Extending CSP with Tests for Availability

Gavin LOWE

Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK.

gavin.lowe@comlab.ox.ac.uk

Abstract. We consider the language of CSP extended with a construct that allows processes to test whether a particular event is available (without actually performing the event). We present an operational semantics for this language, together with two congruent denotational semantic models. We also show how this extended language can be simulated using standard CSP, so as to be able to analyse systems using the model checker FDR.

Keywords. CSP, tests for availability, semantic models

Introduction

Many languages for message-passing concurrency allow programs to test whether a channel is ready for communication, without actually performing that communication. For example, in JCSP [1,2], the input and output ends of channels have a method `pending()`, to test whether there is data ready to be read, or whether there is a reader ready to receive data, respectively. Java `InputStreams` have a method `available()` that returns the number of bytes that are available to be read. Andrews [3] gives a number of examples using such a construct.

In this paper, we study the effect of adding such tests to the process algebra CSP [4]. In particular, we add a single new construct to the language: the process

$$\text{if ready } a \text{ then } P \text{ else } Q \tag{1}$$

tests whether the event a is ready for communication, and then acts like either P or Q, appropriately. More precisely, the process tests whether all other processes that have a in their alphabet (and so who must synchronise on a) are ready to perform a.

We assume that within constructs of the form of (1), the test for the readiness of a is carried out only once: if the event becomes available or unavailable after the test is performed, that does not affect the branch that is selected. We allow processes to test for the readiness of events outside their own alphabet.

In this paper we investigate the effect of adding the construct (1) to semantic models for CSP. In the next section, we give a brief overview of the syntax and standard semantics of CSP. In Section 2 we give some examples using the new construct, both to illustrate its potential usefulness, and to highlight some implications for the semantic models. In Section 3 we give an operational semantics to the language; then in Section 4 we give congruent denotational models, analogous to the traces and stable failures models of CSP [4]. In Section 5 we show how this extended language can be simulated using standard CSP, so as to be able to analyse systems using a model checker such as FDR [5,6]. We sum up in Section 6.

1. A Brief Overview of CSP

In this section we give a brief overview of the syntax of CSP; for simplicity and brevity, we consider a fragment of the language in this paper. We also give a brief overview of the traces and stable failures models of CSP. For more details, see [7,4].

CSP is a process algebra for describing programs or *processes* that interact with their environment by communication. Processes communicate via atomic events, from some set Σ. Events are often passed on channels; for example, the event $c.3$ represents the value 3 being passed on channel c. The notation $\{\!|\, c \,|\!\}$ represents the set of events over channel c.

The simplest process is *STOP*, which represents a deadlocked process that cannot communicate with its environment. The process div represents a divergent process that can only perform internal events.

The process $a \rightarrow P$ offers its environment the event a; if the event is performed, it then acts like P. The process $c?x \rightarrow P$ is initially willing to input a value x on channel c, i.e. it is willing to perform any event of the form $c.x$; it then acts like P (which may use x).

A standard conditional is written as if b then P else Q, where b is a boolean condition on variables within the process (such as variables that hold values previously input)[1]. The process $b \,\&\, P$ is equivalent to if b then P else *STOP*: P is enabled only if the boolean guard b is true. For convenience, we extend this notation to readiness tests and define:

$$\begin{array}{rl} \mathsf{ready}\ a \,\&\, P & \text{as shorthand for} \quad \text{if ready } a \text{ then } P \text{ else } \mathit{STOP}, \\ \mathsf{notReady}\ a \,\&\, P & \text{as shorthand for} \quad \text{if ready } a \text{ then } \mathit{STOP} \text{ else } P. \end{array}$$

The tests now act as guards upon P, so that P can be performed only if a is available or not available, respectively.

The process $P \,\Box\, Q$ can act like either P or Q, the choice being made by the environment: the environment is offered the choice between the initial events of P and Q. By contrast, $P \sqcap Q$ may act like either P or Q, with the choice being made internally, and not under the control of the environment. The process $P \triangleright Q$ represents a sliding choice or timeout: the process initially acts like P, but if no event is performed then it can internally change state to act like Q.

The process $P \;{}_A\|_B\; Q$ runs P and Q in parallel; P is restricted to performing events from A; Q is restricted to performing events from B; the two processes synchronise on events from $A \cap B$. In this paper we will take the alphabets A and B to comprise just standard events, as opposed to actions corresponding to readiness tests. As noted above, we allow processes to test for the readiness of events outside their alphabets, e.g.

$$(\mathsf{ready}\ b \,\&\, a \rightarrow \mathit{STOP}) \;{}_{\{a\}}\|_{\{b\}}\; (b \rightarrow \mathit{STOP} \sqcap \mathit{STOP}).$$

In examples, we will tend to omit the alphabets when they are clear from the context.

The process $P \underset{A}{\|} Q$ runs P and Q in parallel, synchronising on events from A. The process $P \;|||\; Q$ interleaves P and Q, i.e. runs them in parallel with no synchronisation.

The process $P \setminus A$ acts like P, except the events from A are hidden, i.e. turned into internal, invisible events, denoted τ, which do not need to synchronise with the environment.

The process $P[\![R]\!]$ represents P where events are renamed according to the relation R, i.e., $P[\![R]\!]$ can perform an event b whenever P can perform an event a such that $a\,R\,b$. The relation R is often presented as a substitution; for example $P[\![b/a, c/a]\!]$ represents P, with the event a renamed to both b and c, and all other events unchanged.

[1] We use the same syntax for both standard conditionals and readiness tests, but they are semantically different constructs.

Recursive processes may be defined equationally, or using the notation $\mu X \bullet P$, which represents a process that acts like P, where each occurrence of X represents a recursive instantiation of $\mu X \bullet P$.

Prefixing ($\rightarrow$) and guarding ($\&$) bind tighter than each of the binary choice operators, which in turn bind tighter than the parallel operators.

CSP can be given both an operational and denotational semantics. The denotational semantics can either be extracted from the operational semantics, or defined directly over the syntax of the language; see [4]. It is more common to use the denotational semantics when specifying or describing the behaviours of processes, although most tools act on the operational semantics.

A *trace* of a process is a sequence of (visible) events that a process can perform. We write $traces(P)$ for the traces of P. If tr is a trace, then $tr \upharpoonright A$ represents the restriction of tr to the events in A, whereas $tr \setminus A$ represents tr with the events from A removed; concatenation is written "$^\frown$"; A^* represents the set of traces with events from A.

A *stable failure* of a process P is a pair (tr, X), which represents that P can perform the trace tr to reach a stable state (i.e. where no internal events are possible) where X can be refused, i.e., where none of the events of X is available. We write $failures(P)$ for the stable failures of P.

2. Examples

In this section we consider a few examples, firstly to illustrate the usefulness of the new construct, and then to highlight some aspects of the semantics.

Being able to detect readiness on channels can be useful in a number of circumstances. For example, the construct:

$$a \rightarrow P$$
$$\Box\ \mathsf{notReady}\, a \,\&\, b \rightarrow Q$$

gives priority to a over b: the event b can be performed only if the environment is not willing to perform a (at the point at which the test is made). Note, though, that if the environment withdraws its willingness to communicate a after the $\mathsf{notReady}\, a$ test is performed, then the above construct will be blocked, even if b is available: the construct makes the assumption about the environment that a is not withdrawn in this way.

As a slightly larger example, consider the classic readers and writers problem [8]. Here collections of readers and writers share a database. In order to maintain consistency, readers may not use the database at the same time as writers, and at most one writer may use the database at a time. The following guard process supports this: readers (resp. writers) gain entry to the database by performing the event *startRead* (resp. *startWrite*) and perform *endRead* (resp. *endWrite*) when they are finished. The parameters r and w record the number of readers and writers currently using the database, and satisfy the invariant $w \leq 1 \wedge (r > 0 \Rightarrow w = 0)$.

$$\begin{aligned} &Guard(r, w) = \\ &\quad w = 0 \,\&\, startRead \rightarrow Guard(r + 1, w) \\ &\quad \Box\ endRead \rightarrow Guard(r - 1, w) \\ &\quad \Box\ r = 0 \wedge w = 0 \,\&\, startWrite \rightarrow Guard(r, w + 1) \\ &\quad \Box\ endWrite \rightarrow Guard(r, w - 1). \end{aligned}$$

The problem with the above design is that writers may be permanently locked out of the database if there is always at least one reader using the database (even if no individual reader uses the database indefinitely). The following version gives priority to writers, by not allowing a new reader to start using the database if there is a writer waiting:

$$
\begin{aligned}
&\mathit{Guard}(r,w) = \\
&\quad w = 0 \;\&\; \mathsf{notReady}\; \mathit{startWrite} \;\&\; \mathit{startRead} \rightarrow \mathit{Guard}(r+1,w) \\
&\quad \Box\; \mathit{endRead} \rightarrow \mathit{Guard}(r-1,w) \\
&\quad \Box\; r = 0 \wedge w = 0 \;\&\; \mathit{startWrite} \rightarrow \mathit{Guard}(r,w+1) \\
&\quad \Box\; \mathit{endWrite} \rightarrow \mathit{Guard}(r,w-1).
\end{aligned}
$$

This idea can be extended further, to achieve fairness to both types of process; the parameter *priRead* records whether priority should be given to readers.[2]

$$
\begin{aligned}
&\mathit{Guard}(r,w,\mathit{priRead}) = \\
&\quad w = 0 \wedge \mathit{priRead} \;\&\; \mathit{startRead} \rightarrow \mathit{Guard}(r+1,w,\mathit{false}) \\
&\quad \Box\; w = 0 \;\&\; \mathsf{notReady}\; \mathit{startWrite} \;\&\; \mathit{startRead} \rightarrow \mathit{Guard}(r+1,w,\mathit{false}) \\
&\quad \Box\; \mathit{endRead} \rightarrow \mathit{Guard}(r-1,w,\mathit{false}) \\
&\quad \Box\; r = 0 \wedge w = 0 \wedge \neg \mathit{priRead} \;\&\; \mathit{startWrite} \rightarrow \mathit{Guard}(r,w+1,\mathit{true}) \\
&\quad \Box\; r = 0 \wedge w = 0 \;\&\; \mathsf{notReady}\; \mathit{startRead} \;\&\; \mathit{startWrite} \rightarrow \mathit{Guard}(r,w+1,\mathit{true}) \\
&\quad \Box\; \mathit{endWrite} \rightarrow \mathit{Guard}(r,w-1,\mathit{true}).
\end{aligned}
$$

We now consider a few examples in order to better understand aspects of the semantics of processes with readiness tests: it turns out that some standard algebraic laws no longer hold. Throughout these examples, we omit alphabets from the parallel composition operator where they are obvious from the context.

Example 1 Consider $P \parallel Q$ where $P = a \rightarrow \mathit{STOP}$ and $Q = \mathsf{if\ ready}\ a\ \mathsf{then}\ b \rightarrow \mathit{STOP}\ \mathsf{else}\ \mathit{error} \rightarrow \mathit{STOP}$. Clearly, it is possible for Q to detect that a is ready and so perform b. Could Q detect that a is not ready, and so perform *error*? If P makes a available immediately then clearly the answer is no. However, if it takes P some time to make a available, then Q could test for the availability of a before P has made it available.

We believe that any implementation of prefixing will take some time to make a available: for example, in a multi-threaded implementation, scheduling decisions will influence when the a becomes available; further, the code for making a available will itself take some time to run. This is the intuition we follow in the rest of the paper. This decision has a considerable impact on the semantics: it will mean that *all* processes will take some time to make events available (essentially since all the CSP operators maintain this property).

Returning to Example 1, in the combination $P \parallel Q$, Q can detect that a is not available initially and so perform *error*.

Example 2 $(\mathsf{if\ ready}\ a\ \mathsf{then}\ P\ \mathsf{else}\ Q) \setminus \{a\} = P \setminus \{a\}$: the hiding of a means that the $\mathsf{ready}\ a$ test succeeds, since there is nothing to prevent a from happening.

Example 3 External choice is not idempotent. Consider $P = a \rightarrow \mathit{STOP} \sqcap b \rightarrow \mathit{STOP}$ and $Q = \mathsf{ready}\ a \;\&\; \mathsf{ready}\ b \;\&\; \mathit{error} \rightarrow \mathit{STOP}$. Then $P \parallel Q$ cannot perform *error*, but $P \Box P \parallel Q$ can, if the two nondeterministic choices are resolved differently.

We do not allow external choices to be resolved by ready or notReady tests: we consider these tests to be analogous to evaluation of standard boolean conditions in if statements, or boolean guards, which are evaluated internally.

Example 4 The process $R = \mathsf{ready}\ a \;\&\; P \;\Box\; \mathsf{notReady}\ a \;\&\; Q$ is not the same as $\mathsf{if\ ready}\ a\ \mathsf{then}\ P\ \mathsf{else}\ Q$, essentially since the former checks for the readiness of a twice, but the latter checks

[2] In principle one could merge the first two branches, by using a guard $w = 0 \wedge (\mathit{priRead} \vee \mathsf{notReady}\ \mathit{startWrite})$; however, allowing complex guards that mix booleans with readiness testing would complicate the semantic definitions.

only once. When in the presence of the process $a \rightarrow STOP$, R can evolve to the state $P \,\Box\, Q$ (if the ready a test is made after the a becomes available, and the notReady a test is made before the a becomes available) or to $STOP \,\Box\, STOP = STOP$ (if the ready a test is made before the a becomes available, and the notReady a test is made after the a becomes available); neither of these is, in general, a state of if ready a then P else Q.

Example 5 ready a & ready b & P is not the same as ready b & ready a & P. Consider $P = error \rightarrow STOP$ and $Q = a \rightarrow STOP \triangleright b \rightarrow STOP$. Then ready a & ready b & $P \parallel Q$ can perform *error*, but ready b & ready a & $P \parallel Q$ cannot. Similar results hold for notReady, and for a mix of ready and notReady guards.

The above example shows why we do not allow more complex guards, such as ready $a \wedge$ ready b & P: any natural implementation of this process would have to test for the availability of a and b in some order, but the order in which those are tested can make a difference.

3. Operational Semantics

In this section we give operational semantics to the language of CSP extended with tests for the readiness or non-readiness of events. For simplicity, we omit interleaving, the $\underset{A}{\parallel}$ form of parallel composition, and renaming from the language we consider.

As normal, we write $P \xrightarrow{a} P'$, for $a \in \Sigma \cup \{\tau\}$ (where Σ is the set of visible events, and τ represents an internal event), to indicate that P performs the event a to become P'. In addition, we include transitions to indicate successful readiness or non-readiness tests:

- We write $P \xrightarrow{ready\, a} P'$ to indicate that P detects that the event a is ready, and evolves into P';
- We write $P \xrightarrow{notReady\, a} P'$ to indicate that P detects that the event a is not ready, and evolves into P'.

Note the different fonts between ready and notReady, which are part of the syntax, and *ready* and *notReady*, which are part of the semantics.

Define, for $A \subseteq \Sigma$:

$$ready\, A = \{ready\, a \mid a \in A\}, \qquad notReady\, A = \{notReady\, a \mid a \in A\},$$
$$A^\dagger = A \cup ready\, A \cup notReady\, A, \qquad A^{\dagger\tau} = A^\dagger \cup \{\tau\}.$$

Transitions, then, will be labelled by elements of $\Sigma^{\dagger\tau}$. We think of the $\xrightarrow{ready\, a}$ and $\xrightarrow{notReady\, a}$ transitions as being internal in the sense that they cannot be directly observed by any parallel peer. We refer to elements of $\Sigma^{\dagger\tau}$ as *actions*, and restrict the word *events* to elements of Σ^τ.

Below we use standard conventions, writing, e.g., $P \xrightarrow{a}$ for $\exists\, P' \bullet P \xrightarrow{a} P'$, and $P \not\xrightarrow{a}$ for $\neg(\exists\, P' \bullet P \xrightarrow{a} P')$.

Recall our intuition that a process such as $a \rightarrow P$ may not make the a available immediately. We model this by a τ transition to a state where the a is indeed available. It turns out that this latter state is not expressible within the syntax of the language (this follows from Lemma 11, below). Within the operational semantic definitions, we will write this state as $\breve{a} \rightarrow P$. We therefore define the semantics of prefixing by the following two rules.

$$a \rightarrow P \xrightarrow{\tau} \breve{a} \rightarrow P, \qquad \breve{a} \rightarrow P \xrightarrow{a} P.$$

We stress, though, that the $\breve{a} \rightarrow \ldots$ notation is only for the purpose of defining the operational semantics, and is not part of the language.

The following rules for normal events are completely standard. For brevity, we omit the symmetrically equivalent rules for external choice ($\Box$) and parallel composition ($_A\|_B$). The identifier a ranges over *visible* events.

$$\frac{P \xrightarrow{a} P'}{P \,\Box\, Q \xrightarrow{a} P'} \qquad \frac{P \xrightarrow{\tau} P'}{P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q}$$

$$\frac{P \xrightarrow{a} P'}{P \rhd Q \xrightarrow{a} P'} \qquad \frac{P \xrightarrow{\tau} P'}{P \rhd Q \xrightarrow{\tau} P' \rhd Q}$$

$$P \rhd Q \xrightarrow{\tau} Q \qquad P \sqcap Q \xrightarrow{\tau} P \qquad P \sqcap Q \xrightarrow{\tau} Q$$

$$\frac{P \xrightarrow{\alpha} P'}{P_A\|_B Q \xrightarrow{\alpha} P'_A\|_B Q}\ \alpha \in A - B \cup \{\tau\} \qquad \frac{\begin{array}{l}P \xrightarrow{a} P' \\ Q \xrightarrow{a} Q'\end{array}}{P_A\|_B Q \xrightarrow{a} P'_A\|_B Q'}\ a \in A \cap B$$

$$\frac{P \xrightarrow{\alpha} P'}{P \setminus A \xrightarrow{\alpha} P' \setminus A}\ \alpha \in \Sigma - A \cup \{\tau\} \qquad \frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A}\ a \in A$$

$$\mathsf{div} \xrightarrow{\tau} \mathsf{div} \qquad \mu X \bullet P \xrightarrow{\tau} P[\mu X \bullet P/X]$$

The following rules show how the tests for readiness operate.

$$\mathsf{if\ ready}\ a\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{ready\ a} P, \qquad \mathsf{if\ ready}\ a\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{notReady\ a} Q.$$

The remaining rules show how the readiness tests are promoted by various operators. We omit symmetrically equivalent rules for brevity. The rules for the choice operators are straightforward.

$$\frac{P \xrightarrow{ready\ a} P'}{P \,\Box\, Q \xrightarrow{ready\ a} P' \,\Box\, Q} \qquad \frac{P \xrightarrow{notReady\ a} P'}{P \,\Box\, Q \xrightarrow{notReady\ a} P' \,\Box\, Q}$$

$$\frac{P \xrightarrow{ready\ a} P'}{P \rhd Q \xrightarrow{ready\ a} P' \rhd Q} \qquad \frac{P \xrightarrow{notReady\ a} P'}{P \rhd Q \xrightarrow{notReady\ a} P' \rhd Q}$$

The rules for parallel composition are a little more involved. A *ready a* action can occur only if all processes with a in their alphabet are able to perform a.

$$\frac{\begin{array}{l}P \xrightarrow{ready\ b} P' \\ Q \xrightarrow{b}\end{array}}{P\ _A\|_B\ Q \xrightarrow{ready\ b} P'\ _A\|_B\ Q}\ b \in B \qquad \frac{P \xrightarrow{ready\ a} P'}{P\ _A\|_B\ Q \xrightarrow{ready\ a} P'\ _A\|_B\ Q}\ a \notin B$$

A *notReady a* action requires at least one parallel peer with a in its alphabet to be unable to perform a. In this case, the action is converted into a τ.

$$\frac{\begin{array}{l}P \xrightarrow{notReady\ b} P' \\ Q \not\xrightarrow{b}\end{array}}{P\ _A\|_B\ Q \xrightarrow{\tau} P'\ _A\|_B\ Q}\ b \in B \qquad \frac{\begin{array}{l}P \xrightarrow{notReady\ b} P' \\ Q \xrightarrow{b}\end{array}}{P\ _A\|_B\ Q \xrightarrow{notReady\ b} P'\ _A\|_B\ Q}\ b \in B$$

$$\frac{P \xrightarrow{notReady\ a} P'}{P \;{}_A\|_B\; Q \xrightarrow{notReady\ a} P' \;{}_A\|_B\; Q}\ a \notin B$$

Note that in the second rule, the *notReady b* may yet be blocked by some other parallel peer.

If a *ready a* action can be performed in a context where a is then hidden, then all relevant parallel peers are able to perform a; hence the transition can occur; the action is converted into a τ.

$$\frac{P \xrightarrow{ready\ a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A}\ a \in A$$

$$\frac{P \xrightarrow{\alpha} P'}{P \setminus A \xrightarrow{\alpha} P' \setminus A}\ \alpha \in ready(\Sigma - A) \cup notReady(\Sigma - A)$$

Note that there is no corresponding rule for *notReady a*: in the context $P \setminus A$, if P can perform *notReady a* (with $a \in A$) then all parallel peers with a in their alphabet are able to perform a, and so the a is available; hence the *notReady a* action is blocked for $P \setminus A$.

The following two lemmas can be proved using straightforward structural inductions. First, *ready a* and *notReady a* actions are available as alternatives to one another.

Lemma 6 For every process P:

$$(\exists\, Q \bullet P \xrightarrow{ready\ a} Q) \Leftrightarrow (\exists\, Q' \bullet P \xrightarrow{notReady\ a} Q').$$

Informally, the two transitions correspond to taking the two branches of a construct of the form if ready a then R else R'. The if construct may be only part of the process P above, and so R and R' may be only part of Q and Q' above.

Initially, each process can perform no standard events. This is a consequence of our assumption that a process of the form $a \rightarrow P$ cannot perform the a from its initial state.

Lemma 7 For every process P expressible using the syntax of the language (so excluding the $\breve{a} \rightarrow \ldots$ construct), and for every standard event $a \in \Sigma$, $P \not\xrightarrow{a}$.

Of course, P might have a τ transition to a state where visible events are available.

4. Denotational Semantics

We now consider how to build a compositional denotational semantic model for our language. We want the model to record at least the traces of visible events performed by processes: any coarser model is likely to be trivial.

In order to consider what other information is needed in the model, it is useful to consider (informally) a form of testing: we will say that test T distinguishes processes P and Q if $P \parallel T$ and $Q \parallel T$ have different traces of visible events. In this case, the denotational model should also distinguish P and Q.

We want to record within traces the *ready* and *notReady* actions that are performed. For example, the processes $b \rightarrow STOP$ and ready a & $b \rightarrow STOP$ are distinguished by the test *STOP* (with alphabet {a}); we will distinguish them denotationally by including the *ready a* action in the latter's trace.

Further, we want to record the events that were available as alternatives to those events that were actually performed. For example, the processes $a \rightarrow STOP \;\Box\; b \rightarrow STOP$ and

$a \rightarrow STOP \sqcap b \rightarrow STOP$ can be distinguished by the test $\mathsf{ready}\, a \;\&\; b \rightarrow STOP$; we will distinguish them denotationally by recording that the former offers a as an alternative to b.

We therefore add actions *offer* a and *notOffer* a to represent that a process is offering or not offering a, respectively. These actions will synchronise with *ready* a and *notReady* a actions. We write

$$\mathit{offer}\, A = \{\mathit{offer}\, a \mid a \in A\}, \qquad \mathit{notOffer}\, A = \{\mathit{notOffer}\, a \mid a \in A\},$$
$$A^{\ddagger} = A^{\dagger} \cup \mathit{offer}\, A \cup \mathit{notOffer}\, A, \qquad A^{\ddagger\tau} = A^{\ddagger} \cup \{\tau\}.$$

A trace of a process will, then, be a sequence of actions from $\Sigma^{\ddagger}$. We can calculate the traces of a process in two ways: by extracting then from the operational semantics, and by giving compositional rules. We begin with the former.

We augment the operational semantics with extra transitions as follows:

- We add *offer* a loops on every state P such that $P \xrightarrow{a}$;
- We add *notOffer* a loops on every state P such that $P \not\xrightarrow{a}$.

Formally, we define a new transition relation $\longrightarrow\!\!\triangleright$ by:

$$P \xrightarrow{\alpha}\!\!\triangleright\, Q \;\Leftrightarrow\; P \xrightarrow{\alpha} Q, \qquad \text{for } \alpha \in \Sigma^{\dagger\tau},$$
$$P \xrightarrow{\mathit{offer}\, a}\!\!\triangleright\, P \;\Leftrightarrow\; P \xrightarrow{a},$$
$$P \xrightarrow{\mathit{notOffer}\, a}\!\!\triangleright\, P \;\Leftrightarrow\; P \not\xrightarrow{a}.$$

Appendix A gives rules for the $\longrightarrow\!\!\triangleright$ that can be derived from the rules for the $\longrightarrow$ relation and the above definition.

We can then extract the traces (of $\Sigma^{\ddagger}$ actions) from the operational semantics (following [4, Chapter 7]):

Definition 8 We write $P \overset{tr}{\longmapsto} Q$, for $tr = \langle \alpha_1, \ldots, \alpha_n \rangle \in (\Sigma^{\ddagger\tau})^*$, if there exist $P_0 = P$, $P_1, \ldots, P_n = Q$ such that $P_i \xrightarrow{\alpha_{i+1}}\!\!\triangleright\, P_{i+1}$ for $i = 0, \ldots, n-1$. We write $P \overset{tr}{\Longrightarrow} Q$, for $tr \in (\Sigma^{\ddagger})^*$, if there is some tr' such that $P \overset{tr'}{\longmapsto} Q$ and $tr = tr' \setminus \tau$.

The traces of process P can then be defined to be the set of all tr such that $P \overset{tr}{\Longrightarrow}$. The following lemma states some healthiness conditions concerning this set.

Lemma 9 For all processes P expressible using the syntax of the language (so excluding the $\breve{a} \rightarrow \ldots$ construct), the set $T = \{tr \mid P \overset{tr}{\Longrightarrow}\}$ satisfies the following conditions:

1. T is non-empty and prefix-closed.
2. T includes $(\mathit{notOffer}\, \Sigma)^*$, i.e., the process starts in a state where no standard events are available.
3. *offer* and *notOffer* actions can always be remove from or duplicated within a trace:

   $$tr^\frown\langle\alpha\rangle^\frown tr' \in T \Rightarrow tr^\frown\langle\alpha,\alpha\rangle^\frown tr' \in T \wedge tr^\frown tr' \in T,$$

   for $\alpha \in \mathit{offer}\, \Sigma \cup \mathit{notOffer}\, \Sigma$.
4. *ready* a and *notReady* a actions are available as alternatives to one another:

   $$tr^\frown\langle \mathit{ready}\, a\rangle \in T \;\Leftrightarrow\; tr^\frown\langle \mathit{notReady}\, a\rangle \in T.$$

5. Either an *offer* a or *notOffer* a action is always available.

   $$tr^\frown tr' \in T \Rightarrow tr^\frown\langle \mathit{offer}\, a\rangle^\frown tr' \in T \vee tr^\frown\langle \mathit{notOffer}\, a\rangle^\frown tr' \in T.$$

Proof: (Sketch)

1. This follows directly from the definition of $\Longrightarrow$.
2. The follows from Lemma 7 and the definition of $\longrightarrow\!\triangleright$.
3. This follows directly from the definition of $\longrightarrow\!\triangleright$: *offer* and *notOffer* transitions always form self-loops.
4. This follows directly from Lemma 6.
5. This follows directly from the definition of $\longrightarrow\!\triangleright$: each state has either an *offer a* or a *notOffer a* loop.

□

4.1. Compositional Traces Semantics

We now give compositional rules for the traces of a process. The semantics for each process will be an element of the following model.

Definition 10 The *Readiness-Testing Traces Model* contains those sets $T \subseteq (\Sigma^{\ddagger})^*$ that satisfy conditions 2–5 of Lemma 9.

We write $traces_R[\![P]\!]$ for the traces of P[3]. Below we will show that these are congruent to the operational definition above.

STOP and div are equivalent in this model: they can perform no standard events; they can only signal that they are not offering events.

$$traces_R[\![STOP]\!] = traces_R[\![\mathsf{div}]\!] = (notOffer\ \Sigma)^*.$$

The process $a \rightarrow P$ can initially signal that it is not offering events; it can then signal that it is offering a but not offering other events; it can then perform a, and then continue like P.

$$\begin{array}{l} traces_R[\![a \rightarrow P]\!] = \\ \quad Init \cup \{tr^\frown\langle a\rangle^\frown tr' \mid tr \in Init \wedge tr' \in traces_R[\![P]\!]\} \\ \quad \text{where } Init = \{tr^\frown tr' \mid tr \in (notOffer\ \Sigma)^* \wedge tr' \in (\{offer\ a\} \cup notOffer(\Sigma - \{a\}))^*\}. \end{array}$$

The process if ready a then P else Q can initially signal that it is not offering events; it can then either detect that a is ready and continue as P, or detect that a is not ready and continue like Q.

$$\begin{array}{l} traces_R[\![\ \mathsf{if\ ready}\ a\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] = \\ \quad (notOffer\ \Sigma)^* \cup \\ \quad \{tr^\frown\langle ready\ a\rangle^\frown tr' \mid tr \in (notOffer\ \Sigma)^* \wedge tr' \in traces_R[\![P]\!]\} \cup \\ \quad \{tr^\frown\langle notReady\ a\rangle^\frown tr' \mid tr \in (notOffer\ \Sigma)^* \wedge tr' \in traces_R[\![Q]\!]\}. \end{array}$$

The process $P \triangleright Q$ can either perform a trace of P, or can perform a trace of P with no standard events, and then (after the timeout) perform a trace of Q. The process $P \sqcap Q$ can perform traces of either of its components.

$$\begin{array}{l} traces_R[\![P \triangleright Q]\!] = \\ \quad traces_R[\![P]\!] \cup \{tr_P^\frown tr_Q \mid tr_P \in traces_R[\![P]\!] \wedge tr_P \upharpoonright \Sigma = \langle\rangle \wedge tr_Q \in traces_R[\![Q]\!]\}, \end{array}$$

$$traces_R[\![P \sqcap Q]\!] = traces_R[\![P]\!] \cup traces_R[\![Q]\!].$$

Before the first visible event, the process $P \,\Box\, Q$ can perform an *offer a* action if either P or Q can do so; it can perform a *notOffer a* action if *both* P and Q can do so. Therefore, P

[3]We include the subscript "R" in $traces_R[\![P]\!]$ to distinguish this semantics from the standard traces semantics, $traces[\![P]\!]$.

and Q must synchronise on all *notOffer* actions before the first visible event. Let $tr \underset{notOffer\,\Sigma}{\|} tr'$ be the set of ways of interleaving tr and tr', synchronising on all *notOffer* actions (this operator is a specialisation of the $\underset{X}{\|}$ operator defined in [4, page 70]). The three sets in the definition below correspond to the cases where (a) neither process performs any visible events (so the two processes synchronise on *notOffer* actions throughout the execution), (b) P performs at least one visible event (after which, Q is turned off), and (c) the symmetric case where Q performs at least one visible event.

$$
\begin{aligned}
&traces_R[\![P \,\Box\, Q]\!] = \\
&\quad \{tr \mid \exists\, tr_P \in traces_R[\![P]\!], tr_Q \in traces_R[\![Q]\!] \bullet \\
&\qquad tr_P \upharpoonright \Sigma = tr_Q \upharpoonright \Sigma = \langle\rangle \wedge tr \in tr_P \underset{notOffer\,\Sigma}{\|} tr_Q\} \cup \\
&\quad \{tr^\frown\langle a\rangle^\frown tr'_P \mid \exists\, tr_P{}^\frown\langle a\rangle^\frown tr'_P \in traces_R[\![P]\!], tr_Q \in traces_R[\![Q]\!] \bullet \\
&\qquad tr_P \upharpoonright \Sigma = tr_Q \upharpoonright \Sigma = \langle\rangle \wedge a \in \Sigma \wedge tr \in tr_P \underset{notOffer\,\Sigma}{\|} tr_Q\} \cup \\
&\quad \{tr^\frown\langle a\rangle^\frown tr'_Q \mid \exists\, tr_P \in traces_R[\![P]\!], tr_Q{}^\frown\langle a\rangle^\frown tr'_Q \in traces_R[\![Q]\!] \bullet \\
&\qquad tr_P \upharpoonright \Sigma = tr_Q \upharpoonright \Sigma = \langle\rangle \wedge a \in \Sigma \wedge tr \in tr_P \underset{notOffer\,\Sigma}{\|} tr_Q\}.
\end{aligned}
$$

In order to give a semantic equation for parallel composition, we define a relation to capture how traces of parallel components are combined[4]. We write $(tr_P, tr_Q) \overset{A\|_B}{\longmapsto} tr$ if the traces tr_P of P and tr_Q of Q can lead to the trace tr of $P_A\|_B Q$. Let $private_A = (A - B) \cup offer(A - B) \cup notOffer\, A \cup ready\,(\Sigma - B) \cup notReady(\Sigma - B)$; these are the actions that the process with alphabet A can perform without any cooperation from the other process. Let $sync_{A,B} = (A \cap B) \cup offer(A \cap B)$; these are the actions that the two processes synchronise upon. The relation $\overset{A\|_B}{\longmapsto}$ is defined by:

$$(\langle\rangle, \langle\rangle) \overset{A\|_B}{\longmapsto} \langle\rangle,$$

if $(tr_P, tr_Q) \overset{A\|_B}{\longmapsto} tr$ and $b \in B$, then

$$
\begin{aligned}
&(\langle\alpha\rangle^\frown tr_P, tr_Q) \overset{A\|_B}{\longmapsto} \langle\alpha\rangle^\frown tr, \qquad \text{for } \alpha \in private_A, \\
&(\langle\alpha\rangle^\frown tr_P, \langle\alpha\rangle^\frown tr_Q) \overset{A\|_B}{\longmapsto} \langle\alpha\rangle^\frown tr, \qquad \text{for } \alpha \in sync_{A,B}, \\
&(\langle ready\, b\rangle^\frown tr_P, \langle offer\, b\rangle^\frown tr_Q) \overset{A\|_B}{\longmapsto} \langle ready\, b\rangle^\frown tr, \\
&(\langle notReady\, b\rangle^\frown tr_P, \langle notOffer\, b\rangle^\frown tr_Q) \overset{A\|_B}{\longmapsto} tr, \\
&(\langle notReady\, b\rangle^\frown tr_P, \langle offer\, b\rangle^\frown tr_Q) \overset{A\|_B}{\longmapsto} \langle notReady\, b\rangle^\frown tr, \\
&\langle \text{The symmetric equivalents of the above cases}\rangle.
\end{aligned}
$$

In the second clause: the first case corresponds to P performing a private action; the second case corresponds to P and Q synchronising on a shared action; the third case corresponds to a readiness test of P detecting that Q is offering b; the fourth case corresponds to a non-readiness test of P detecting that Q is not offering b; the fifth case corresponds to a non-readiness test of P detecting that Q is offering b. The reader might like to compare this definition with the corresponding operational semantics rules for parallel composition.

The semantics of parallel composition is then as follows; note that each component is restricted to its own alphabet, and that the composition can perform arbitrary *notOffer* $(\Sigma - A - B)$ actions:

[4]One normally defines a set-valued function to do this, but in our case it is more convenient to define a relation, since this leads to a much shorter definition.

$$
\begin{array}{l}
traces_R[\![P_A \|_B Q]\!] = \\
\quad \{tr \mid \exists\, tr_P \in traces_R[\![P]\!], tr_Q \in traces_R[\![Q]\!] \bullet \\
\qquad tr_P \upharpoonright (\Sigma - A) \cup offer(\Sigma - A) \cup notOffer(\Sigma - A) = \langle\rangle \wedge \\
\qquad tr_Q \upharpoonright (\Sigma - B) \cup offer(\Sigma - B) \cup notOffer(\Sigma - B) = \langle\rangle \wedge \\
\qquad (tr_P, tr_Q) \stackrel{A\|_B}{\longmapsto} tr \setminus notOffer(\Sigma - A - B)\}.
\end{array}
$$

The semantic equation for hiding of A captures that *notReady A* and *offer A* actions are blocked, A and *ready A* actions are internalised, and arbitrary *notOffer A* actions can occur.

$$
\begin{array}{l}
traces_R[\![P \setminus A]\!] = \\
\quad \{tr \mid \exists\, tr_P \in traces_R[\![P]\!] \bullet tr_P \upharpoonright (notReady\, A \cup offer\, A) = \langle\rangle \wedge \\
\qquad tr_P \setminus (A \cup ready\, A) = tr \setminus notOffer\, A\}.
\end{array}
$$

We now consider the semantics of recursion. Our approach follows the standard method using complete partial orders; see, for example, [4, Appendix A.1].

Lemma 11 The Readiness-Testing Traces Model forms a complete partial order under the subset ordering $\subseteq$, with $traces_R[\![\mathsf{div}]\!]$ as the bottom element.

Proof: That $traces_R[\![\mathsf{div}]\!]$ is the bottom element follows from item 2 of Lemma 9. It is straightforward to see that the model is closed under arbitrary unions, and hence is a complete partial order. □

The following lemma can be proved using precisely the same techniques as for the standard traces model; see [4, Section 8.2].

Lemma 12 Each of the operators is continuous with respect to the $\subseteq$ ordering.

Hence from Tarski's Theorem, each mapping F definable using the operators of the language has a least fixed point given by $\bigcup_{n\geq 0} F^n(\mathsf{div})$. This justifies the following definition.

$$
\begin{array}{l}
traces_R[\![\, \mu X \bullet F(X)]\!] = \\
\quad \text{the } \subseteq\text{-least fixed point of the semantic mapping corresponding to } F.
\end{array}
$$

The following theorem shows that the two ways of capturing the traces are congruent.

Theorem 13 For all traces $tr \in (\Sigma^{\ddagger})^*$:

$$tr \in traces_R[\![P]\!] \text{ iff } P \stackrel{tr}{\Longrightarrow} .$$

Proof: (Sketch) By structural induction over the syntax of the language. We give a couple of cases in Appendix B. □

Theorem 14 For all processes, $traces_R[\![P]\!]$ is a member of the Readiness-Testing Traces Model (i.e., it satisfies conditions 2–5 of Lemma 9).

Proof: This follows directly from Lemma 9 and Theorem 13. □

We can relate the semantics of a process in this model to the standard traces semantics. Let ϕ be the function that replaces readiness tests by nondeterministic choices, i.e.,

$$\phi(\mathsf{if\ ready}\ a\ \mathsf{then}\ P\ \mathsf{else}\ Q) = \phi(P) \sqcap \phi(Q)$$

and ϕ distributes over all other operators (e.g. $\phi(P_A\|_B Q) = \phi(P)_A\|_B\phi(Q)$). The standard traces of $\phi(P)$ are just the projection onto standard events of the readiness-testing traces of P.

Theorem 15 $traces[\![\phi(P)]\!] = \{tr \upharpoonright \Sigma \mid tr \in traces_R[\![P]\!]\}$.

4.2. Failures

We now consider how to refine the semantic model, to make it analogous to the stable failures model [4], i.e. to record information about which events can by stably refused.

The refusal of events seems, at first sight, to be very similar to those events not being offered, as recorded by *notOffer* actions. The difference is that refusals are recorded only in stable states, i.e. where no internal events are available: this means that if an event is stably refused, it will continue to be refused (until a visible event is performed); on the other hand, *notOffer* actions can occur in any states, and may subsequently become unavailable. So, for example:

- $a \rightarrow STOP \sqcap STOP$ is equivalent to $a \rightarrow STOP$ in the Readiness-Testing Traces model, since the traces of $STOP$ are included in the initial traces of $a \rightarrow STOP$; but $a \rightarrow STOP \sqcap STOP$ can stably refuse a initially, whereas $a \rightarrow STOP$ cannot.
- $a \rightarrow STOP \triangleright STOP \triangleright a \rightarrow STOP$ has the trace $\langle offer\, a, notOffer\, a, offer\, a\rangle$ (where the $notOffer\, a$ action is from the intermediate $STOP$ state) whereas $a \rightarrow STOP$ does not; but neither process can stably refuse a before a is performed.

Recall that in the standard model, stable failures are of the form (tr, X), where tr is a trace and X is a set of events that are stably refused. For the language in this paper, should refusal sets contain actions other than standard events?

Firstly, we should not consider states with *ready* or *notReady* transitions to be stable: recall that we consider these actions to be similar to τ events, in that they are not externally visible. We define:

$$stable\, P \Leftrightarrow \forall\, \alpha \in ready\, \Sigma \cup notReady\, \Sigma \cup \{\tau\} \bullet \neg P \xrightarrow{\alpha} .$$

Therefore such actions are necessarily unavailable in stable states, so there is no need to record them in refusal sets.

There is also no need to record the refusal of an $offer\, a$ action, since this will happen precisely when the event a is refused. It turns out that including *notOffer* actions within refusal sets can add to the discriminating power of the model. Consider

$$P = a \rightarrow STOP \triangleright STOP,$$

$$Q = (a \rightarrow STOP \triangleright STOP) \sqcap a \rightarrow STOP.$$

Then P and Q have the same traces, and have the same stable refusals of standard events. However, Q can, after the empty trace, stably refuse $\{b, notOffer\, a\}$ (i.e., stably offer a and stably refuse b), whereas P cannot.

We therefore have a choice as to whether or not we include *notOffer* actions within refusal sets. We choose not to, because the distinctions one can make by including them do not seem useful, and excluding them leads to a simpler model: in particular, the refusal of *notOffer* actions do not contribute to the performance or refusal of any standard events. I suspect that including *notOffer* actions within refusal sets would lead to a model similar in style to the stable ready sets model [9,10].

Hence, we define, for $X \subseteq \Sigma$:

$$P\ ref\ X \Leftrightarrow stable\, P \wedge \forall\, x \in X \bullet \neg P \xrightarrow{x} .$$

We then define the stable failures of a process in the normal way:

$$(tr, X) \in failures_R[\![P]\!] \Leftrightarrow \exists\, Q \bullet P \stackrel{tr}{\Longrightarrow} Q \wedge Q\ ref\ X. \tag{2}$$

Definition 16 The Readiness-Testing Stable Failures Model contains those pairs (T, F) where $T \subseteq (\Sigma^{\ddagger})^*$, $F \subseteq (\Sigma^{\ddagger})^* \times \mathbf{P}\,\Sigma$, T satisfies conditions 2–5 of the Readiness-Testing Traces Model, and also

6. If $(tr, X) \in F$ then $tr \in T$.

Below, we give compositional rules for the stable failures of a process. Since the notion of refusal is identical to as in the standard stable failures model, the refusal components are calculated precisely as in that model, and so the equations are straight-forward adaptations of the rules for traces. The only point worth noting is that in the construct $\mathsf{if\ ready}\ a\ \mathsf{then}\ P\ \mathsf{else}\ Q$, no failures are recorded before the if is resolved.

$$failures_R[\![\mathsf{div}]\!] = \{\},$$

$$failures_R[\![STOP]\!] = \{(tr, X) \mid tr \in (notOffer\,\Sigma)^* \wedge X \subseteq \Sigma\},$$

$$\begin{aligned}
&failures_R[\![a \rightarrow P]\!] = \\
&\quad \{(tr, X) \mid tr \in Init \wedge a \notin X\} \cup \\
&\quad \{(tr^\frown\langle a\rangle^\frown tr', X) \mid tr \in Init \wedge (tr', X) \in failures_R[\![P]\!]\} \\
&\quad \text{where } Init = \{tr^\frown tr' \mid tr \in (notOffer\,\Sigma)^* \wedge tr' \in (\{offer\,a\} \cup notOffer(\Sigma - \{a\}))^*\},
\end{aligned}$$

$$\begin{aligned}
&failures_R[\![\,\mathsf{if\ ready}\ a\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] = \\
&\quad \{(tr^\frown\langle ready\,a\rangle^\frown tr', X) \mid tr \in (notOffer\,\Sigma)^* \wedge (tr', X) \in failures_R[\![P]\!]\} \cup \\
&\quad \{(tr^\frown\langle notReady\,a\rangle^\frown tr', X) \mid tr \in (notOffer\,\Sigma)^* \wedge (tr', X) \in failures_R[\![Q]\!]\},
\end{aligned}$$

$$\begin{aligned}
&failures_R[\![P \triangleright Q]\!] = \\
&\quad \{(tr, X) \mid (tr, X) \in failures_R[\![P]\!] \wedge tr \restriction \Sigma \neq \langle\rangle\} \cup \\
&\quad \{(tr_P^\frown tr_Q, X) \mid tr_P \in traces[\![P]\!] \wedge tr_P \restriction \Sigma = \langle\rangle \wedge (tr_Q, X) \in failures_R[\![Q]\!]\},
\end{aligned}$$

$$failures_R[\![P \sqcap Q]\!] = failures_R[\![P]\!] \cup failures_R[\![Q]\!],$$

$$\begin{aligned}
&failures_R[\![P \,\Box\, Q]\!] = \\
&\quad \{(tr, X) \mid \exists (tr_P, X) \in failures_R[\![P]\!], (tr_Q, X) \in failures_R[\![Q]\!] \bullet \\
&\qquad tr_P \restriction \Sigma = tr_Q \restriction \Sigma = \langle\rangle \wedge tr \in tr_P \underset{notOffer\,\Sigma}{\|} tr_Q\} \cup \\
&\quad \{(tr^\frown\langle a\rangle^\frown tr'_P, X) \mid \\
&\qquad \exists (tr_P^\frown\langle a\rangle^\frown tr'_P, X) \in failures_R[\![P]\!], tr_Q \in traces[\![Q]\!] \bullet \\
&\qquad\quad tr_P \restriction \Sigma = tr_Q \restriction \Sigma = \langle\rangle \wedge a \in \Sigma \wedge tr \in tr_P \underset{notOffer\,\Sigma}{\|} tr_Q\} \cup \\
&\quad \{(tr^\frown\langle a\rangle^\frown tr'_Q, X) \mid \\
&\qquad \exists\, tr_P \in traces[\![P]\!], (tr_Q^\frown\langle a\rangle^\frown tr'_Q, X) \in failures_R[\![Q]\!] \bullet \\
&\qquad\quad tr_P \restriction \Sigma = tr_Q \restriction \Sigma = \langle\rangle \wedge a \in \Sigma \wedge tr \in tr_P \underset{notOffer\,\Sigma}{\|} tr_Q\},
\end{aligned}$$

$$\begin{aligned}
&failures_R[\![P_A \|_B Q]\!] = \\
&\quad \{(tr, Z) \mid \exists (tr_P, X) \in failures_R[\![P]\!], (tr_Q, Y) \in failures_R[\![Q]\!] \bullet \\
&\qquad tr_P \restriction (\Sigma - A) \cup offer(\Sigma - A) \cup notOffer(\Sigma - A) = \langle\rangle \wedge \\
&\qquad tr_Q \restriction (\Sigma - B) \cup offer(\Sigma - B) \cup notOffer(\Sigma - B) = \langle\rangle \wedge \\
&\qquad (tr_P, tr_Q) \overset{A\|_B}{\longmapsto} tr \setminus notOffer(\Sigma - A - B) \wedge Z - A - B = X \cap A \cup Y \cap B\},
\end{aligned}$$

$$\begin{aligned}
&failures_R[\![P \setminus A]\!] = \\
&\quad \{(tr, X) \mid \exists (tr_P, X \cup A) \in failures_R[\![P]\!] \bullet \\
&\qquad tr_P \restriction (notReady\,A \cup offer\,A) = \langle\rangle \wedge tr_P \setminus (A \cup ready\,A) = tr \setminus notOffer\,A\},
\end{aligned}$$

$$\begin{aligned}
&failures_R[\![\,\mu X \bullet F(X)]\!] = \\
&\quad \text{the } \subseteq\text{-least fixed point of the semantic mapping corresponding to } F.
\end{aligned}$$

The fixed-point definition for recursion can be justified in a similar way to as for traces. The congruence of the above rules to the operational definition of stable failures —i.e., equation (2)— can be proved in a similar way to Theorem 13. Conditions 2–5 of the Readiness-Testing Stable Failures Model are satisfied, because of the corresponding result for traces

(Theorem 14). Condition 6 follows directly from the definition of a stable failure, and the congruence of the operational and denotational semantics.

The following theorem relates the semantics of a process in this model to the standard stable failures semantics.

Theorem 17 $\mathit{failures}[\![\phi(P)]\!] = \{(tr \upharpoonright \Sigma, X) \mid (tr, X) \in \mathit{failures}_R[\![P]\!]\}$.

5. Model Checking

In this section we illustrate how one can use a standard CSP model checker, such as FDR [5, 6], to analyse processes in the extended language of this paper. We just give an example here, in order to give the flavour of the translation; we discuss prospects for generalising the approach in the concluding section of the paper.

We consider the following solution to the readers and writers problem.

$$
\begin{array}{l}
\mathit{Guard}(r, w) = \\
\quad w = 0 \wedge r < N \;\&\; \mathsf{notReady}\, \mathit{startWrite} \;\&\; \mathit{startRead} \rightarrow \mathit{Guard}(r+1, w) \\
\quad \Box\ r > 0 \;\&\; \mathit{endRead} \rightarrow \mathit{Guard}(r-1, w) \\
\quad \Box\ r = 0 \wedge w = 0 \;\&\; \mathit{startWrite} \rightarrow \mathit{Guard}(r, w+1) \\
\quad \Box\ w > 0 \;\&\; \mathit{endWrite} \rightarrow \mathit{Guard}(r, w-1).
\end{array}
$$

This is the solution from Section 2 that gives priority to writers, except we impose a bound of N upon the number of readers, and add guards to the second and fourth branches, in order to keep the state space finite. We will show that this solution is starvation free as far as the writers is concerned: i.e. if a writer is trying to gain access then one such writer eventually succeeds.

We will simulate the above guard process using standard CSP, in particular simulating the *ready*, *notReady*, *offer* and *notOffer* actions by fresh CSP events on channels `ready`, `notReady`, `offer` and `notOffer`. Each process is translated into a form that uses these channels, following the semantics presented earlier; the simulation will have transitions that correspond to the $\longrightarrow\!\!\triangleright$ transitions of the original, except it will have a few additional τ transitions that do not affect the failures-divergences semantics. More precisely, let $\hat{\alpha}$ be the event used to simulate the action α; for example, if $\alpha = \mathit{ready}\, e$ then $\hat{\alpha} = \texttt{ready}.e$. Then each process P is simulated by a translation $\mathit{trans}(P)$, where if $P \xrightarrow{\alpha}\!\!\triangleright\ Q$ then $\mathit{trans}(P) \xrightarrow{\hat{\alpha}} (\xrightarrow{\tau})^* \mathit{trans}(Q)$, and vice versa. In particular, for each standard event e, we must add an `offer`.e or `notOffer`.e loop to each state.

For convenience, and to distinguish between the source and target languages, we present the simulation using prettified machine-readable CSP.[5]

The standard events and the channels to simulate the non-standard actions are declared as follows:

```
channel startWrite, endWrite, startRead, endRead
E = {startWrite, endWrite, startRead, endRead}
channel ready, notReady, offer, notOffer : E
```

We start by defining some helper processes. The following process is the translation of *STOP*: it can only signal that it is not offering standard events.

```
STOPT = notOffer?e → STOPT
```

[5]The CSP text below is produced (almost) directly from the machine-readable CSP using LaTeX macros.

The following process is the translation of $e \rightarrow P$: initially it can signal that it is not offering standard events; it can then timeout into a state where `e` is available, after which it acts like `P`; in this latter state it can also signal that it is offering `e` but no other standard events.[6]

```
Prefix(e,P) = notOffer?e1 → Prefix(e,P) ▷ Prefix1(e,P)

Prefix1(e,P) =
  e → P
  □ offer.e → Prefix1(e,P)
  □ notOffer?e1:diff(E,{e}) → Prefix1(e,P)
```

The reader might like to compare these with the $\longrightarrow\!\!\triangleright$ semantics in Appendix A.

In order to simulate the *Guard* process, we simulate each branch as a separate parallel process: the branches of *Guard* synchronise on *notOffer* actions before the choice is resolved, so the processes simulating these branches will synchronise on appropriate `notOffer` events. The first branch is simulated as below:

```
Branch(1,r,w) =
  if w==0 and r<N then
    notReady.startWrite → Prefix(startRead, Restart(1,r+1,w))
    □ ready.startWrite → STOPT
    □ notOffer?e → Branch(1,r,w)
  else STOPT
```

We explain the `Restart` process below. Note how the notReady *startWrite* test is simulated by the `notReady.startWrite` and `ready.startWrite` events. Note also how the process signals which standard events are and are not available in the different states. The other branches are slightly simpler, as they do not include readiness tests.

```
Branch(2,r,w) =
  if r>0 then Prefix(endRead, Restart(2,r−1,w)) else STOPT
Branch(3,r,w) =
  if r==0 and w==0 then Prefix(startWrite, Restart(3,r,w+1)) else STOPT
Branch(4,r,w) =
  if w>0 then Prefix(endWrite, Restart(4,r,w−1)) else STOPT
```

When one branch executes and reaches a point corresponding to a recursion within the *Guard* process, all the other branch processes need to be restarted, with new values for `r` or `w`. We implement this by the executing branch signalling on the channel `restart`.

```
R = {0..N} -- possible values of r
W = {0..1} -- possible values of w
BRANCH = {1..4} -- branch identifiers
channel restart : BRANCH.R.W

Restart(i,r,w) = restart!i.r.w → Branch(i,r,w)
```

Each branch can receive such a signal from another branch, as an interrupt, and restart with the new values for `r` and `w`.[7]

```
Branch'(i,r,w) =
  Branch(i,r,w) △ restart?j:diff(BRANCH,{i})?r'.w' → Branch'(i,r',w')
```

[6] The operator `diff` represents set difference.

[7] The $\triangle$ is an interrupt operator; the left hand side is interrupted when the right hand side performs an event.

Below we will combine these `Branch'` processes in parallel so as to simulate *Guard*. We will need to be able to identify which branch performs certain events. For events `e` other than `notOffer` events, we rename `e` performed by branch `i` to `c.i.e`. We rename each `notOffer.e` event performed by branch `i` to both itself and `notOffer1.i.e`: the former will be used before the choice is resolved (synchronised between all branch processes), and the latter will be used after the choice is resolved (privately to branch `i`):[8]

```
EE = union(E, {|ready, notReady, offer|}) -- events other than notOffer
channel c : BRANCH.EE -- c.i.e represents event e done by Branch(i,_,_)
channel notOffer1 : BRANCH.E

Branch''(i,r,w) =
  Branch'(i,r,w)
    ⟦e \ c.i.e | e ← EE⟧
    ⟦notOffer.e \ notOffer.e, notOffer.e \ notOffer1.i.e | e ← E⟧

alpha(i) = {|c.i,restart,notOffer,notOffer1.i|} -- alphabet of branch i
```

Below we will combine the branch processes in parallel, together with a regulator process `Reg` that, once a branch has done a standard event to resolve the choice, blocks all events of the other branches until a restart occurs; further, it forces processes to synchronise on `notOffer` events before the choice is resolved, and subsequently allows the unsynchronised `notOffer1` events.[9]

```
Reg = c?i?e → (if member(e,E) then Reg'(i) else Reg)
      □ notOffer?_ → Reg
Reg'(i) = c.i?_ → Reg'(i) □ restart.i?_?_ → Reg □ notOffer1.i?_ → Reg'(i)
```

We build the guard process by combining the branches and regulator in parallel, hiding the `restart` events, and reversing the above renaming.[10]

```
Guard0(r,w) =
  (||| i : BRANCH • [alpha(i)] Branch''(i,r,w))
    [| {|c,restart,notOffer,notOffer1|} |] Reg
Guard(r,w) =
  (Guard0(r,w) \ {| restart |})
    ⟦c.i.e \ e | e ← EE, i ← BRANCH⟧
    ⟦notOffer1.i.e \ notOffer.e | e ← E, i ← BRANCH⟧
```

We can check the simple safety property that the guard allows at most one active writer at a time, and never allows both readers and writers to be active.

```
Spec(r,w) =
  w==0 and r<N & startRead → Spec(r+1,w)
  □ r>0 & endRead → Spec(r−1,w)
  □ r==0 and w==0 & startWrite → Spec(r,w+1)
  □ w>0 & endWrite → Spec(r,w−1)

internals = {|ready, notReady, offer, notOffer, writerTrying|}

assert Spec(0,0) ⊑T Guard(0,0) \ internals
```

[8] `union` represents the union operation.

[9] `member` tests for membership of a set.

[10] The ||| is an indexed parallel composition, indexed by `i`; here the `i`th component has alphabet `alpha(i)`. The notation `[| A |]` is the machine-readable CSP version of $\|_A$.

This test succeeds, at least for small values of N.

In order to verify a liveness property, we need to model the readers and writers themselves. Each reader alternates between performing startRead and endRead, or may decide to stop (when not reading). Each writer is similar, but, for later convenience, we add an event writerTrying to indicate that it is trying to perform a write. It is important that the startWrite event becomes available *immediately* after the writerTrying event, in order for the liveness property below to be satisfied, hence we have the following form.

```
channel writerTrying

Reader = Prefix(startRead, Prefix(endRead, Reader)) ⊓ STOPT
Writer =
  (writerTrying → Prefix1(startWrite , Prefix(endWrite, Writer)) ⊓ STOPT)
  □ notOffer.startWrite → Writer
```

Following the semantic definitions, we need to synchronise the notOffer.startWrite events of the individual writers, and we need to synchronise the offer.startWrite and notOffer.startWrite events of the writers with the ready.startWrite and notReady.startWrite events of the guard, respectively. For convenience, we block the remaining offer and notOffer events, since we make no use of them, and processes do not change state when they perform such an event.

```
Readers = (||| r:{1..N} • Reader)

Writers = ([|{notOffer.startWrite}|] w:{1..N} • Writer)
              ⟦offer.startWrite \ ready.startWrite,
               notOffer.startWrite \ notReady.startWrite⟧

ReadersWriters = Readers ||| Writers

System =
  let SyncSet = union(E,{ready.startWrite,notReady.startWrite}) within
  (Guard(0,0) [| SyncSet |] ReadersWriters) [| {|offer,notOffer|} |] STOP
```

We now consider the liveness property that the guard is fair to the writers, in the sense that if at least one writer is trying to gain access, then one of them eventually succeeds. Testing for this property is not easy: the only way to test that a startWrite event *eventually* becomes available is to hide the readers' events, and to check that startWrite becomes available without a divergence (so after only finitely many readers' events); however, hiding all the readers' events will lead to a divergence when no writer is trying to gain access (at which point refinement tests do not act in the way we would like). What we therefore do is use a construction that has the effect of hiding the readers' events when a writer is trying to gain access, but leaving the startRead events visible when no writer is trying to gain access. We then test against the following specification (where all other irrelevant events are hidden).

```
WLSpec(n) = n<N & (writerTrying → WLSpec(n+1) ⊓ STOP)
            □ n>0 & startWrite → WLSpec(n−1)
            □ n==0 & (startRead → WLSpec(n) ⊓ STOP)
```

The parameter n records how many writers are currently trying; when n>0 (i.e. at least one writer is trying), this process insists that a writer can start after a finite amount of (hidden) activity by the readers; when n==0 (i.e. no writer is trying), the process allows arbitrary startRead events.

The way to implement the state-dependent hiding described above is to rename `startRead` events both to themselves and a new event `startRead'`, put this in parallel with a regulator that allows `startRead` events when no writer is trying and `startRead'` events when at least one writer is trying, and then hide `startRead'` and other irrelevant events.

```
channel startRead'

System' =
  (System ⟦startRead \ startRead, startRead \ startRead'⟧
   [| {writerTrying,startWrite,startRead,startRead'} |] Reg1(0))
      \ {|endRead,endWrite,ready,notReady,startRead'|}

Reg1(n) = n<N & writerTrying → Reg1(n+1)
          □ n>0 & startWrite → Reg1(n−1)
          □ n==0 & startRead → Reg1(n)
          □ n>0 & startRead' → Reg1(n)
```

We can then use FDR to verify

```
assert WLSpec(0) ⊑FD System'
```

In particular, this means that the right hand side is divergence-free, so when `n>0` the `startWrite` events will become available after a finite number of the hidden events.

6. Discussion

In this paper we have considered an extension of CSP that allows processes to test whether an event is available. We have formalised this construct by giving an operational semantics and congruent denotational semantic models. We have illustrated how we can use a standard model checker to analyse systems in this extended language.

In this final section we discuss some related work and possible extensions to this work.

6.1. Comparison with Standard Models

There have been several different denotational semantic models for CSP. Most of these are based on the standard syntax of CSP, with the standard operational semantics. It is not possible to compare these models directly with the models in this paper, since we have used a more expressive language, with the addition of readiness tests; however, we can compare them with the sub-language excluding this construct.

For processes that do not use readiness tests, the two models of this paper are more distinguishing than the standard traces and stable failures models, respectively. Theorems 15 and 17 show that our models make at least as many distinctions as the standard models. They distinguish processes that the standard models identify, such as

$$a \rightarrow STOP \sqcap (a \rightarrow STOP \triangleright b \rightarrow STOP) \qquad \text{and} \qquad a \rightarrow STOP \sqcap b \rightarrow STOP :$$

the former, but not the latter, has the trace $\langle \mathit{offer}\, a, b \rangle$.

In [10], Roscoe gives a survey of the denotational models based on the standard syntax. The most distinguishing of those models based on finite observations (and so not modelling divergences) is the *finite linear model*, $\mathcal{FL}$. This model uses observations of the form $\langle A_0, a_0, A_1, a_1, \ldots, a_{n-1}, A_n \rangle$, where each a_i is an event that is performed, and each A_i is either (a) a set of events, representing that those events are offered in a stable state (and so $a_i \in A_i$), or (b) the special value $\bullet$ representing no information about what events are stably offered (perhaps because the process did not stabilise).

For processes with no readiness tests, $\mathcal{FL}$ is incomparable with the models in this paper. Our models distinguish processes that $\mathcal{FL}$ identifies, essentially because the latter records the availability of events only in stable states, whereas our models record this information also in unstable states. For example, the processes

$$(b \rightarrow STOP \triangleright a \rightarrow STOP) \sqcap b \rightarrow STOP \qquad \text{and} \qquad a \rightarrow STOP \sqcap b \rightarrow STOP$$

are distinguished in our models since just the former has the trace $\langle \mathit{offer}\, b, a \rangle$; however they are identified in $\mathcal{FL}$ (and hence all the other finite observation models from [10]) because this b is not *stably* available.

Conversely, $\mathcal{FL}$ distinguishes processes that our models identify, such as

$$a \rightarrow b \rightarrow STOP \sqcap (a \rightarrow STOP \triangleright STOP) \qquad \text{and} \qquad a \rightarrow (b \rightarrow STOP \sqcap STOP) \triangleright STOP,$$

since the former has the observation $\langle \{a\}, a, \bullet, b, \bullet \rangle$, but the latter does not since its a is performed from an unstable state; however, they are identified by our failures model (and hence our traces model) since this records stability information only at the end of a trace. I believe it would be straightforward to extend our models to record stability information *throughout* the trace in the way $\mathcal{FL}$ does.

Roscoe also shows that each of the standard finite observation models can be extended to model divergences in three essentially different ways. I believe it would be straightforward to extend the model in this paper to include divergences following any of these techniques.

6.2. Comparison with Other Prioritised Models

As we described in the introduction, the readiness tests can be used to implement a form of priority. There have been a number of previous attempts to add priority to CSP.

Lawrence [11] models priorities by representing a process as a set of triples of the form (tr, X, Y), meaning that after performing trace tr, if a process is offered the set of events X, then it will be willing to perform any of the events from Y. For example, a process that initially gives priority to a over b would include the triple $\langle\rangle, \{a, b\}, \{a\})$.

Fidge [12] models priorities using a set of "preferences" relations over events. For example, a process that gives priority to a over b would have the preferences relation $\{a \mapsto a, b \mapsto b, a \mapsto b\}$.

In [13,14], I modelled priorities within timed CSP using an order over the sets (actually, multi-sets) of events that the process could do at the same time. For example, a process that gives priority to a over b (and would rather do either than nothing) at time 0 would have the ordering $(0, \{a\}) \sqsupset (0, \{b\}) \sqsupset (0, \{\})$.

All the above models are rather complex: I would claim that the model in this paper is somewhat simpler, and has the advantage of allowing a translation into standard CSP, in order to use the FDR model checker.

One issue that sometimes arises when considering priority is what happens when two processes with opposing priorities are composed in parallel. For example, consider $P \parallel Q$ where P and Q give priority to a and b respectively:

$$P = a \rightarrow P_1 \;\Box\; \mathsf{notReady}\, a \;\&\; b \rightarrow P_2,$$

$$Q = b \rightarrow Q_1 \;\Box\; \mathsf{notReady}\, b \;\&\; a \rightarrow Q_2.$$

There are essentially three ways in which this parallel composition can behave (in a context that doesn't block a or b):

- If P performs its notReady test before Q, the test succeeds; if, further, P makes the b available before Q performs its notReady test, then Q's notReady test will fail, and so the parallel composition will perform b;

- The symmetric opposite of the previous case, where Q performs its notReady test and makes a available before P performs its notReady test, and so the parallel composition performs a;
- If both processes perform their notReady tests before the other process makes the corresponding event available, then both tests will succeed, and so both events will be possible.

I consider this to be an appropriate solution. By contrast, the model in [11] leads to this system deadlocking; in [13,14] I used a prioritised parallel composition operator to give priority to the preferences of one components, thereby avoiding this problem, but at the cost of considerable complexity.

6.3. Readiness Testing for Channels

Most CSP-like programming languages make use of channels that pass data. In such languages, one can often test for the availability of *channels*, rather than of individual events. Note, though, that there is an asymmetry between the reader and writer of the channel: the reader will normally want to test whether there is *any* event on the channel that is ready for communication (i.e. the writer is ready to communicate), whereas the writer will normally want to test if *all* events on the channel are ready for communication (i.e. the reader is ready to communicate). It would be interesting to extend the language of this paper with constructs

$$\mathsf{if\ ready\ all}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q \qquad \text{and} \qquad \mathsf{if\ ready\ any}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q,$$

where A is a set of events (e.g. all events on a channel), to capture this idea.

6.4. Model Checking

In Section 5, we gave an indication as to how to simulate the language of this paper using standard CSP, so as to use a model checker such as FDR.

This technique works in general. This can be shown directly, by exhibiting the translation. Alternatively, we can make use of a general result from [15], where Roscoe shows that any operator with an operational semantics that is "CSP-like" —essentially that the operator can turn arguments on, interact with arguments via visible events, promote τ events of arguments, and maybe turn arguments off— can be simulated using standard CSP operators. The $\longrightarrow\!\triangleright$ semantics of this paper is "CSP-like" in this sense, so we can use those techniques to simulate this semantics. We intend to automate the translation.

One difficulty is that the translation from [15] can produce processes that are infinite state because they branch off extra parallel processes at each recursion of the simulated process. In Section 5 we avoided this problem by *restarting* the `Branch` processes that constituted the guard at each recursion (this uses an idea also due to Roscoe), whereas Roscoe's approach would branch off new processes at each recursion. I believe the technique in Section 5 can be generalised and probably automated.

6.5. Full Abstraction

The denotational semantic models we have presented turn out not to be fully abstract with respect to may-testing [16]. Consider the process $\mathsf{if\ ready}\ a\ \mathsf{then}\ P\ \mathsf{else}\ P$. This is denotationally distinct from P, since its traces have *ready* a or *notReady* a events added to the traces of P. Yet there seems no way to distinguish the two processes by testing: i.e., there is no good reason to consider those processes as distinct.

We believe that one could form a fully abstract semantics as follows. Consider the relation $\sim$ over sets of traces defined by

$$(S \cup \{tr^\frown tr'\}) \sim (S \cup \{tr^\frown\langle ready\, a\rangle^\frown tr', tr^\frown\langle notReady\, a\rangle^\frown tr'\})$$
$$\text{for all } S \in \mathbf{P}(\Sigma^*),\ tr, tr' \in \Sigma^*,\ a \in \Sigma.$$

In other words, two sets are related if one is formed from the other by adding *ready a* and *notReady a* actions in the same place. Let $\approx$ be the transitive reflexive closure of $\sim$. This relation essentially abstracts away irrelevant readiness tests. We conjecture that P and Q are testing equivalent iff $traces[\![P]\!] \approx traces[\![Q]\!]$, and that it might be possible to produce a compositional semantics corresponding to this equivalence. It is not clear that the benefits of full abstraction are worth this extra complexity, though.

Acknowledgements

I would like to thank Bill Roscoe and Bernard Sufrin for interesting discussions on this work. I would also like to thank the anonymous referees for a number of very useful suggestions.

References

[1] Peter Welch, Neil Brown, James Morres, Kevin Chalmers, and Bernhard Sputh. Integrating and extending JCSP. In *Communicating Process Architectures*, pages 48–76, 2007.
[2] Peter Welch and Neil Brown. Communicating sequential processes for Java (JCSP). `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`, 2009.
[3] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.
[4] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
[5] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.
[6] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement—FDR 2 User Manual*, 1997.
[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
[8] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.
[9] E. R. Olderog and C. A. R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23(1):9–66, 1986.
[10] A. W. Roscoe. Revivals, stuckness and the hierarchy of CSP models. *Journal of Logic and Algebraic Programming*, 78(3):163–190, 2009.
[11] A. E. Lawrence. Triples. In *Proceedings of Communicating Process Architectures*, pages 157–184, 2004.
[12] C. J. Fidge. A formal definition of priority in CSP. *ACM Transactions on Programming Languages and Systems*, 15(4):681–705, 1993.
[13] Gavin Lowe. *Probabilities and Priorities in Timed CSP*. DPhil thesis, Oxford, 1993.
[14] Gavin Lowe. Probabilistic and prioritized models of Timed CSP. *Theoretical Computer Science*, 138:315–352, 1995.
[15] A.W. Roscoe. On the expressiveness of CSP. Available via `http://web.comlab.ox.ac.uk//files/1383/complete(3).pdf`, 2009.
[16] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

A. Derived Operational Semantics

The definition of the $\longrightarrow\!\!\triangleright$ relation, and the operational semantic rules for the $\longrightarrow$ relation can be translated into the following defining rules for $\longrightarrow\!\!\triangleright$.

$$STOP \xrightarrow{notOffer\, a} STOP \qquad \text{for } a \in \Sigma$$
$$a \to P \xrightarrow{notOffer\, b} a \to P \qquad \text{for } b \in \Sigma$$
$$\breve{a} \to P \xrightarrow{offer\, a} a \to P$$
$$a \to P \xrightarrow{\tau} \breve{a} \to P$$
$$\breve{a} \to P \xrightarrow{a} P$$
$$\breve{a} \to P \xrightarrow{notOffer\, b} a \to P \qquad \text{for } b \neq a$$

if ready a then P else $Q \xrightarrow{ready\ a} P$

if ready a then P else $Q \xrightarrow{notReady\ a} Q$

if ready a then P else $Q \xrightarrow{notOffer\ b}$ if ready a then P else $Q \qquad$ for $b \in \Sigma$

$$\frac{P \xrightarrow{a} P'}{P \,\Box\, Q \xrightarrow{a} P'} \qquad \frac{P \xrightarrow{\tau} P'}{P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q}$$

$$\frac{P \xrightarrow{ready\ a} P'}{P \,\Box\, Q \xrightarrow{ready\ a} P' \,\Box\, Q} \qquad \frac{P \xrightarrow{notReady\ a} P'}{P \,\Box\, Q \xrightarrow{notReady\ a} P' \,\Box\, Q}$$

$$\frac{P \xrightarrow{offer\ a} P'}{P \,\Box\, Q \xrightarrow{offer\ a} P' \,\Box\, Q} \qquad \frac{\begin{array}{l} P \xrightarrow{notOffer\ a} P' \\ Q \xrightarrow{notOffer\ a} Q' \end{array}}{P \,\Box\, Q \xrightarrow{notOffer\ a} P' \,\Box\, Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \,\triangleright\, Q \xrightarrow{a} P'} \qquad \frac{P \xrightarrow{\tau} P'}{P \,\triangleright\, Q \xrightarrow{\tau} P' \,\triangleright\, Q}$$

$$\frac{P \xrightarrow{ready\ a} P'}{P \,\triangleright\, Q \xrightarrow{ready\ a} P' \,\triangleright\, Q} \qquad \frac{P \xrightarrow{notReady\ a} P'}{P \,\triangleright\, Q \xrightarrow{notReady\ a} P' \,\triangleright\, Q}$$

$$\frac{P \xrightarrow{offer\ a} P'}{P \,\triangleright\, Q \xrightarrow{offer\ a} P' \,\triangleright\, Q} \qquad \frac{P \xrightarrow{notOffer\ a} P'}{P \,\triangleright\, Q \xrightarrow{notOffer\ a} P' \,\triangleright\, Q}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \setminus A \xrightarrow{\alpha} P' \setminus A}\ \alpha \in (\Sigma - A)) \cup \{\tau\} \cup ready(\Sigma - A) \cup notReady(\Sigma - A) \cup offer(\Sigma - A) \cup notOffer(\Sigma - A)$$

$$\frac{P \xrightarrow{\alpha} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A}\ \alpha \in A \cup ready\, A \qquad P \setminus A \xrightarrow{notOffer\ a} P \setminus A, \qquad \text{for } a \in A$$

$$\frac{P \xrightarrow{\alpha} P'}{P_A \|_B Q \xrightarrow{\alpha} P'_A \|_B Q}\ \alpha \in private_A \cup \{\tau\} \qquad \frac{\begin{array}{l} P \xrightarrow{\alpha} P' \\ Q \xrightarrow{\alpha} Q' \end{array}}{P_A \|_B Q \xrightarrow{\alpha} P'_A \|_B Q}\ \alpha \in sync_{A,B}$$

$$\frac{\begin{array}{l} P \xrightarrow{ready\ b} P' \\ Q \xrightarrow{offer\ b} Q' \end{array}}{P_A \|_B Q \xrightarrow{ready\ b} P'_A \|_B Q'}\ b \in B \qquad \frac{\begin{array}{l} P \xrightarrow{notReady\ b} P' \\ Q \xrightarrow{notOffer\ b} Q' \end{array}}{P_A \|_B Q \xrightarrow{\tau} P'_A \|_B Q'}\ b \in B$$

$$\frac{\begin{array}{l} P \xrightarrow{notReady\ b} P' \\ Q \xrightarrow{offer\ b} Q' \end{array}}{P_A \|_B Q \xrightarrow{notReady\ b} P'_A \|_B Q'}\ b \in B \qquad P_A \|_B Q \xrightarrow{notOffer\ d} P_A \|_B Q, \quad \text{for } d \in \Sigma - A - B.$$

B. Congruence of the Operational Semantics

In this appendix, we prove some of the cases in the proof of Theorem 13:

$tr \in traces_R[\![P]\!]$ iff $P \stackrel{tr}{\Longrightarrow}$.

Hiding

We prove the case of hiding in Theorem 13 using the derived rules in Appendix A.

(⇒) Suppose $tr \in traces_R[\![P \setminus A]\!]$. Then there exists some $tr_P \in traces_R[\![P]\!]$ such that $tr_P \upharpoonright (notReady\,A \cup offer\,A) = \langle\rangle$ and $tr_P \setminus (A \cup ready\,A) = tr \setminus notOffer\,A$. By the inductive hypothesis, $P \stackrel{tr_P}{\Longrightarrow}$, i.e., $P \stackrel{\alpha_1}{\longrightarrow\!\!\triangleright} \ldots \stackrel{\alpha_n}{\longrightarrow\!\!\triangleright}$ for some $\alpha_1, \ldots, \alpha_n$ such that $tr_P = \langle \alpha_1, \ldots, \alpha_n \rangle \setminus \{\tau\}$. From the derived operational semantics rules, $P \backslash A$ has the same transitions but with each $\alpha_i \in A \cup ready\,A$ replaced by a τ, i.e., transitions corresponding to the trace $tr_P \setminus (A \cup ready\,A)$. Further, using the third derived rule for hiding, arbitrary $notOffer\,A$ self-loops can be added to the transitions, giving transitions corresponding to trace tr. Hence $P \setminus A \stackrel{tr}{\Longrightarrow}$.

(⇐) Suppose $P \setminus A \stackrel{tr}{\Longrightarrow}$. Consider the transitions of P that lead to this trace. By consideration of the derived rules, we see that $P \stackrel{tr_P}{\Longrightarrow}$ for some trace tr_P such that $tr_P \upharpoonright (notReady\,A \cup offer\,A) = \langle\rangle$ and $tr_P \setminus (A \cup ready\,A) = tr \setminus notOffer\,A$. By the inductive hypothesis, $tr_P \in traces_R[\![P]\!]$. Hence, $tr \in traces_R[\![P \setminus A]\!]$.

Parallel Composition

We prove the case of parallel composition in Theorem 13 using the derived rules from Appendix A.

(⇒) Suppose $tr \in traces_R[\![P_A \|_B Q]\!]$. Then there exist $tr_P \in traces_R[\![P]\!]$ and $tr_q \in traces_R[\![Q]\!]$ such that $tr_P \upharpoonright (\Sigma - A) \cup offer(\Sigma - A) \cup notOffer(\Sigma - A) = \langle\rangle$, $tr_Q \upharpoonright (\Sigma - B) \cup offer(\Sigma - B) \cup notOffer(\Sigma - B) = \langle\rangle$ and $(tr_P, tr_Q) \stackrel{A\|_B}{\longmapsto} tr \backslash notOffer(\Sigma - A - B)$. By the inductive hypothesis, $P \stackrel{tr_P}{\Longrightarrow}$ and $Q \stackrel{tr_Q}{\Longrightarrow}$. So $P \stackrel{\alpha_1}{\longrightarrow\!\!\triangleright} \ldots \stackrel{\alpha_n}{\longrightarrow\!\!\triangleright}$ and $Q \stackrel{\beta_1}{\longrightarrow\!\!\triangleright} \ldots \stackrel{\beta_m}{\longrightarrow\!\!\triangleright}$ for some $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m$ such that $tr_P = \langle \alpha_1, \ldots, \alpha_n \rangle \setminus \{\tau\}$ and $tr_q = \langle \beta_1, \ldots, \beta_m \rangle \setminus \{\tau\}$. We then have that $P_A \|_B Q \stackrel{tr \setminus notOffer(\Sigma - A - B)}{\Longrightarrow}$, since each event implied by $(tr_P, tr_Q) \stackrel{A\|_B}{\longmapsto} tr \setminus notOffer(\Sigma - A - B)$ has a corresponding transition implied by the operational semantics rules (formally, this is a case analysis over the clauses of $\stackrel{A\|_B}{\longmapsto}$, combined with a straightforward induction on $m + n$). Further, using the final derived rule for parallel composition, arbitrary $notOffer(\Sigma - A - B)$ self-loops can be added to the transitions, giving transitions corresponding to trace tr. Hence $P_A \|_B Q \stackrel{tr}{\Longrightarrow}$.

(⇐) Suppose $P_A \|_B Q \stackrel{tr}{\Longrightarrow}$. Then by item 3 of Lemma 9, $P_A \|_B Q \stackrel{tr \setminus notOffer(\Sigma - A - B)}{\Longrightarrow}$. Consider the transitions of P and Q that lead to this trace according to the operational semantics rules, say $P \stackrel{\alpha_1}{\longrightarrow\!\!\triangleright} \ldots \stackrel{\alpha_n}{\longrightarrow\!\!\triangleright}$ and $Q \stackrel{\beta_1}{\longrightarrow\!\!\triangleright} \ldots \stackrel{\beta_m}{\longrightarrow\!\!\triangleright}$. Let $tr_P = \langle \alpha_1, \ldots, \alpha_n \rangle \setminus \{\tau\}$ and $tr_q = \langle \beta_1, \ldots, \beta_m \rangle \setminus \{\tau\}$; so $P \stackrel{tr_P}{\Longrightarrow}$ and $Q \stackrel{tr_Q}{\Longrightarrow}$. By the inductive hypothesis, $tr_P \in traces_R[\![P]\!]$ and $tr_Q \in traces_R[\![Q]\!]$. Also, by consideration of the operational semantics rules, $tr_P \upharpoonright (\Sigma - A) \cup offer(\Sigma - A) \cup notOffer(\Sigma - A) = \langle\rangle$ and $tr_Q \upharpoonright (\Sigma - B) \cup offer(\Sigma - B) \cup notOffer(\Sigma - B) = \langle\rangle$. Further, $(tr_P, tr_Q) \stackrel{A\|_B}{\longmapsto} tr \setminus notOffer(\Sigma - A - B)$, since each transition implied by the operational semantics rules has a corresponding event implied by the definition of $\stackrel{A\|_B}{\longmapsto}$ (formally, this is a case analysis over the operational semantics rules, combined with a straightforward induction on $m + n$). Hence $tr \in traces_R[\![P_A \|_B Q]\!]$.