pair from which data is to be read on a particular low-level read may be fully determined before their identity has been discovered from the relevant control variables.

[5] We use *low-level write* to mean a particular execution of the events in *FullFSlot* which implement the write procedure of the 4-slot; similarly, *low-level read* is used to mean a particular execution of the events in *FullFSlot* which implement the read procedure of the 4-slot. A high-level write is then simply an event occurring on channel *write*; a high-level read is an event occurring on channel *read*.

[6] We regard a particular low-level read or write as having completed when it has performed its last event *prior* to the relevant return event. At this point, the relevant procedure execution has completed all of its work and the effects of that work will be visible to subsequent procedure executions. Thus, a low-level write effectively completes when it updates the control variable *latest* and a low-level read completes when it reads the value from the appropriate slot in the data array.

The first point has the consequence that we cannot always extract to a high-level write event at exactly the point at which the low-level write has completed (i.e. we cannot always extract on the occurrence of the event which updates the variable *latest*). If we were to do this, the reader side may have already read and extracted the value written and, at the specification level, we will get a trace which apparently manages to read a value before it has been written. However, we must also be careful not to extract the current write yet if the reader could still read the value written by the previous write.

As soon as it is fully determined which slot and pair the reader will read from, the value to be read is also fully determined. This is because the writer will not be able to access the relevant slot of the data array until this read has finished, since the 4-slot maintains the property of *data coherence*.[7] As a result, by the second point given above, we may know exactly which value the reader is to read *before* it has completed interrogating the necessary control variables. And we must extract to a high-level read as soon as the value to be read is determined: if we did not do this, the reader could wait until an arbitrary number of further writes had been completed and extracted and only then complete and extract this read. This would give the apppearance of reading an old value and so of having more memory than the single slot of the register.

Before proceeding, it is also necessary to observe that the event on the occurrence of which we actually extract to a high-level write is not always the same and depends on the way in which low-level reads and writes have been interleaved; a similar comment applies with regard to extraction to high-level read events.

6.1 An annotated 4-slot

Figure 6 presents an annotated version of the 4-slot mechanism, using numbers to indicate positions within the read and write procedures (recall that "writer" is used as shorthand for "write procedure" and "reader" as shorthand for "read procedure"). These annotations are used both in the definition of the extraction mapping and in its explanation. The call and return events do not appear in this annotated version of the mechanism since their occurrence does not provide any of the information needed to define the extraction mapping and we never extract to anything on their occurrence. We therefore assume that the mechanism is also at position 1 with respect to a particular procedure when it is immediately prior to a return from that procedure or immediately prior to a call to that procedure.

Before presenting the mapping, we consider in greater detail the points at which a particular low-level read or write should be extracted. Note that the

[7] Data coherence is preserved if and only if a reader and writer may not simultaneously access the same data slot in the 4-slot. It is this property which allows us to make the simplifying assumption that reads and writes to the 4-slot data array are atomic. (See discussion of related work in Section 9.)

The Writer:

```
        begin
1           $pair := not(reading);$
2           $index := not(slot[pair]);$
3           $data[pair, index] := item;$
4           $slot[pair] := index;$
5           $latest := pair;$
        end;
```

The Reader:

```
        begin
1           $pair := latest;$
2           $reading := pair;$
3           $index := slot[pair];$
4           $item := data[pair, index];$

        end
```

Figure 6: Simpson's 4-slot mechanism annotated

following discussion is intended to provide intuition regarding the definition of the mapping, rather than an exhaustive justification of its every clause.

6.2 Considering the writer side in more detail

We cannot extract a write by positions 2 and 3 in the writer, since the relevant value has not yet been written into the mechanism. If the writer is at position 4, it is impossible for the reader to read what has just been written since data coherence is preserved (again, see discussion of related work in Section 9). Finally, we must have extracted once we have executed *latest* := *pair*: by this point, any new execution of the read procedure can read the value which has just been written into the mechanism by the current write. This means that, if we have not already extracted on the current write, we must do so on the occurrence of $latest := pair$.

We therefore consider position 5 in the write procedure and the conditions under which we need to have extracted a high-level write event by the time that we reach it: in other words, when do we extract a write event on the occurrence of $slot[pair] := index$. In general, we need to have extracted by this point if the reader already knows, or can discover without any further writer action, the pair into which the writer has just written. If this is the case, the reader can proceed to find out which slot in the pair was written to and so read and extract the value just written. This can happen in the following circumstances:

- If we have already extracted in the reader and the global variable *latest* stores the same value as the variable *pair* in the writer. (The value of *pair* in the writer tells us the pair which the writer has just written to.)
- If we are at position 1 in the reader and the global variable *latest* stores the same value as the variable *pair* in the writer.
- If we are at position 2 or 3 in the reader but have not extracted yet, and the value of *pair* in the reader is the same as the value of *pair* in the writer. (In the corresponding conditional branches in the extraction mapping definition given in Figure 7, we do not actually state explicitly the requirement that the reader has not yet extracted. This is simply because, if the value of *pair* in the reader is the same as the value of *pair* in the writer, then the reader cannot have extracted yet (this issue is expained more fully in [2])).

Note that, by the detail below, the reader must always have extracted by the time that it reaches position 4.

6.3 Considering the reader side in more detail

Recall that we will extract to a high-level read event as soon as we are certain of the pair and slot combination from which we will read on the current low-level read.

By position 2 in the reader, we know the pair we must read from. In order for it to be fully determined by this point the slot from which we will read, it has to be the case that the writer is unable to write again to this pair before we have completed the current read. (If the writer could write to this pair again, it would first write to the other slot of the pair, to which element the reader could then be directed.) If the writer is to be unable to write to this pair, it is necessary that the value of *pair* in the reader is the same as the value of *reading.*

We therefore have to have extracted a read by position 2 in the reader — that is, extracted on the occurrence of *pair* := *latest* — in the following circumstances:

– If the writer is at position 1 or position 5, and *pair* in the reader has the same value as *reading*.

– If the writer is at positions 2, 3 or 4, the value of *pair* in the writer is *not* the same as the value of *pair* in the reader and *pair* in the reader has the same value as *reading*.

In order to check these conditions in practice, we use the value stored in *latest* in place of that stored in *pair* in the reader: the conditions must be checked at position 1, when *pair* has not yet been updated with the value of *latest*.

By position 3, we know the pair we will read from and have also indicated this to the writer. We have to have extracted by position 3 if the writer is at position 1 or position 5 or if the value of *pair* in the reader is not the same as the value of *pair* in the writer. These conditions are essentially the same as those given for position 2, when we bear in mind the fact that we have just assigned the value of *pair* in the reader to *reading*.

Finally, we must always have extracted by position 4 since, at this point, we know both the slot and pair of the data item which we shall read.

It can be seen from the above discussion that the position of the writer plays a role in whether or not we extract a read event. And, in fact, the writer *moving* to position 5 may necessitate the extraction of a read event. This means that the event *slot*[*pair*] := *index* will, in some cases, be extracted to both a read *and* a write event. This can be seen in the definition of the extraction mapping in Figure 7.

6.4 Presenting the extraction mapping

We now proceed to define $extr_{FS}$. Before giving the definition of this mapping, it is necessary to introduce some auxiliary notation.

– For any trace $t \in \tau FullFSlot$:
  - we take $exR(t) = yes$ if and only if we have already extracted a *read* event during the current low-level read and take $exR(t) = no$ otherwise.
  - we take $exW(t) = yes$ if and only if we have already extracted a *write* event during the current low-level write and have $exW(t) = no$ otherwise.

– *late* gives the current value stored by the control variable *latest*.

– *rp* gives the current value of the variable *pair* in the reader and *wrp* gives the value of the variable *pair* in the writer.

- $rPos$ gives the current position of the reader and $wPos$ gives the current position of the writer.
- $rdng$ gives the value currently stored in the variable $reading$.
- $slot[i]$ gives the value currently stored at position i in the array $slot$.
- $wVal$ gives the last value written into the mechanism.
- $data[i][j]$ gives the data value stored by the mechanism in pair i, slot j.

We then have that $extr_{FS}(\langle\,\rangle) \triangleq \langle\,\rangle$ and, for $t \circ \langle a\rangle \in \tau FullFSlot$,

$$extr_{FS}(t \circ \langle a\rangle) \triangleq extr_{FS}(t) \circ u,$$

where u is as defined in Figure 7. A brief comment is required on the clauses used for the extraction of write events, since at first sight some of them may not appear to be mutually exclusive. That they are mutually exclusive follows from the fact that, if the reader is at position 1, then it cannot yet have extracted, and, as observed above, the reader cannot yet have extracted if the value of $pair$ in the reader is the same as the value of $pair$ in the writer.

Using the mapping $extr_{FS}$ and the means of automatic verification described in [2], we were able to verify that $FullFSlot \sqsupseteq^{rah} Register$. We also show in [2] that $FullFSlot$ refines-after-hiding $Register$ in the stable failures and failures divergences models. Verification of refinement-after-hiding in both of these models reuses the mapping $extr_{FS}$, while the additional effort required to show correctness is minimal. Thus, we have that the 4-slot is a valid implementation of the register.

7 Verifying processes which communicate with the 4-slot

We face certain difficulties when attempting to use standard refinement-after-hiding — the details of which were sketched in Section 4 — to show the correctness of any process with which the 4-slot might be composed. This is for a number of reasons. To begin with, we have verified $FullFSlot$, a version of the 4-slot with all internal events left visible, while any environment process with which the 4-slot would be composed would engage only in the relevant call and return events. This causes problems since the definition of refinement-after-hiding requires that the extraction mapping $extr_{FS}$ also be used when interpreting the behaviours of any such environment process. In addition, although we verify $FullFSlot$, it is the correctness of $FSlot$ when placed in context in which we are interested. Finally, the fact that a single extraction mapping — i.e. $extr_{FS}$ — is used to interpret the behaviours of $FullFSlot$ causes problems with respect to

$$
u \triangleq \begin{cases}
\langle write.wVal \rangle & \text{if } (a \text{ is } slot[pair] := index) \wedge \\
 & \quad (rPos = 1 \ \wedge \ late = wrp) \\
\langle write.wVal \rangle & \text{if } (a \text{ is } slot[pair] := index) \wedge \\
 & \quad (exR(t) = yes \ \wedge \ late = wrp) \\
\langle write.wVal \rangle & \text{if } (a \text{ is } slot[pair] := index) \wedge \\
 & \quad (rPos = 2 \ \vee \ rPos = 3) \wedge \\
 & \quad (rp = wrp \ \wedge \ rp \neq rdng) \\
\langle write.wVal \rangle & \text{if } (a \text{ is } latest := pair) \wedge \\
 & \quad (exW(t) = no) \\
\langle write.wVal, read.wVal \rangle & \text{if } (a \text{ is } slot[pair] := index) \wedge \\
 & \quad (rPos = 2 \ \vee \ rPos = 3) \wedge \\
 & \quad (rp = wrp \ \wedge \ rp = rdng) \\
\langle read.(data[late][slot[late]]) \rangle & \text{if } (a \text{ is } pair := latest) \wedge \\
 & \quad (wPos = 1 \ \vee \ wPos = 5) \wedge \\
 & \quad (late = rdng) \\
\langle read.(data[late][slot[late]]) \rangle & \text{if } (a \text{ is } pair := latest) \wedge \\
 & \quad (wPos = 2 \ \vee \ wPos = 3 \vee \\
 & \quad\ wPos = 4) \wedge \\
 & \quad (wrp \ ! = late \ \wedge \ late = rdng) \\
\langle read.(data[rp][slot[rp]]) \rangle & \text{if } (a \text{ is } reading := pair) \wedge \\
 & \quad (exR(t) = no) \wedge \\
 & \quad (wPos = 1 \ \vee \ wPos = 5 \vee \\
 & \quad\ wrp \ ! = rp) \\
\langle read.(data[rp][slot[rp]]) \rangle & \text{if } (a \text{ is } index := slot[pair]) \wedge \\
 & \quad (exR(t) = no) \\
\langle \rangle & \quad \text{otherwise}
\end{cases}
$$

Figure 7: Defining $extr_{FS}$

compositionality: in effect, we would need to compose *FullFSlot* with the relevant environment process before we could verify the latter. (A full account of these problems may be found in [2].) Thus, work is described in [2] that builds on and uses the standard notion of refinement-after-hiding and which does allow us to verify processes which communicate with the 4-slot.

7.1 A solution to the problem

We first impose on any network to be verified the restriction that any component process which is part of that network can communicate with at most a single instance of the 4-slot. Any such process may, of course, communicate with other processes and the network as a whole may contain multiple 4-slot instances. Consider, then, an implementation process, IW, which communicates with the 4-slot via the write procedure; the corresponding specification process, SW, communicates with the register using channel *write*. Consider also an implementation process, *IR*, which communicates with the 4-slot via the read procedure; the corresponding specification process, *SR*, communicates with the register using channel *read*. Prior to verification, IW is composed in parallel with a small additional process to deal with the fact that it only engages in call and return events from the 4-slot write procedure while all internal events are left visible when the 4-slot itself is verified. We then (attempt to) show that the resulting process refines-after-hiding SW. The verification of *IR* is treated in a similar manner. Assuming that the respective verifications of IW and *IR* are successful, a result is given in [2] which lets us infer that

$$(IW \otimes \mathit{IR}) \otimes \mathit{FSlot} \sqsupseteq^{rah} (SW \otimes \mathit{SR}) \otimes \mathit{Register}.^{8}$$

Thus, we are able to verify the correctness of processes which communicate with the 4-slot, we have reclaimed for such processes and the 4-slot the facility for compositional verification given by Rah2 from Section 4, and we have derived a correctness result stated in terms of *FSlot* rather than in terms of *FullFSlot*. (Full details on the approach described here, and on the relevant result and its proof, can be found in [2].)

The reader might speculate at this point that, by imposing the restriction we do, we are ruling out a class of systems that might expose errors in the 4-slot, which errors are able to remain uncovered with our more restricted networks. We make two points in relation to this. Firstly, the restriction is imposed for a reason connected solely to the technical treatment of refinement-after-hiding, namely that relevant environment processes have to be augmented before they

[8] Note that $IW \otimes \mathit{IR}$ synchronizes in parallel with *FSlot* on the read and write procedure call and return events and then all communication between the two processes is hidden. Note also that an equivalent result holds in the stable failures and failures divergences models.

– $WriteEnviron = in?val \rightarrow cw.val \rightarrow rw \rightarrow WriteEnviron$

– $ReadEnviron = cr \rightarrow rr?val \rightarrow out.val \rightarrow ReadEnviron$

– $FourSlotEnviron \triangleq WriteEnviron \mid\mid\mid ReadEnviron$

Figure 8: Environment processes for *FSlot*

are verified. Secondly, we do not restrict the way in which any relevant process may interact with the 4-slot instance that it does communicate with: simply allowing individual environment processes to communicate with multiple instances of the 4-slot would not change the range of interactions possible with any single instance. In other words, our environment processes are still general enough that they may exercise all possible behaviours of the 4-slot. Nonetheless, further work is needed to lift the restriction imposed here on the form of implementation networks which use the 4-slot and which can be verified using refinement-after-hiding.

7.2 Example environment processes

We now introduce some simple environment processes with which *FSlot* might be composed. This allows us to present (the result of) a simple verification using the approach described in this section, which then facilitates a consideration of the sense in which the 4-slot may be regarded as "correct" (see discussion in Section 8). Figure 8 describes these environment processes: their composition, *FourSlotEnviron*, may take a value from *dataint* on the channel *in*, call the write procedure from *FSlot* with this value as a parameter and then wait for the write procedure to return; it may also call the read procedure in *FSlot*, receive a value from *FSlot* via the return of that procedure and then output the result on channel *out*. (Note that the channels *in* and *out* used here are assumed to be different to those used in the definition of the processes from Figure 3.) Figure 9 then defines corresponding specification processes with which *Register* might be composed: they are essentially a pair of single-slot buffers to be placed on the read and write channels respectively of the register. In respect of the definition of *FourSlotEnviron*, we observe that $WriteEnviron \mid\mid\mid ReadEnviron$ is syntactically equivalent to $WriteEnviron \otimes ReadEnviron$, since *WriteEnviron* and *ReadEnviron* do not have any events in common. A similar comment applies with respect to *RegisterEnviron*. Note also that the events on channels *in* and *out* are regarded as finally visible.

Using the process of verification described in this section, we show that *WriteEnviron* composed in parallel with the necessary additional process refines-

- $\mathit{RegWriteEnviron} = \; \mathit{in}?\mathit{val} \rightarrow \mathit{write}.\mathit{val} \rightarrow \mathit{RegWriteEnviron}$
- $\mathit{RegReadEnviron} = \; \mathit{read}?\mathit{val} \rightarrow \mathit{out}.\mathit{val} \rightarrow \mathit{RegReadEnviron}$
- $\mathit{RegisterEnviron} \triangleq \; \mathit{RegWriteEnviron} \;|||\; \mathit{RegReadEnviron}$

Figure 9: Corresponding environment processes for the register

after-hiding *RegWriteEnviron* and we also show that a similarly augmented *ReadEnviron* refines-after-hiding *RegReadEnviron* (full details may be found in [2]). Using the result described in Section 7.1, we are therefore able to infer that

$$\mathit{FourSlotEnviron} \otimes \mathit{FSlot} \sqsupseteq^{rah} \mathit{RegisterEnviron} \otimes \mathit{Register}.$$

Hence, since *FourSlotEnviron* ⊗ *FSlot* engages only in finally visible events — namely those on channels *in* and *out* — and by RAH1 from Section 4, we have that

$$\mathit{FourSlotEnviron} \otimes \mathit{FSlot} \sqsupseteq \mathit{RegisterEnviron} \otimes \mathit{Register}.$$

8 Discussion

The behaviours of the 4-slot could not have been related to those of a register using standard CSP refinement. However, by displacing the boundary at which we directly observe behaviour from the level of component processes to the level of networks of processes we *have* been able to establish the necessary relationship. This fact is in evidence as we show in Section 7 that *FourSlotEnviron* ⊗ *FSlot* refines *RegisterEnviron* ⊗ *Register* according to standard CSP refinement. In essence, we may build an implementation of an (albeit restricted) network which communicates data internally using a register by modifying in a suitable fashion the necessary communication interface and then substituting the 4-slot for the register. This is a significant result since the fact that the 4-slot has more than a single memory slot may be made apparent to a user: a read may begin, interrogate the necessary control variables and then wait for an arbitrary number of writes before completing, thereby appearing to read an old value. It is not immediately clear that such behaviour should be permissible in any valid implementation of a register, which has only a single memory slot. Nonetheless, the result described in Section 7 indicates that, once the 4-slot and register have been placed in suitable contexts and thereby all relevant communication hidden, it is effectively impossible for an observer to distinguish between them.

This raises the question of why, for example, *FourSlotEnviron* ⊗ *FSlot* should refine *RegisterEnviron* ⊗ *Register* in the traces model. From the definition of the extraction mapping, extr_{FS}, it can be seen that read and write procedures appear to take effect at a single point between the occurrence of their call and return events. Indeed, it this possibility of abstracting an operation with duration — in this case a procedure execution — to a computational effect at a single instant which is crucial to our ability to define an extraction mapping (see also the discussion of linearizability in Section 9.2 below). However, this means that, if a read and write procedure are executing in parallel, we have no way of knowing which will "take effect" first: i.e. will the read return the result of this write or of an earlier write. In other words, since an observer of the 4-slot can see only call and return events — i.e. that observer can only know when a particular procedure execution has begun and when it has finished, rather than knowing anything about its progress during that execution — there will be a degree of non-determinism in the behaviour of the 4-slot as it appears to that observer. In contrast, there is no such non-determinism in evidence in the behaviour of the register, and reads and writes may not proceed in parallel in that process. However, once *Register* has been composed with *RegisterEnviron* and thereby all inter-process communication hidden, all an observer can see are events on channels *in* and *out*; similarly, once *FSlot* has been composed with *FourSlotEnviron*, all that can be observed are events on channels *in* and *out*. This adds non-determinism in the specification, since all we know of the specification network is whether or not a value has been provided for writing into the register or whether a value read from the register has just been output to the outside world. In other words, we cannot observe when a value is actually written into or read from the register: thus, to our hypothetical observer, it is no longer clear when exactly those writes and reads take effect whose occurrence is implied by the occurrence of events on channels *in* and *out*. Moreover, placing the 4-slot in context and hiding inter-process communication does not seem to result in any additional non-determinism: we do not know when exactly reads and writes to the 4-slot actually take effect but then we did not know this previously either. Thus, disallowing direct observation of the 4-slot and register and allowing us only to observe their behaviours when placed in appropriate contexts seems to have added sufficient non-determinism in the specification such that it matches the non-determinism already exhibited by the 4-slot.

The above discussion is based on intuition regarding what happens when the 4-slot is composed with the simple environment given by *FourSlotEnviron* and further work is needed to explore these issues more formally. Moreover, the main focus in the above discussion is on the ordering of reads and writes in the 4-slot and register respectively. Just as important is the way in which calls and returns are dealt with in environment processes: for example, it is possible to define an environment process which can detect the fact that the 4-slot is exhibiting additional concurrency not in evidence in the register. This can be done by allowing the occurrence of two consecutive procedure calls in the environment without any associated returns — i.e. something which can only happen because the 4-slot allows reads and writes to proceed concurrently — to take us to a process algebraic term which allows the execution of some distinguished event denoting an error or which deals with output from the 4-slot in a different way to the way it is dealt with after sequentially-executing procedure calls (see discussion in [3] in this volume). The type discipline of *uniform receptiveness* described in [21] deals with similar issues and it is necessary to explore further the role they play in relation to relaxation of atomicity.

9 Related work

9.1 Standard approaches to verifying the 4-slot

The standard approach taken in the literature is *not* to consider correctness of the 4-slot with respect to a register or, indeed, any explicit specification process. Rather, certain intuitive properties are identified which it is felt must hold of the 4-slot[9] if it is to work in an acceptable manner (see, for example, [5]). For example, the property of *data coherence* is preserved if and only if a reader process and a writer process may not simultaneously access the same slot in the 4-slot data array. It is essentially a mutual exclusion property and is used to guarantee that the 4-slot behaves as if reads and writes to the data array were actually atomic. Other properties are concerned with how recently the value which may be read by any low-level read was written into the mechanism and the order in which data values are read from the mechanism. A number of authors — [5, 8, 9, 17, 20, 23] — have considered the problem of checking these conditions for the 4-slot. In particular, the property of data coherence has been shown to be met under the assumption that the variables *latest*, *reading* and *slot* will never suffer from *metastability.* The phenomenon of metastability — see [5] for an explanation and a list of references — ensures that accesses to these variables are not atomic in the general case. However, Simpson ([22]) takes an engineering view and states that, in practice, it is possible to design and implement underlying hardware so that metastability is a negligible problem, from which the 4-slot can recover immediately in any case. As a result, he works from the assumption that accesses to these variables are atomic and we take advantage of this assumption in our modelling of the 4-slot in section 5.

9.2 Other approaches dealing with relaxation of atomicity

One of the best-known approaches from the literature which may be related conceptually to refinement-after-hiding is *action refinement*, where the focus is on

[9] These properties have been used in the verification of a number of different asynchronous communication mechanisms.

defining an operator to allow substitutition of precise, low-level behaviours for high-level specification actions (see [7] for a survey). However, action refinement suffers from the problem that, given a particular refinement mapping, only a single implementation process may be derived from a given specification; moreover, only a limited degree of relaxation of atomicity is possible. The notion of *vertical implementation*, described in [18], uses an implementation relation parameterized by a refinement mapping to address some of these problems of action refinement and is very close conceptually to refinement-after-hiding. Nonetheless, refinement-after-hiding is based on the use of an *abstraction* mapping, rather than on a refinement mapping which makes abstract behaviours more concrete. It seems that using an abstraction mapping allows for the possibility of a wider range of implementations for a given specification, although this is an issue which needs further exploration. Moreover, relating the behaviours of the register to those of the 4-slot using a refinement mapping seems to require at least one of two things: either that different occurrences of the same events on channels *read* and *write* in the register may be implemented in different ways depending on the way in which the implementations of earlier read and write events have been interleaved as the 4-slot executes; or that data refinement can be used in tandem with the application of the refinement mapping to transform the single variable of the register into the multiple data variables used in the 4-slot (with which data variables the respective refinements of the register read and write events would communicate). However, to the best of the author's knowledge, neither of these things is currently possible within the action refinement or vertical implementation approaches. We also comment briefly on the work described in [15] and which uses the *i/o automaton* model from [14] as a means of representing systems. This approach is concerned with the correctness of transaction-processing systems and relies on the use of a wrapper process to hide direct evidence of additional concurrent behaviour in any implementation process under consideration. This use of a wrapper process is necessary since the i/o automaton model cannot relate concurrent to sequential behaviours when those behaviours are to be regarded as visible in the processes under consideration: in essence, it suffers from the same problem as standard process algebraic refinement.

As stated in Section 1, the problem of the verification of the 4-slot is typical of those to which the correctness condition of *linearizability* ([10]) may be applied. Linearizability is a correctness condition for concurrent objects which allows us to relate the behaviour of any such concurrent object to that of a sequential specification object. If a concurrent object is correct with respect to the notion of linearizability then it presents the illusion that each procedure or method it provides appears to take effect instantaneously at some point between its call and its return. It is in this sense that linearizability is apt to be related to the verification presented in this paper: this notion of each procedure execution taking effect at a single instant is used in the definition of $extr_{FS}$, which is then used in the verification of the 4-slot. However, there are a number of practical points of difference between linearizability and refinement-after-hiding. Refinement-after-hiding can deal with other types of component process than concurrent objects. Linearizability is able to deal with systems composed of arbitrary concurrent objects, while the work described in Section 7 dealing with the correctness of environment processes with which the 4-slot might be composed currently only works in terms of the 4-slot itself. Refinement-after-hiding could, of course, be used to verify the relevant concurrent objects but further work is needed to allow the verification of the processes with which they might communicate. A network in the linearizability framework is given by a set of concurrent objects and a set of *sequential* processes, each of which processes may access those concurrent objects. In our framework, any individual process forming part of a larger component environment process which is to be composed with the 4-slot can itself exhibit concurrent behaviour; nonetheless, linearizability does allow individual processes to interact with multiple instances of the same object. Linearizability deals only with safety properties, while refinement-after-hiding is defined in all three CSP semantic models and the correctness of the 4-slot has been shown in all of them. Most significantly, the framework within which linearizability is presented lacks a fully general model of computation — for example, there is no general notion of process or of how processes may be composed — and this limits its wider usefulness. (Further comments regarding linearizability can be found in [3] in this volume; a fuller discussion of work related to refinement-after-hiding can be found in [2].)

10 Concluding remarks

In this paper, we have shown that Simpson's 4-slot mechanism is a valid implementation of a register and have also explored some of the wider issues which pertain to verifications of this sort. However, much further work is needed to develop a more general, easily usable and fully-formed notion of correctness to be used when an implementation has been derived from a specification through the use of relaxation of atomicity.

Acknowledgements
Thanks are due to Howard Bowman, Ian Clark, Neil Henderson, Cliff Jones, Maciej Koutny, Steve Paynter and Fei Xia for useful comments and discussions. The paper was also improved due to the comments of the referees. The author is grateful to (UK) EPSRC for the support provided by the "Splitting Atoms Safely (In Software Design)" project.

References

1. P. Bernstein, V. Hadzilacos and N. Goodman: *Concurrency Control and Recovery in Database Systems.* Addison-Wesley (1987).
2. J. Burton: *The Theory and Practice of Refinement-After-Hiding.* University of Newcastle upon Tyne (PhD Thesis) (2004). Available as University of Newcastle upon Tyne Technical Report CS-TR-904.
3. J. Burton and C. B. Jones: Investigating atomicity and observability. *Journal of Universal Computer Science* 11(5) (2005) 661–686.
4. J. Burton, M. Koutny and G. Pappalardo: Relating Communicating Processes with Different Interfaces. *Fundamenta Informaticae* 59(1) (2004) 1–37.
5. I. Clark: *A Unified Approach to the Study of Asynchronous Communication Mechanisms in Real Time Systems.* King's College, London University (PhD Thesis) (2000).
6. *FDR2 User Manual* : Available at: *http : //www.fsel.com/documentation/fdr2/*
7. R. Gorrieri and A. Rensink: Action Refinement. In: *Handbook of Process Algebra*, J. A. Bergstra, A. Ponse and S. A. Smolka (Eds.). Elsevier (2001) 1047–1147.
8. N. Henderson and S. Paynter: The Formal Classification and Verification of Simpson's 4-Slot Asynchronous Communication Mechanism. Proc. of *FME 2002*, L-H. Eriksson and P. Lindsay (Eds.). LNCS 2391 (2002) 350–369.
9. N. Henderson: Proving the Correctness of Simpson's 4-slot ACM Using an Assertional Rely-Guarantee Proof Method. Proc. of *FME 2003*, K. Araki, S. Gnesi and D. Mandrioli(Eds.). LNCS 2805 (2003) 244–263.
10. M. Herlihy and J. Wing: Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12(3) (1990) 463–492.
11. C. A. R. Hoare: *Communicating Sequential Processes.* Prentice Hall (1985).
12. M. Koutny, L. Mancini and G. Pappalardo: Two Implementation Relations and the Correctness of Communicating Replicated Processes. *Formal Aspects of Computing* 9 (1997) 119–148.
13. M. Koutny and G. Pappalardo: Behaviour Abstraction for Communicating Sequential Processes. *Fundamenta Informaticae* 48 (2001) 21–54.
14. N. A. Lynch and M. R. Tuttle: Hierarchical Correctness Proofs for Distributed Algorithms. Proc. of *6th ACM Symposium on Principles of Distributed Computing*, ACM (1987) 137–151.
15. N. Lynch, M. Merritt, W. Weihl and A. Fekete: *Atomic Transactions.* Morgan Kaufmann (1994).
16. R. Milner: *Communication and Concurrency.* Prentice Hall (1989).
17. S. E. Paynter, N. Henderson and J. M. Armstrong: Ramifications of Metastability in Bit Variables Explored Via Simpson's 4-Slot Mechanism. Technical Report 789, School of Computing Science, University of Newcastle upon Tyne (2003).
18. A. Rensink and R. Gorrieri: Vertical Implementation. *Information and Computation* 170 (2001) 95–133.
19. A. W. Roscoe: *The Theory and Practice of Concurrency.* Prentice-Hall (1998).
20. J. Rushby: Model Checking Simpson's Four-Slot Fully Asynchronous Communication Mechanism. Technical Report, Computer Science Laboratory, SRI International (2002).
21. D. Sangiorgi and D. Walker: *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press (2001).
22. H. R. Simpson: Four-slot Fully Asynchronous Communication Mechanism. *IEE Proceedings* 137, Pt E(1) (January 1990) 17–30.
23. H. R. Simpson: Correctness Analysis for Class of Asynchronous Communication Mechanisms. *IEE Proceedings* 139, Pt E(1) (January 1992) 35–49.
24. G. Weikum and G. Vossen: *Transactional Information Systems.* Morgan Kaufmann (2002).