Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Draft, 2013

The Meaning and Implementation of SKIP in CSP

Thomas Gibson-Robinson and Michael Goldsmith

Department of Computer Science, University of Oxford
Email: `thomas.gibson-robinson@cs.ox.ac.uk`
`michael.goldsmith@cs.ox.ac.uk`

Abstract. The CSP model checker FDR has long supported Hoare's termination semantics for CSP, but has not supported the more theoretically complete construction of Roscoe's, largely due to the complexity of adding a second termination semantics. In this paper we provide a method of simulating Roscoe's termination semantics using the Hoare termination semantics and then prove the equivalence of the two different approaches. We also ensure that FDR can support the simulation reasonably efficiently.

Keywords. CSP, FDR, Model Checking, Termination

Introduction

The process algebra CSP [1,2,3] provides a way of composing processes *sequentially*, such that a second process is only started once the first process has *terminated*. There have been many different semantics proposed for termination in CSP. The first semantics was proposed by Hoare in [1]. This semantics is perhaps the most intuitive, and probably the most useful for constructing CSP systems in practice, since parallel compositions terminate only when all components do, providing support for *distributed* termination.

In [2], Roscoe proposes a second termination semantics that that has since become the de-facto formalisation of termination in CSP. Roscoe's primary motivation for defining this second semantics was to admit a correct treatment of the algebraic semantics of termination in CSP, which was known [4] to be problematic[1] under Hoare's termination semantics.

The CSP model checker, FDR [5], has only ever supported the Hoare termination semantics, which is particularly unfortunate as it means that those learning CSP are unable to experiment with the more usual theoretical formalisation of termination. Unfortunately, adding support for a second termination semantics in FDR imposes a large overhead. For efficiency reasons, FDR requires several different definitions of every operator, meaning that an increase in the number of operator variants quickly leads to the number of lines of code increasing. Further, the different termination semantics require different ways of calculating the denotational value of a process (in particular, the failures of a process), which requires changes deep inside FDR. Therefore, for the sake of efficiency and in order to reduce the number of lines of code required, it is highly desirable to develop a solution that allows Rosoce's termination semantics to be simulated using the already implemented termination semantics within FDR.

[1]For example, one would reasonably expect $P \mathbin{;} SKIP = P$, but this is not always the case under Hoare's termination semantics. An example is given at the end of Section 1.3.

Contributions In this paper we propose a method of simulating Roscoe's termination semantics using the Hoare termination semantics that requires only minimal changes to the existing operator definitions. Further, we prove that the simulation is equivalent to the original formulation. Thus, this provides a way of implementing both termination semantics inside of FDR with minimal changes.

Outline of Paper In Section 1, we introduce the relevant fragment of CSP and define the two main termination semantics. In Section 2 we then develop and prove the correctness of a simulation of the Roscoe semantics in the Hoare semantics. Further, this simulation will be efficient, in that it requires minimal changes to the existing operators. We also discuss an optimisation to the transformation that will improve the performance of the simulation under FDR. In Section 3 we summarise the results of the paper.

1. CSP and its Termination Semantics

In this section we give a brief overview of CSP and detail the two standard termination semantics for CSP.

1.1. CSP

CSP [1,2,3] is a *process algebra* in which programs or *processes* that communicate events from a set Σ with an environment may be described. We sometimes structure events by sending them along a *channel* . For example, $c.3$ denotes the value 3 being sent along the channel c. Further, given a channel c the set $\{|c|\} \subseteq \Sigma$ contains those events of the form $c.x$.

The simplest CSP process is the process $STOP$, that can perform no events and thus represents a deadlocked process. The process $a \to P$ offers the environment the event $a \in \Sigma$ and then when it is performed, behaves like P. The process $P \mathbin{\Box} Q$ offers the environment the choice of the events offered by P and by Q. Alternatively, the process $P \sqcap Q$, non-deterministically chooses which of P or Q to behave like. Note that the environment cannot influence the choice, the process chooses *internally*. $P \triangleright Q$ initially offers the choice of the events of P but can *timeout* and then behaves as Q.

The process $P \; {}_A\|_B \; Q$ allows P and Q to perform only events from A and B respectively and forces P and Q to synchronise on events in $A \cap B$. The process $P \underset{A}{\|} Q$ allows P and Q to run in parallel, forcing synchronisation on events in A and allowing arbitrary interleaving of events not in A. The *interleaving* of two processes, denoted $P \mathbin{|||} Q$, runs P and Q in parallel but enforces no synchronisation. The process $P \setminus A$ behaves as P but hides any events from A by transforming them into a special internal event, τ. This event does not synchronise with the environment and thus can always occur. The process $P[\![R]\!]$, behaves as P but renames the events according to the relation R. Hence, if P can perform a, then $P[\![R]\!]$ can perform each b such that $(a, b) \in R$, where the choice (if more than one such b) is left to the environment (like $\Box$). The process $P \mathbin{\triangle} Q$ initially behaves like P but allows Q to *interrupt* at any point and perform an event, at which point P is discarded and the process behaves like Q. The process $P \mathbin{\Theta_A} Q$ initially behaves like P, but if P ever performs an event from A (the *exception* set), P is discarded and $P \mathbin{\Theta_A} Q$ behaves like Q.

Recursive processes can be defined either equationally or using the notation $\mu X \cdot P$. In the latter, every occurrence of X within P represents a recursive call to $\mu X \cdot P$.

Non-determinism can arise in a variety of ways in CSP processes. For example, non-determinism can be explicitly introduced via operators such as the internal choice operator (providing the arguments are semantically distinct). Further (and more subtly), other operators can introduce non-determinism when combined in certain ways. For example, $a \to STOP \mathbin{\Box} a \to b \to STOP$ is non-deterministic since b can be both accepted and refused after performing an a. For a more thorough description of non-determinism see, e.g. [3].

There are a number of ways of giving meaning to CSP processes. The simplest approach is to give an operational semantics. The operational semantics of a CSP process naturally creates a labelled transition system (LTS) where the edges are labelled by events from $\Sigma \cup \{\tau\}$ and the nodes are process states. The usual way of defining the operational semantics of CSP processes is by presenting *Structured Operational Semantics* (SOS) style rules in order to define the transition relation $\xrightarrow{e}$ for $e \in \Sigma \cup \{\tau\}$. For instance, the operational semantics of the exception operator can be defined by the following inductive rules:

$$\frac{P \xrightarrow{a} P'}{P \;\Theta_A\; Q \xrightarrow{a} Q}\; a \in A \qquad \frac{P \xrightarrow{b} P'}{P \;\Theta_A\; Q \xrightarrow{b} P' \;\Theta_A\; Q}\; b \notin A \qquad \frac{P \xrightarrow{\tau} P'}{P \;\Theta_A\; Q \xrightarrow{\tau} P' \;\Theta_A\; Q}$$

The interesting rule is the first, which specifies that if P performs an event $a \in A$, then $P \;\Theta_A\; Q$ can perform the event a and then behave like Q (i.e. the exception has been *thrown*). The last rule is known as a *tau-promotion* rule as it promotes any τ performed by a component (in this case P) into a τ performed by the operator. The justification for this rule is that τ is an unobservable event, and therefore the environment cannot prevent P from performing the event. Note that τs from Q are not promoted, since Q is not active. Formally, an argument P of a CSP operator Op is **on** iff it can perform an event, i.e. there exists a SOS rule of the form:

$$\frac{P \xrightarrow{a} P'}{Op(\ldots, P, \ldots) \xrightarrow{a} Op(\ldots)}$$

P is **off** iff no such rule exists. For example, the left argument of the exception operator is **on**, whilst the right argument is **off**. Also, given that the SOS rules for internal choice are:

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

it follows that both arguments of $\sqcap$ are **off**. Conversely, both arguments of $\Box$ are **on**.

CSP also has a number of *denotational models*, such as the traces, failures and failures-divergences models. In these models, each process is represented by a set of behaviours. Two processes are equal in a denotational model iff they have the same set of behaviours. In this paper we consider only the traces and failures models. In the traces model, a process is modelled by the set of finite traces (which are sequences of events from Σ) that it can perform. In the failures model, a process is represented by a set of pairs, each consisting of a trace and a set of events that the process can stably (i.e. the process must not be able to perform a τ) refuse to perform after the trace.

Notation Given a sequence $tr \in A^*$ and $X \subseteq A$, the *restriction of* tr *to* X, denoted $tr \upharpoonright X$, is inductively defined by removing events not in X, as follows:

$$\langle\rangle \upharpoonright X = \langle\rangle$$
$$(\langle x\rangle^\frown xs) \upharpoonright X = \begin{cases} \langle x\rangle^\frown(xs \upharpoonright X) & \text{if } x \in X \\ xs \upharpoonright X & \text{otherwise} \end{cases}$$

If $tr \in (\Sigma \cup \{\tau\})^+$, $P \overset{tr}{\longmapsto} Q$ iff there exist $P_1, \ldots, P_N = Q$, where $N = |tr|$, $tr = \langle a_1, \ldots, a_N\rangle$ and such that $P_1 \xrightarrow{a_1} P_2 \xrightarrow{a_2} \ldots \xrightarrow{a_N} P_N$. If $tr \in \Sigma^+$, $P \overset{tr}{\Longrightarrow} Q$ iff there exists tr' such that $tr' \upharpoonright \Sigma = tr$ and $P \overset{tr'}{\longmapsto} Q$. $P \overset{e}{\not\longrightarrow}$ denotes that there does not exist a process P' such that $P \xrightarrow{e} P'$.

1.2. Termination in CSP

CSP also allows processes to be composed sequentially. The *sequential composition of* P *and* Q, denoted $P\,;Q$, runs P until it terminates, at which point Q is run. CSP defines the process $SKIP$ as the process that immediately terminates. Thus, the process $P = a \to SKIP$ is the process that terminates after performing an a, whilst the process $Q = P\,;Q$ is a process that never terminates, but performs an infinite number of a's (it is equivalent to $R \mathrel{\widehat{=}} a \to R$). The process Ω represents a process that has terminated. Note that whilst this is (operationally) equivalent to $STOP$, we require a syntactically distinct process in order to define Roscoe's termination semantics. The reason that this is required will become clear when the operational semantics of $|||$ are defined under the Roscoe termination semantics in Section 1.4.

In order to define formally the semantics of termination, the event $\checkmark \notin \Sigma$ is added and $SKIP$ is defined as $\checkmark \to \Omega$. The operational semantics of sequential composition are defined as:

$$\frac{P \xrightarrow{a} P'}{P\,;Q \xrightarrow{a} P'\,;Q}\ a \in \Sigma \cup \{\tau\} \qquad \frac{P \xrightarrow{\checkmark} \Omega}{P\,;Q \xrightarrow{\tau} Q}$$

Note that $\checkmark$, since it represents termination, should only ever appear at the end of a trace. Clearly, the above definition of sequential composition respects this.

In order to define termination fully in CSP, we need to define how each of the CSP operators treat a $\checkmark$ performed by one of their arguments. If an operator has no **on** arguments (e.g. $\sqcap$ and $\to$), then these processes are simply defined as non-terminating. If an operator other than sequential composition (which has a special definition, due to its central role in termination) has one **on** argument (e.g. $\Theta.$, $\setminus\,\cdot$, $[\![\cdot]\!]$ and $\triangleright$), then the operator terminates precisely when its **on** argument does (thus termination is *promoted*). The remaining CSP operators, which have two **on** arguments, are grouped into one of the following two categories with regards to how they treat termination:

Independent Operators that treat termination *independently* terminate whenever *any* of their **on** arguments terminate (e.g. $\Box$ and $\triangle$);

Synchronising Operators that *synchronise* termination terminate only when *all* of their **on** arguments terminate (e.g. $|||$, $\underset{A}{\|}$ and ${}_A\|_B$).

The semantics we consider in this paper treat $\Box$ and $\triangle$ independently. This makes sense intuitively: since neither of these operators actually runs its arguments in parallel, but instead allows actions from one to occur and then discards the other, at some point.

Independent operators, by definition, simply terminate when either of their arguments do. Therefore, the termination operational semantics of independent operators do not vary between different termination semantics, and we can therefore define them now. We formally define the semantics for $\Box$: the operational semantics for other independent operators can be defined analogously:

$$\frac{P \xrightarrow{\checkmark} \Omega}{P \Box Q \xrightarrow{\checkmark} \Omega} \qquad \frac{Q \xrightarrow{\checkmark} \Omega}{P \Box Q \xrightarrow{\checkmark} \Omega}$$

All of the CSP parallel operators (i.e. $|||$, $\|$ etc.) synchronise termination of their arguments[2]. Thus, a parallel composition terminates only when all of its arguments do, which can be useful when constructing systems since it corresponds to distributed termination. In the following sections we describe how the two different termination semantics define the semantics of the synchronising operators.

[2]Whilst this is true of the standard CSP semantics, in [6] Davies defines a version of interleave that does not synchronise on termination, but instead treats it independently. The results of this paper could equally be applied to treat interleave's termination independently instead, but we concentrate on the more usual semantics.

1.3. ✓-as-Refusable

The Hoare semantics, as used by FDR, treats ✓ as a *refusable* event. Thus, henceforth we refer to this as the the *✓-as-Refusable* semantics. The motivation for this semantics is that a process should be able to offer to the environment the option of terminating, but the environment should be able to decide whether the process terminates or not. For example, consider the process $\mathit{SKIPChoice}_a \mathrel{\widehat{=}} \mathit{SKIP} \mathrel{\Box} a \rightarrow \mathit{STOP}$ (which we use as a running example): under the ✓-as-Refusable semantics, this process offers the environment the choice of either terminating, or performing the event a.

Under the ✓-as-Refusable semantics, synchronising operators only terminate when both of their arguments are ready to terminate, and the termination is performed in lock-step. The operational semantics for termination of $|||$ is defined as follows. Again, the operational semantics for other synchronising operators can be defined analogously.

$$\frac{P \xrightarrow{\checkmark} \Omega \wedge Q \xrightarrow{\checkmark} \Omega}{P \mathbin{|||} Q \xrightarrow{\checkmark} \Omega}$$

For example, the above process $\mathit{SKIPChoice}_a$ can terminate immediately. Equally, $\mathit{SKIPChoice}_a \mathbin{|||} \mathit{STOP} = a \rightarrow \mathit{STOP}$, since the right-hand argument can never terminate and thus blocks $\mathit{SKIPChoice}_a$ from terminating. Conversely, $\mathit{SKIPChoice}_a \underset{\{a\}}{\parallel} \mathit{SKIP} = \mathit{SKIP}$, since the right-hand argument cannot perform an a and thus blocks the left-hand argument from performing it.

The value of a process in the denotational models is also affected by the termination semantics in use. Under any of the termination semantics, the traces of a process can be easily extracted using:

$$\mathit{traces}(P) \mathrel{\widehat{=}} \{tr \mid \exists P' \cdot P \overset{tr}{\Longrightarrow} P'\}.$$

The failures of a process P under the ✓-as-Refusable semantics, denoted $\mathit{failures}^{r}(P)$, can easily be extracted from the operational semantics:

$$\mathit{failures}^{r}(P) \mathrel{\widehat{=}} \{(tr, X) \mid \exists Q \cdot P \overset{tr}{\Longrightarrow} Q \wedge X \subseteq \Sigma \cup \{\checkmark\} \wedge Q \mathbf{\ ref\ } X\}.$$

where $Q \mathbf{\ ref\ } X$ iff Q is stable (i.e. $Q \overset{\tau}{\not\longrightarrow}$), and, $\forall x \in X \cdot Q \overset{x}{\not\longrightarrow}$.

For example, considering $P \mathrel{\widehat{=}} a \rightarrow \mathit{STOP} \sqcap b \rightarrow \mathit{STOP}$, $(\langle\rangle, \{a\}) \in \mathit{failures}^{r}(P)$ because, although the initial state of P is unstable, $P \xrightarrow{\tau} (b \rightarrow \mathit{STOP})$. Further, since $b \rightarrow \mathit{STOP}$ is stable it follows that $(\langle\rangle, \{a\}) \in \mathit{failures}^{r}(b \rightarrow \mathit{STOP})$ and hence $(\langle\rangle, \{a\}) \in \mathit{failures}^{r}(P)$ since $P \overset{\langle\rangle}{\Longrightarrow} b \rightarrow \mathit{STOP}$.

A direct consequence of the above definitions is that P is not necessarily equal to $P\,;\,\mathit{SKIP}$, which might reasonably be expected. For example, $\mathit{SKIPChoice}_a\,;\,\mathit{SKIP}$ is equal to $a \rightarrow \mathit{STOP} \rhd \mathit{SKIP}$, which is not equal to $\mathit{SKIPChoice}_a$. This is because:

$$(\mathit{SKIPChoice}_a\,;\,\mathit{SKIP}) \xrightarrow{\tau} \mathit{SKIP}$$

since $\mathit{SKIPChoice}_a$ can perform a ✓ which is converted into a τ by ;. Hence $(\langle\rangle, \{a\}) \in \mathit{failures}^{r}(\mathit{SKIPChoice}_a\,;\,\mathit{SKIP})$ since $\mathit{SKIPChoice}_a\,;\,\mathit{SKIP} \overset{\langle\rangle}{\Longrightarrow} \mathit{SKIP}$ and $(\langle\rangle, \{x\}) \in \mathit{failures}^{r}(\mathit{SKIP})$ for all $x \in \Sigma$. However, $(\langle\rangle, \{a\}) \notin \mathit{failures}^{r}(\mathit{SKIPChoice}_a)$ since $\mathit{SKIPChoice}_a$ is stable and explicitly offers an a, meaning the two sides are not equal.

1.4. ✓-as-Signal

Roscoe's termination semantics for CSP_M, known as the *✓-as-Signal* semantics, instead treats ✓ as an event that cannot be refused by the environment. Thus, when a process decides that it is terminating, the environment is not allowed to prevent termination: instead the process immediately terminates and signals this to the environment via the ✓ event. Thus, in the ✓-as-Refusable semantics, the offer of a ✓ can be thought of as a communication that the process can terminate if desired, whilst in the ✓-as-Signal semantics, the offer of a ✓ means the process can terminate on its own accord.

The operational semantics for termination of $|||$ is defined as follows. Again, the operational semantics for other synchronising operators can be defined analogously.

$$\frac{P \xrightarrow{\checkmark} \Omega}{P \mathbin{|||} Q \xrightarrow{\tau} \Omega \mathbin{|||} Q} \qquad \frac{Q \xrightarrow{\checkmark} \Omega}{P \mathbin{|||} Q \xrightarrow{\tau} P \mathbin{|||} \Omega} \qquad \frac{Q \xrightarrow{\checkmark} \Omega}{\Omega \mathbin{|||} Q \xrightarrow{\tau} SKIP} \qquad \frac{P \xrightarrow{\checkmark} \Omega}{P \mathbin{|||} \Omega \xrightarrow{\tau} SKIP}$$

In the above, the first two clauses ensure that as soon as an argument wishes to terminate, it is allowed to do so. This is in contrast to the ✓-as-Refusable semantics, in which an argument is not allowed to terminate until the other argument is ready to do so. The second two clauses specify that as soon as one argument has terminated, if the other argument wishes to terminate then the whole operator becomes $SKIP$, which can immediately terminate.

The above rules also illustrate the need for Ω and $STOP$ to be syntactically distinct. Clearly, $STOP \mathbin{|||} SKIP$ should not be allowed to terminate since $STOP$ cannot terminate. However, if the last two rules above used $STOP$ rather than Ω, then this would allow the interleave to erroneously terminate. Thus, we need $STOP$ and Ω to be syntactically distinct so that we can differentiate between a process that has terminated via a tick and a process that has just deadlocked, so we can define the termination semantics of synchronising operators.

The most important difference between the two semantics is how the denotational values are extracted from the operational semantics. As stated above, Roscoe views termination as something a process tells the environment it is going to do, rather than something the environment can choose. Therefore, after a trace, if a process can terminate (i.e. a ✓ is available), then it should be able to refuse to communicate anything apart from the ✓ (i.e. all events from Σ). Thus, the failures of a process P under the ✓-as-Signal semantics, denoted $failures^s(P)$, contains an extra clause versus $failures^r(P)$ in order to add these extra failures:

$$\begin{aligned} failures^s(P) \mathrel{\widehat{=}} \{(tr, X) \mid \exists\, Q \cdot P \overset{tr}{\Longrightarrow} Q \wedge X \subseteq \Sigma \cup \{\checkmark\} \wedge Q \;\mathbf{ref}\; X\} \\ \cup \{(tr, X) \mid P \xRightarrow{tr^\frown\langle\checkmark\rangle} \Omega, X \subseteq \Sigma\} \end{aligned} \tag{1.1}$$

However, as noted above, the traces extracted from the two semantics are identical.

The addition of the above failures has a number of interesting consequences for the behaviour of processes that involve choices between ✓ and visible events. For example, consider the process $SKIPChoice_a$ defined in Section 1.3. Under the ✓-as-Signal semantics this now has the failure $(\langle\rangle, \{a\})$ (since it can perform a ✓ immediately) which was not a failure under the ✓-as-Refusable semantics. Thus, $SKIPChoice_a \underset{\{a\}}{\parallel} a \to STOP = a \to STOP \rhd STOP$ under the ✓-as-Signal semantics, since $SKIPChoice_a$ may possibly refuse the a which would cause a deadlock. Under the ✓-as-Refusable semantics $SKIPChoice_a \underset{\{a\}}{\parallel} a \to STOP = a \to STOP$, since the ✓ can just be ignored. Equally, $SKIPChoice_a \underset{\{a\}}{\parallel} SKIP = SKIP$ under both semantics since environmental control over the a is retained. This suggests the most important difference between the two semantics: under the ✓-as-Signal semantics the environment is unable to stop ✓ from occurring whilst under the ✓-as-Refusable semantics the environment is free to choose ✓ just like any other visible event.

Recall that under the ✓-as-Refusable semantics, it is not necessarily true that $P = P \mathbin{;} SKIP$. In particular, recall that $SKIPChoice_a \mathbin{;} SKIP$ is actually equal to $a \to STOP \rhd$

$SKIP$. However, under the ✓-as-Signal semantics $a \to STOP \triangleright SKIP = SKIPChoice_a$, since the extra failures that $SKIPChoice_a$ has (as per Equation 1.1) mean that the a can be refused, as it can in $a \to STOP \triangleright SKIP$. More generally, $P = P \mathbin{;} SKIP$ since in any state where the ✓ can be performed by P, all events other than ✓ can already be refused.

2. Simulating ✓-as-Signal

We now consider how to simulate the ✓-as-Signal semantics under the ✓-as-Refusable semantics. Intuitively, the difference between the two different semantics is that the ✓-as-Refusable semantics just treats ✓ as a regular event and therefore, in $SKIPChoice_a = SKIP \mathrel{\Box} a \to STOP$, offers the deterministic choice between the ✓ and the a. In the ✓-as-Signal semantics, whilst the choice is still offered, it is no longer a deterministic choice since P can refuse the a (cf. Equation 1.1).

In order to define our translation, we firstly consider how operators with Independent termination semantics are affected. Firstly, consider the process: $SKIPChoice_a \mathbin{|||} STOP$. The LTSs for this process under the two different semantics are given in Figure 1.

(a) ✓-as-Refusable (b) ✓-as-Signal

Figure 1. The LTS of the process $SKIPChoice_a \mathbin{|||} STOP$ under different termination semantics.

Note, in particular, that under the ✓-as-Signal semantics, this process can refuse to do an a, but under the ✓-as-Refusable semantics, it cannot. Thus, it follows that to simulate this process correctly in the ✓-as-Signal semantics, somehow the process must be altered to allow the a to be refused. In the above example, the only way that the a can be refused is if the external choice is resolved[3]. In order to solve this we add a new *resolving tau* event, denoted τ_r, that is treated by all operators precisely as they treat ✓. Thus, in the above example, a τ_r would cause the $\Box$ to be resolved, leading to a state in which only ✓ is offered and, in particular, refuses the a as required. In our translation, if we arrange that τ_r always (and only) occurs directly before a ✓, then it follows that we can correctly resolve $\Box$ (and other operators with Independent termination semantics) and introduce the failures required by Equation 1.1.

We now discuss how to simulate correctly operators that have synchronising termination semantics. In Section 1.4, we saw that $SKIPChoice_a \mathbin{|||} STOP$ is equivalent under ✓-as-Refusable semantics to $a \to STOP$, whilst under ✓-as-Signal semantics it is equivalent to $a \to STOP \triangleright STOP$ (since $SKIPChoice_a$ can decide to terminate independently). We can achieve the same effect by simulating the left hand side as $SKIPChoice_a \mathbin{;} SKIP$, thus allowing the termination to resolve the external choice and introduce the failures required by Equation 1.1. Note that this is essentially taking advantage of the equivalence $P = P \mathbin{;} SKIP$ that holds under the ✓-as-Signal semantics, but not in the ✓-as-Refusable semantics.

Outline In Section 2.1 we formalise the translation that we have sketched above. We then prove the equivalence of the translation in Section 2.2 and then discuss the efficency of the translation in the context of FDR in Section 2.3.

[3] An alternative method of solving this problem is to simulate each external choice $P \mathrel{\Box} Q$ as $(P \mathrel{\Box} Q);SKIP$. The problem with this simulation is that it produces processes that are much larger than the original process, and more problematically, will cause FDR to be unable to compile many recursive definitions.

2.1. Formalising The Translation

We now formalise the above translation, as follows. Firstly, we define how the new resolving tau τ_r is treated by the existing operators (by assumption, $\tau_r \notin \Sigma$, so none of the existing operational semantic rules apply). We define $BSkip \mathrel{\widehat{=}} \tau_r \to \checkmark \to \Omega$. In order to ensure that a τ_r occurs *only* directly before a ✓, we add the following additional rule to $P \mathbin{;} Q$:

$$\frac{P \xrightarrow{\tau_r} P'}{P \mathbin{;} Q \xrightarrow{\tau} P' \mathbin{;} Q}$$

As a result of this rule, note that $P \mathbin{;} Q$ can only perform a τ_r if Q can after P has terminated (since any τ_r of P is converted into a τ). Thus, assuming Q only performs a τ_r directly before a ✓, the desired property immediately follows. Further, to each other operator we add an extra rule to promote τ_r exactly as τ is promoted. Note that if none of the arguments of some CSP operator perform a τ_r, then the above changes do not alter the semantics of the operator at all. This should simplify the implementation of these rules.

In order to simplify the formalisation of the transformation, we consider only a restricted subset of CSP that includes $STOP$, $STOP$, $\to$, ;, $\Box$ and $|||$. Note that this includes all categories of operators (in terms of their termination semantics), and thus the proof and transformation could easily be generalised to all CSP operators.

Definition 2.1. The *✓-as-Signal translation function*, Sig, is defined on CSP processes as follows:

$$\begin{aligned}
Sig(SKIP) &\mathrel{\widehat{=}} BSkip \\
Sig(STOP) &\mathrel{\widehat{=}} STOP \\
Sig(a \to P) &\mathrel{\widehat{=}} a \to Sig(P) \\
Sig(P \mathbin{\Box} Q) &\mathrel{\widehat{=}} Sig(P) \mathbin{\Box} Sig(Q) \\
Sig(P \mathbin{;} Q) &\mathrel{\widehat{=}} Sig(P) \mathbin{;} Sig(Q) \\
Sig(P \mathbin{|||} Q) &\mathrel{\widehat{=}} (Sig(P) \mathbin{;} BSkip) \mathbin{\underset{\{\tau_r\}}{\|}} (Sig(Q) \mathbin{;} BSkip)
\end{aligned}$$

As discussed informally, we ensure that a τ_r occurs exactly once before a ✓ by redefining $SKIP$ in the obvious way. We then ensure that $|||$ synchronises on τ_r to ensure that only one τ_r can occur. We prove that this holds in Lemma 2.3.

Notation In the following we use $\Sigma^{\checkmark}$ to denote $\Sigma \cup \{\checkmark\}$, Σ^{τ_r} to denote $\Sigma \cup \{\tau_r\}$ and $\Sigma^{\checkmark,\tau_r}$ to denote $\Sigma \cup \{\checkmark, \tau_r\}$.

2.2. Proving Equivalence

We now prove that our translation gives the correct result. In particular, since we are interested in using our translation in the context of FDR, we check that the denotational value of a process P under the ✓-as-Signal semantics is the same as the value of $Sig(P) \setminus \{\tau_r\}$[4] under the ✓-as-Refusable semantics.

In order to prove that the translation produces the correct denotational values, we firstly prove a couple of results about the operational semantics. Together, these essentially prove that τ_r functions as discussed above. Firstly we prove that τ_r's always occur directly before ✓'s.

[4]Clearly we need to hide the newly introduced τ_r event, otherwise we will obtain extra traces.

Lemma 2.2. $(Sig(P) \setminus \{\tau_r\}) \xRightarrow{\langle\checkmark\rangle} \Omega$ iff $Sig(P) \xRightarrow{\langle\tau_r,\checkmark\rangle} \Omega$.

Proof (Sketch). This follows by a trivial structural induction over P, using the definitions of the modified operational semantics and the definition of Sig. In particular, note that $BSkip$ always performs a τ_r immediately before a ✓. Otherwise, if an operator performs a ✓, then it must not be ; (since ; cannot perform a ✓), and thus it must be because an argument performs a ✓. Thus the inductive hypothesis applies, noting that all operators promote τ_r. □

We now prove a second operational result, and prove that if a process can do a τ_r, then the resulting state must be operationally equivalent to $SKIP$ (i.e. it can do a ✓, but can perform no other event).

Lemma 2.3. $Sig(P) \xRightarrow{\langle\tau_r\rangle} X$ iff $X \xrightarrow{\checkmark} \Omega$ and, for all $a \in \Sigma^{\tau_r}$, $X \not\xrightarrow{a} X'$, for some X'.

Proof (Sketch). This follows by a trivial induction over P, noting that the translation ensures that whenever τ_r occurs, the resulting state is equivalent to $SKIP$. □

Using the above we can now prove that our translation produces the correct traces.

Theorem 2.4. $traces(P) = traces(Sig(P) \setminus \{\tau_r\})$

Proof. This follows by a trivial induction on P, noting that $traces(SKIP) = traces(BSkip \setminus \{\tau_r\})$. □

We can now prove our that the translation produces the correct failures.

Theorem 2.5. $\mathit{failures}^s(P) = \mathit{failures}^r(Sig(P) \setminus \{\tau_r\})$.

Proof. We prove the lemma by structural induction on P. We elide the case for $STOP$ since it is trivial.

$P = SKIP$**:**

$$\begin{aligned}
&\mathit{failures}^r(Sig(SKIP) \setminus \{\tau_r\}) \\
&= \mathit{failures}^r(BSkip \setminus \{\tau_r\}) \\
&= \mathit{failures}^r(SKIP \setminus \{\tau_r\}) \\
&= \mathit{failures}^r(SKIP) \\
&= \mathit{failures}^s(SKIP)
\end{aligned}$$

$P = a \rightarrow Q$**:**

$$\begin{aligned}
&\mathit{failures}^s(a \rightarrow Q) \\
&= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\checkmark,\tau_r}, a \notin X\} \\
&\qquad \cup \{(\langle a\rangle^\frown tr, X) \mid (tr, X) \in \mathit{failures}^s(Q)\} \\
&= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\checkmark,\tau_r}, a \notin X\} \\
&\qquad \cup \{(\langle a\rangle^\frown tr, X) \mid (tr, X) \in \mathit{failures}^r(Sig(Q) \setminus \{\tau_r\})\} && \langle\text{IH}\rangle \\
&= \mathit{failures}^r(a \rightarrow (Sig(Q)) \setminus \{\tau_r\}) \\
&= \mathit{failures}^r(Sig(a \rightarrow Q) \setminus \{\tau_r\})
\end{aligned}$$

$P = Q \,\Box\, R$:

$$
\begin{aligned}
&\mathit{failures}^s(Q \,\Box\, R) \\
&= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathit{failures}^s(Q) \cap \mathit{failures}^s(R)\} \\
&\quad \cup \{(tr, X) \mid (tr, X) \in \mathit{failures}^s(Q) \\
&\qquad\qquad \cup \mathit{failures}^s(R), tr \neq \langle\rangle\} \\
&\quad \cup \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\tau_r} \wedge \langle\checkmark\rangle \in traces(Q) \cup traces(R)\} \\
&= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathit{failures}^r(Sig(Q) \setminus \{\tau_r\}) \\
&\qquad\qquad \cap \mathit{failures}^r(Sig(R) \setminus \{\tau_r\})\} \\
&\quad \cup \{(tr, X) \mid (tr, X) \in \mathit{failures}^r(Sig(Q) \setminus \{\tau_r\}) \\
&\qquad\qquad \cup \mathit{failures}^r(Sig(R) \setminus \{\tau_r\}), tr \neq \langle\rangle\} \\
&\quad \cup \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\tau_r} \wedge \langle\checkmark\rangle \in traces(Sig(Q) \setminus \{\tau_r\}) \\
&\qquad\qquad \cup traces(Sig(R) \setminus \{\tau_r\})\} && \langle\text{IH, Theorem 2.4}\rangle \\
&= \{(\langle\rangle, X) \mid tr \neq \langle\rangle \wedge \exists\, tr \in \{\tau_r\}^* \cdot \\
&\qquad (tr, X \cup \{\tau_r\}) \in \mathit{failures}^r(Sig(Q)) \\
&\qquad\qquad \cap \mathit{failures}^r(Sig(R))\} \\
&\quad \cup \{(tr, X) \mid \exists\, tr' \cdot tr' \upharpoonright \Sigma^{\checkmark} = tr \\
&\qquad\qquad \wedge (tr', X \cup \{\tau_r\}) \in \mathit{failures}^r(Sig(Q)) \\
&\qquad\qquad\qquad \cup \mathit{failures}^r(Sig(R))\} \\
&\quad \cup \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\tau_r} \wedge \langle\tau_r, \checkmark\rangle \in traces(Sig(Q)) \\
&\qquad\qquad\qquad \cup traces(Sig(R)\} && \langle\text{Lemma 2.2}\rangle \\
&= \{(\langle\rangle, X) \mid \exists\, tr \in \{\tau_r\}^* \cdot \\
&\qquad (tr, X \cup \{\tau_r\}) \in \mathit{failures}^r(Sig(Q)) \\
&\qquad\qquad \cap \mathit{failures}^r(Sig(R))\} \\
&\quad \cup \{(tr \upharpoonright \Sigma^{\checkmark}, X) \mid (tr, X \cup \{\tau_r\}) \in \mathit{failures}^r(Sig(Q)) \\
&\qquad\qquad \cup \mathit{failures}^r(Sig(R)) \wedge tr \neq \langle\rangle\} && \langle\dagger\rangle \\
&= \{(tr \upharpoonright \Sigma^{\checkmark}, X) \mid (tr, X \cup \{\tau_r\}) \in \mathit{failures}^r(Sig(Q) \,\Box\, (R))\} \\
&= \mathit{failures}^r((Sig(Q) \,\Box\, Sig(R)) \setminus \{\tau_r\}) \\
&= \mathit{failures}^r(Sig(Q \,\Box\, R) \setminus \{\tau_r\})
\end{aligned}
$$

We can prove that the step marked † holds by proving the following equivalence:

$$
\begin{aligned}
&\{(tr, X) \mid tr \neq \langle\rangle \wedge \exists\, tr' \cdot tr' \upharpoonright \Sigma^{\checkmark} = tr \wedge \\
&\qquad (tr', X \cup \{\tau_r\}) \in \mathit{failures}^r(Sig(Q)) \cup \mathit{failures}^r(Sig(R))\} \\
&\quad \cup \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\tau_r} \wedge \langle\tau_r, \checkmark\rangle \in traces(Sig(Q)) \cup traces(Sig(R))\} \\
&= \{(tr \upharpoonright \Sigma^{\checkmark}, X) \mid (tr, X \cup \{\tau_r\}) \in \mathit{failures}^r(Sig(Q)) \cup \mathit{failures}^r(Sig(R)) \\
&\qquad \wedge tr \neq \langle\rangle\}
\end{aligned}
$$

Firstly, suppose (tr, X) is a member of the left-hand equation. Then, if (tr, X) is a member of the first clause, $tr \neq \langle\rangle$ and there exists tr' such that $tr' \upharpoonright \Sigma^{\checkmark} = tr$ and

$(tr', X \cup \{\tau_r\}) \in \mathit{failures}^r(\mathit{Sig}(Q))$. Hence, (tr, X) is a member of the right-hand equation, as required. Otherwise, (tr, X) must be a member of the second clause, and hence $tr = \langle\rangle$, $X \subseteq \Sigma^{\tau_r}$ and $\langle \tau_r, \checkmark \rangle \in traces(\mathit{Sig}(Q)) \cup traces(\mathit{Sig}(R))$. WLOG, assume there exists S such that $\mathit{Sig}(Q) \stackrel{\langle\tau_r\rangle}{\Longrightarrow} S \stackrel{\langle\checkmark\rangle}{\Longrightarrow} \Omega$ (note that the only state that can occur after $\checkmark$ is Ω). Hence, by Lemma 2.3, $(\langle\tau_r\rangle, Y) \in \mathit{failures}^r(S)$ for any $Y \subseteq \Sigma^{\tau_r}$. Hence, as $X \subseteq \Sigma^{\tau_r}$, it follows that $(\langle\tau_r\rangle, X \cup \{\tau_r\}) \in \mathit{failures}^r(S)$, and thus $(\langle\rangle, X)$ is a failure of the right-hand equation, as required
Otherwise, suppose $(tr \upharpoonright \Sigma^{\checkmark}, X)$ is a member of the right-hand equation. Thus, $tr \neq \langle\rangle$ and $(tr, X \cup \{\tau_r\}) \in \mathit{failures}^r(\mathit{Sig}(Q)) \cup \mathit{failures}^r(\mathit{Sig}(R))$. WLOG, assume $(tr, X \cup \{\tau_r\}) \in \mathit{failures}^r(\mathit{Sig}(Q))$. There are two cases to consider. Firstly, assume $tr \upharpoonright \Sigma^{\checkmark} \neq \langle\rangle$. Hence, trivially it follows that $(tr \upharpoonright \Sigma^{\checkmark}, X)$ is a member of the first clause of the left-hand equation. Otherwise, $tr \upharpoonright \Sigma^{\checkmark} = \langle\rangle$, and hence tr must consist of one or more τ_r's. By Lemma 2.3, $tr = \langle\tau_r\rangle$, and, letting S be such that $\mathit{Sig}(Q) \stackrel{\langle\tau_r\rangle}{\Longrightarrow} S$, $S \stackrel{\checkmark}{\longrightarrow} \Omega$. Hence, $\langle\tau_r, \checkmark\rangle \in traces(\mathit{Sig}(Q))$. Further, since the above holds for any such S, it follows $\checkmark$ cannot be refused after τ_r and thus $X \subseteq \Sigma^{\tau_r}$. Hence, $(\langle\rangle, X)$ is a member of the second clause of the left-hand equation, as required.

$P = Q \,;\, R$:

$$
\begin{aligned}
&\mathit{failures}^s(Q \,;\, R) \\
&= \{(tr, X) \mid (tr, X \cup \{\checkmark\}) \in \mathit{failures}^s(Q)\} \\
&\quad \cup \{(tr^\frown tr', X) \mid tr^\frown\langle\checkmark\rangle \in traces(Q) \\
&\qquad\qquad\qquad\qquad \wedge (tr', X) \in \mathit{failures}^s(R)\} \\
&= \{(tr, X) \mid (tr, X \cup \{\checkmark\}) \in \mathit{failures}^r(\mathit{Sig}(Q) \setminus \{\tau_r\})\} \\
&\quad \cup \{(tr^\frown tr', X) \mid tr^\frown\langle\checkmark\rangle \in traces(\mathit{Sig}(Q) \setminus \{\tau_r\}) \\
&\qquad\qquad\qquad\qquad \wedge (tr', X) \in \mathit{failures}^s(\mathit{Sig}(R) \setminus \{\tau_r\})\} \quad \langle\text{IH, Theorem 2.4}\rangle \\
&= \mathit{failures}^r((\mathit{Sig}(Q) \setminus \{\tau_r\}) \,;\, (\mathit{Sig}(R) \setminus \{\tau_r\})) \\
&= \mathit{failures}^r(\mathit{Sig}(Q \,;\, R) \setminus \{\tau_r\}).
\end{aligned}
$$

$P = Q \,|||\, R$:

In order to prove this case we firstly define, using the definition from [2], the set of all interleavings of two traces s and t, denoted $s \,|||\, t$:

$$
\begin{aligned}
s^\frown\langle\checkmark\rangle \,|||\, t^\frown\langle\checkmark\rangle &= \{u^\frown\langle\checkmark\rangle \mid u \in s \,|||\, t\} \\
s^\frown\langle\checkmark\rangle \,|||\, t &= s \,|||\, t \\
s \,|||\, t^\frown\langle\checkmark\rangle &= s \,|||\, t \\
s \,|||\, \langle\rangle &= \{s\} \\
\langle\rangle \,|||\, t &= \{t\} \\
\langle x\rangle^\frown xs \,|||\, \langle y\rangle^\frown ys &= \{\langle x\rangle^\frown u \mid u \in xs \,|||\, \langle y\rangle^\frown ys\} \cup \{\langle y\rangle^\frown u \mid u \in ys \,|||\, \langle x\rangle^\frown xs\}
\end{aligned}
$$

Using the above, we can now prove the required result as follows:

$$
\begin{aligned}
&\mathit{failures}^s(Q \,|||\, R) \\
&= \mathit{failures}^s((Q \,;\, \mathit{SKIP}) \,|||\, (R \,;\, \mathit{SKIP}))
\end{aligned}
$$

$$
\begin{aligned}
&= \{(tr, Y \cup Z) \mid Y \setminus \{\checkmark\} = Z \setminus \{\checkmark\} \\
&\qquad\qquad \wedge \exists\, s, t \cdot tr \in s \mathbin{|||} t \wedge (s, Y) \in \mathit{failures}^s(Q \mathbin{;} \mathit{SKIP}) \\
&\qquad\qquad\qquad \wedge (t, Z) \in \mathit{failures}^s(R \mathbin{;} \mathit{SKIP})\} \\
&= \{(tr, Y \cup Z) \mid Y \setminus \{\checkmark\} = Z \setminus \{\checkmark\} \\
&\qquad\qquad \wedge \exists\, s, t \cdot tr \in s \mathbin{|||} t \\
&\qquad\qquad\qquad \wedge (s, Y) \in \mathit{failures}^r((\mathit{Sig}(Q) \mathbin{;} BSkip) \setminus \{\tau_r\}) \\
&\qquad\qquad\qquad \wedge (t, Z) \in \mathit{failures}^r((\mathit{Sig}(R) \mathbin{;} BSkip) \setminus \{\tau_r\})\} && \langle \text{IH} \rangle \\
&= \mathit{failures}^r(((\mathit{Sig}(Q) \mathbin{;} BSkip) \setminus \{\tau_r\}) \mathbin{|||} ((\mathit{Sig}(R) \mathbin{;} BSkip) \setminus \{\tau_r\})) \\
&= \mathit{failures}^r(((\mathit{Sig}(Q) \mathbin{;} BSkip) \mathbin{|||} (\mathit{Sig}(R) \mathbin{;} BSkip)) \setminus \{\tau_r\}) \\
&= \mathit{failures}^r(((\mathit{Sig}(Q) \mathbin{;} BSkip) \underset{\{\tau_r\}}{\parallel} (\mathit{Sig}(R) \mathbin{;} BSkip)) \setminus \{\tau_r\}) && \langle \dagger \rangle \\
&= \mathit{failures}^r(\mathit{Sig}(Q \mathbin{|||} R) \setminus \{\tau_r\})
\end{aligned}
$$

The step marked † us justified by observing that forcing synchronisation on τ_r does not alter the failures. This is because, in any state in which the first equation could perform a τ_r, it must be because a $BSkip$ is doing so. However, note that it does not matter if a $BSkip$ is blocked from proceeding by a synchronisation on a ✓ or a τ_r; due to the hiding involved, in such a state both equations can refuse the whole of $\Sigma^{\checkmark,\tau_r}$, as required.

Hence, as this covers all possible cases for P, the required result follows. □

2.3. *Efficiency*

The solution sketched above is certainly efficient with regards to implementation, since it requires only a straightforward substitution to be performed on a process. However, unfortunately, the transformation would cause FDR to produce less efficient representations of the various LTSs, thus slowing down the refinement checking process and causing more memory to be consumed. In order to explain the issue, we firstly briefly review how FDR internally represents labelled transition systems (LTSs).

FDR has two main internal representations of LTSs. The simplest, known as a *low-level* machine, is an explicit graph. FDR's most useful representation is known as a *supercombinator*. A supercombinator takes a number of other LTSs and a set of rules that say how the transitions of the component machines combine together to produce the transitions of the overall machine. For example, a supercombinator for $P \mathbin{|||} Q$, assuming P and Q are represented as explicit graphs, would have: two component machines, P and Q; a rule for each non-✓ event that P or Q can perform to promote the event; a rule that causes the machine to perform a ✓ when both P and Q do. Further, in order to support operators such as $P \mathbin{;} Q$, these rules are arranged into *formats*. For example, the supercombinator for $P \mathbin{;} Q$ has two *formats*. The first format represents the rules that are active before P has performed a ✓, whilst the second format is active after P has performed a ✓ (which will have been converted into a τ) and represents how the transitions of Q are promoted. Thus rules can also specify which format they transition into. Further, supercombinators are actually constructed recursively (this is the key to efficiency). For example, assuming that each P_i is a low-level machine, then a single supercombinator is constructed for $(P_1 \mathbin{;} P_2) \mathbin{|||} (P_3 \mathbin{;} P_4)$, rather than three different supercombinators. This is constructed by combining together the rules for the $;$ and $|||$ operators.

In our particular simulation it is this recursive construction that is the source of the inefficiency. For example, consider $Q = |||_{i:\{1..N\}}(P_i \,;\, \mathit{SKIP})$: assuming that each of the P_i is represented as a low-level machine, this machine will have 2^N formats since the supercombinator has to have a format that represents Q being in any possible combination of the formats of each $P_i ; \mathit{SKIP}$ process. Since it is not possible (or at least, certainly not desirable), to lazily construct formats, this could add significant time and memory consumption to the check.

Recall that the only difference in the denotational semantics between the ✓-as-Signal and ✓-as-Refusable semantics is that the former adds extra failures. In particular, according to Equation 1.1, $\mathit{failures}^s(P)$ contains the extra failures:

$$\{(tr, X) \mid P \xRightarrow{tr^\frown\langle\checkmark\rangle} \Omega, X \subseteq \Sigma\}$$

Note that any process that just offers a ✓ will already have all of the above failures. Therefore, the only processes that the above will add failures to are those that contain a choice between a visible event and a ✓. For example, $\mathit{SKIPChoice}_a$ and $\mathit{SKIP} \;|||\; \mathit{SKIPChoice}_a$ both contain choices between visible events and ✓'s, and thus have extra failures added. However, as processes such as $P \;\Box\; \mathit{SKIP}$ are sufficiently unusual[5], if we only apply the transformation to the relevant portions of a process the increase in cost will be negligible. We formalise this as follows.

Definition 2.6. A process P is *operationally discretionary* iff $P \xrightarrow{\checkmark} \Omega$ and there exists $a \in \Sigma$ such that $P \xrightarrow{a} P'$.

Theorem 2.7. Let P be a process such that for all P' such that $P \xRightarrow{tr} P'$, P' is not operationally discretionary. Then $\mathit{failures}^s(P) = \mathit{failures}^r(P)$.

Proof (Sketch). This can be proven by induction over the process, noting the above observation regarding the extra failures in $\mathit{failures}^s(P)$ as opposed to $\mathit{failures}^r(P)$. □

Checking if every state has a ✓ alongside a visible event available is clearly not feasible due to the time required to check such a property. Therefore, we need to compute a sound over-approximation of the operationally-discretionary property. We can do so by noting that an Independent operator only yields an operationally discretionary process if it has an argument that is SKIP and another argument that is a visible event. Further, a synchronising operator only yields an operationally discretionary process if one of its arguments is operationally discretionary, and the remaining arguments can terminate.

Whilst the above works for a large class of operators, incorporating hiding requires a little more complexity. In particular, consider the processes $P \mathrel{\widehat{=}} a \to \mathit{SKIP}$ and $P \setminus \{Y\} \;\Box\; b \to \mathit{STOP}$. Note that this process has an operationally-discretionary argument iff $a \in Y$. Hence, to detect the above, we also need to track the set of events that are hidden and thus could result in a SKIP being reached by only τ events. We formalise this in the following definition.

Definition 2.8. A process P has a *X-guarded SKIP* iff: $P \xRightarrow{s^\frown\langle\checkmark\rangle} \Omega$ for $s \in X^*$ iff P is one of the following forms:

1. $P = \mathit{SKIP}$;
2. $P = a \to Q$, $a \in X$ and Q has an X-guarded SKIP;
3. P is either an Independent operator, is a non-deterministic choice, or is a sequential composition, and P has at least one argument that has an X-guarded SKIP;
4. P is a synchronising operator such that all of its arguments have X-guarded SKIP's;

[5] As Roscoe notes in [2], Hoare actually banned this process because it appears so unnatural. Thus, whilst our simulation is less efficient on such processes, this is arguably a reasonable price for perversity!

5. $P = Q[\![R]\!]$ and Q has a Y-guarded $SKIP$ where Y is the pre-image of X under R;
6. $P = Q \setminus Y$ and Q has an $(X \cup Y)$-guarded $SKIP$.

A process P has an *unguarded* $SKIP$ iff it has a $\{\}$-guarded $SKIP$. A process P is *problematic* iff either:

1. P is an Independent operator, one argument of P has an unguarded $SKIP$, and there exists an **on** argument P' of P such that $P' \overset{\langle a \rangle}{\Longrightarrow}$ for some $a \in \Sigma$; or
2. P has a problematic process argument.

For example, consider the process $P = a \to SKIP \;\Box\; SKIP$. By the above definition, this is rightly considered problematic since P is an independent operator and has an argument with an unguarded $SKIP$. However, $P \setminus \{a\} = SKIP$ and is thus intuitively unproblematic. It is also formally unproblematic since $a \to SKIP \;\Box\; SKIP$ has a $\{a\}$-guarded $SKIP$.

Note that the above is indeed an over-approximation. For example, consider the process $P = a \to SKIP \;\Box\; b \to SKIP$. Whilst P is unproblematic, $P \setminus \{a\}$ is identified as problematic, even though $P \setminus \{a\} = b \to SKIP \triangleright SKIP$, which is clearly unproblematic. We believe that such processes should be sufficiently uncommon to ensure that the over-approximation is a useful one.

We now prove that the above computes a sound approximation, as follows.

Theorem 2.9. If P is not problematic then P contains no operationally-discretionary subprocess.

Proof (Sketch). This can be proven by a structural induction over P, noting that in the definition of X-guarded, X is a sound approximation to the set of events that can be τ's. Hence, since a non-problematic process has no Independent operators that have **on** arguments that have unguarded $SKIP$'s and visible events, the required result immediately follows. □

3. Conclusions

In this paper we have discussed the difference between the ✓-as-Refusable and ✓-as-Signal semantics. In Section 2.1, we specified a way of simulating the ✓-as-Signal semantics within the ✓-as-Refusable semantics before proving, in Section 2.2, that the simulation is correct, in that it produces the same denotational value in the ✓-as-Refusable semantics as the ✓-as-Signal semantics. Lastly, in Section 2.3, we discuss how the simulation can be inefficient when considering how FDR internally represents LTSs and proposed a solution that reduces the overhead for all but the most unusual of processes.

We believe that the technique that we have presented in this paper provides a way of simulating the ✓-as-Signal semantics within FDR with a relatively minimal number of changes. We hope to implement the simulation in the context of FDR3, which will be released towards of the end of the year. This will be the first time that the FDR refinement checker has supported checking refinements under different termination semantics, which will hopefully be of wider interest to those who are interested in studying termination in a more theoretical context.

Acknowledgements

We would like to thank Bill Roscoe for several interesting discussions on this work. We would also like to thank the anonymous reviewers for providing many useful comments and interesting discussions.

References

[1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

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

[3] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[4] H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337. 1997.

[5] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement—FDR 2 User Manual*, 2011.

[6] J. Davies. *Specification and Proof in Real Time CSP*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1993.