Compositional Verification of a Network of CSP Processes: using FDR2 to verify refinement in the event of interface difference

Jonathan Burton

Department of Computing Science, University of Newcastle,
Newcastle upon Tyne NE1 7RU, U.K.
j.i.burton@ncl.ac.uk

Abstract. The paper [5] presented an implementation relation formalising what it means for one process to "implement" another in the CSP (Communicating Sequential Processes, [15]) framework in the event that the two processes have different interfaces. An improved version of the relation appears in [6] and allows for *compositional verification* of a network of CSP processes.
The model checker FDR2([15]) may be used to check CSP refinement in the event that specification and implementation processes have the *same* interface. In this paper, we show how to transform the problem of checking the conditions from [6], where the specification and implementation processes have *different* interfaces, into one where the specification and implementation processes have the *same* interface. This allows us to take advantage of the existing tool FDR2 and allows automatic, compositional verification using the relation developed.

Keywords: Compositional verification, communicating sequential processes, behaviour abstraction.

1 Introduction

We first give some motivating detail behind the implementation relation scheme presented in [6] and reproduced in section 3 before going on to describe the results presented in this paper.

Consider the situation that we have a specification network, P_{Net}, composed of n processes P_i, where all interprocess communication is hidden. Consider an implementation network, Q_{Net}, also composed of n processes, again with all interprocess communication hidden. Assume that there is a one-to-one correspondence between component processes in P_{Net} and those in Q_{Net}. Intuitively, P_i is intended to specify Q_i. Finally, assume that the interface of Q_{Net}, in terms of externally observable actions, is the same as that of P_{Net}.

In process algebras, such as those used in [12, 15], the notion that a process Q_{Net} *implements* a process P_{Net} is based on the idea that Q_{Net} is more deterministic than (or equivalent to) P_{Net} in terms of the chosen semantics. We then say that (the behaviour of) Q_{Net} *refines* (the behaviour of) P_{Net}. Moreover, the interface of Q_{Net} (the implementation process) must be the same as that of P_{Net} (the specification process) to facilitate comparison.

What if we wish to approach this verification question compositionally? What if we want to verify that Q_{Net} implements P_{Net} simply by verifying that Q_i implements P_i, for each $1 \leq i \leq n$. In general, this is only possible if Q_i and P_i have the same communication interface. Thus, Q_i may implement P_i by describing its internal (and so hidden) computation in a more concrete manner, but it may not do so by refining its external interface, at least if we wish to carry out compositional verification.

Yet in deriving an implementation from a specification we will often wish to implement abstract, high-level *interface* actions at a lower level of detail and in a more concrete manner. For example, the channel connecting P_i to another component process P_j may be unreliable and so it may need to be replaced by a data channel and an acknowledgement channel. Or P_i itself may be liable to fail and so its behaviour may need to be replicated, with each new component having its own communication channels to avoid a single channel becoming a bottleneck. Or it may simply be the case that a high-level action of P_i has been rendered in a more concrete, and so more implementable, form. As a result, the interface of an implementation process may end up being expressed at a lower (and so different) level of abstraction to that of the corresponding specification process. In the process algebraic context, where our interest lies only in *observable* behaviour, this means that verification of correctness must be able to deal with the case that the implementation and specification processes have different interfaces.

The relation between processes detailed in [6] and given in section 3 enjoys two crucial properties that allow us to carry out compositional verification in the event that Q_i and P_i have different interfaces. The first is that of *compositionality*, meaning that the implementation relation distributes over system composition. Thus, a specification composed of a number of connected systems may be implemented by connecting their respective implementations and so we may verify compositionally that Q_{Net} implements P_{Net} *according to our implementation relation scheme* simply by verifying that each Q_i implements P_i.

Moreover, although in general Q_i and P_i do not have the same interface, we know that, when all of the components Q_i have been composed, the result — namely Q_{Net} — will have the same interface as the corresponding specification process — namely P_{Net}. By the compositionality property, we know that Q_{Net} implements P_{Net} according to the scheme presented here. We then impose the requirement that the implementation relation presented here reduces to a standard notion of behaviour refinement in the CSP framework in the event that the implementation and specification processes have the *same* interface. This allows us to verify compositionally that Q_{Net} implements P_{Net}.

In order to apply the implementation relation described above to non-trivial examples, it is necessary that we have some means for its automatic verification. The approach taken in this paper is to transform the verification question presented here into one which may be answered by an existing, industrial strength tool. In a sense, we create a new "tool", but one which inherits a level of robustness and maturity of development that only comes after a number of years of use and modification.

The tool which is used here is FDR2 [13–15] developed for the purposes of model-checking CSP. As implied above, FDR2 takes as input two processes, one a specification and the other an implementation, which must have the same interface. The implementation relation described in section 3 takes as input a specification process and an implementation process, with possibly different interfaces, along with a means of interpretation — called *extraction patterns* and described in section 3 — which acts as a parameter to the relation. We show here how to transform this set of inputs into a set of inputs acceptable to FDR2, using only the syntatic operators of the CSP language, such that a set of refinement checks within FDR2 are successful if and only if the conditions from the implementation relation are met. This then gives automatic, compositional verification of a network of CSP processes.

It turns out there are a number of different ways in which this problem may be approached, the most fundamental distinction perhaps being whether we choose to work at the level of abstraction of the implementation process or at that of the specification process. For we must transform syntactically one of the processes so that it has the same interface as the other (it is this transformation which allows us to use FDR2). It seems that different approaches may work better in different contexts.

For example, working at the level of abstraction of the specification means that we can make the inputs supplied to FDR2 as small as possible, which allows us to verify larger processes. However, this can cause problems when we approach debugging, since all counter-examples are generated at this higher level of abstraction and it is more difficult to relate them to the behaviours of the implementation.

In the future, it is likely that different approaches will be implemented and a choice will be made between them by the user as appropriate. For the momement, however, in the absence of the necessary experimental data, the rationale followed in this report has been to make the inputs to FDR2 as small as possible: this is because reducing the size of processes seems unlikely to lead to significant, if any, slowdown, while allowing us to work with a greater range of, possibly quite large, examples.

The remainder of the paper is organised as follows. In the next section we present some preliminary definitions. In section 3 we describe the device of extraction patterns, introduce the implementation relation and give the results which allow us to use it for the purposes of compositional verification.

Section 3 also introduces a running example which will be used to illustrate how the verification method works in practice. Section 4 describes the preprocessing which must be carried out on an implementation process before verification can begin and the subsequent sections detail the manner in which the individual conditions of the implementation relation are checked. Finally, we give some concluding remarks in section 10.

2 Preliminaries

In this section, we first recall those parts of the CSP theory which are needed throughout the paper.

2.1 Actions and traces

Communicating Sequential Processes (CSP) [2, 3, 10, 15] is a formal model for the description of concurrent computing systems. A CSP *process* can be regarded as a black box which may engage in interaction with its environment. Atomic instances of this interaction are called *actions* and must be elements of the *alphabet* of the process. The alphabet of a process P is denoted αP. A *trace* of the process is a finite sequence of actions that a process can be observed to engage in. In this report, structured actions of the form $b.v$ will be used, where v is a *message* and b is a communication *channel.* $b.v$ is said to *occur* at b and to cause v be *exchanged* between processes communicating over b. For every channel b, μb is the *message set* of b - the set of all v such that $b.v$ is a valid action. We define $\alpha b = \{b.v \mid v \in \mu b\}$ to be the *alphabet* of channel b. It is assumed that μb is always finite and non-empty. For a set of channels B, $\alpha B = \bigcup_{b \in B} \alpha b$.

The following notation is similar to that of [10] (below $t, u, t_1, t_2, \ldots$ are traces; b, b', b'' are channels; $B_1, \ldots, B_n$, B are disjoint sets of channels; a is an action; A is a set of actions; and T, T' are non-empty sets of traces):

- $t = \langle a_1, \ldots, a_n \rangle$ is the trace whose i-th element is a_i, and whose length, $|t|$, is n.
- $t \circ u$ is the trace obtained by appending u to t.
- A^* is the set of all traces of actions from A, including the empty trace, $\langle\,\rangle$.
- A^ω is the set of all *infinite* traces of actions from A.
- T^* is the set of all traces $t = t_1 \circ \cdots \circ t_n$ $(n \geq 0)$ such that $t_1, \ldots, t_n \in T$.
- $\leq$ denotes the prefix relation on traces, and $t < u$ if $t \leq u$ and $t \neq u$.
- $\mathit{Pref}(T) = \{u \mid \exists t \in T : \ u \leq t\}$ if the *prefix-closure* of T; T is *prefix-closed* if $T = \mathit{Pref}(T)$.
- $t[b'/b]$ is a trace obtained from t by replacing each action $b.v$ by $b'.v$.
- $t \upharpoonright B$ is obtained by deleting from t all the actions that do not occur on the channels in B; for example, $\langle b''.3, b.1, b''.2, b.3, b'.3, b''.6, b'.2\rangle \upharpoonright \{b, b'\} = \langle b.1, b.3, b'.3, b'.2\rangle$.
- $t \upharpoonright A$ is obtained by deleting from t all the actions that do not appear in A.
- An infinite sequence $t_1, t_2, \ldots$ is an ω*-sequence* if $t_1 \leq t_2 \leq \ldots$ and $\lim_{i \to \infty} |t_i| = \infty$.
- A mapping $f : T \to T'$ is *monotonic* if $t, u \in T$ and $t \leq u$ implies $f(t) \leq f(u)$; f is *strict* if $\langle\,\rangle \in T$ and $f(\langle\,\rangle) = \langle\,\rangle$; and f is a *homomorphism* if $t, u, t \circ u \in T$ implies $f(t \circ u) = f(t) \circ f(u)$.
- A family of sets $\mathcal{X}$ is *subset-closed* if $Y \subset X \in \mathcal{X}$ implies $Y \in \mathcal{X}$.

2.2 CSP operators

We now give a brief description of the CSP operators which we shall use. We use deterministic choice, $P \,\Box\, Q$, non-deterministic choice $P \sqcap Q$, and prefixing, $a \to P$. Parallel composition $P \| Q$ models synchronous communication between processes in such a way that each of them is free to engage independently in any action that is not in the other's alphabet, but they have to engage simultaneously in all actions that are in the intersection of their alphabet. The operators $\Box$, $\sqcap$ and $\|$ are all commutative and associative and may be indexed over finite sets.

Let P be a process and A be a set of events; then $P\backslash A$ is a process that behaves like P with the actions from A made invisible. (Note that the operator $\backslash$ is overloaded since it is also used to denote set subtraction.) Hiding is associative in that $(P\backslash A)\backslash A' = P\backslash(A \cup A')$. If t is a trace of a process P, then $t \setminus A = t \upharpoonright (\alpha P \setminus A)$.

Let R be a relation and P a process. Then $P[R]$ is a process that behaves like P except that every action a has been replaced by $R(a)$; note that the relation R does not need to be total over the alphabet of P and if a is not in the domain of R, we assume that $R(a) = a$. $R(a)$ will return a set of values and so wherever the action a might have been enabled in P, *each* of the events in $R(a)$ will be enabled in its place in $P[R]$. The definition of R extends to traces and sets in the obvious way. We define R^* as follows: $\langle a_1, \ldots, a_n\rangle R^* \langle b_i, \ldots, b_m\rangle \Leftrightarrow n = m \wedge \forall i \leq n, a_i R b_i$. And $R(A) = \bigcup_{a \in A} R(a)$. The semantic definitions of the above operators may be found in section A in the appendix.

We now give details of the process models we will use. In what follows, let P and Q be CSP process terms: that is, syntactic definitions of CSP processes. (Note that the word process may denote either a syntactic term or a semantic object. In what follows, the description *process* will be suitably qualified where the context requires it.)

The simplest model of behaviour in CSP is the traces model, where a process term P is modelled by a pair $(\alpha P, \tau P)$. τP or the traces of P is the set of *finite* sequences of *visible* actions which that process may execute. In the stable failures model, a process term P is modelled by a triple $(\alpha P, \tau P, \phi P)$, where ϕP — the *stable* failures of P — is a subset of $\alpha P^* \times 2^{\alpha P}$. If $(t, R) \in \phi P$ then P is able to *refuse* R after t. Intuitively, this means that if the environment only offers R as a set of possible events to be executed after t, then P can deadlock when placed in parallel with the environment. In the failures divergences model, a process term P is modelled as a triple, $(\alpha P, \phi_\perp P, \delta P)$, where δP — *divergences* — is a subset of αP^* and $\phi_\perp P$ — *failures* — is a subset of $\alpha P^* \times 2^{\alpha P}$. If $t \in \delta P$ then P is said to *diverge* after t. In the CSP model this means the process behaves in a totally uncontrollable way. Such a semantical treatment is based on what is often referred to as 'catastrophic' divergence whereby the process in a diverging state is modelled as being able to accept any trace and generate any refusal. When verifying a given property, we shall always use the simplest model in which it is expressible, since this simplifies proofs and the tool FDR2 is more efficient when used with simpler models [15].

In the rest of this paper, we deal with specification and implementation processes, along with processes defined to facilitate refinement checking using FDR2. If W is either a specification or implementation process, we associate with W a set of communication channels and denote this set χW. χW may be partitioned into input channels ($\mathit{in}\, W$) and output channels ($\mathit{out}\, W$). We identify the alphabet of such a process W with the alphabet of χW and stipulate that $\alpha W = \bigcup_{b \in \chi W} \alpha b$. As a result, the alphabet of such a process W is taken as a given, while the alphabet of any other process may be derived compositionally according to the rules in section A in the appendix. These two ways of deriving the alphabet of a process are not inconsistent: process alphabets are only used to calculate the semantics of processes defined using the parallel composition operator, $\|$, and, in FDR2, we can effectively give a process any alphabet we want when using the parallel composition operator (see below).

Processes $W_1, \ldots, W_n$ form a *network* if no channel is shared by more than two W_i's. We define $W_1 \otimes \cdots \otimes W_n$ to be the process obtained by taking the parallel composition of the processes and then hiding all interprocess communication, i.e., the process $(W_1 \| \cdots \| W_n)\backslash \alpha B$, where B is the set of channels shared by at least two different processes W_i. Note that for $i \neq j$, if $c \in (\chi W_i \cap \chi W_j)$, then $c \in \mathit{in}\, W_i$ and $c \in \mathit{out}\, W_j$, or $c \in \mathit{out}\, W_i$ and $c \in \mathit{in}\, W_j$.

In CSP, that a process Q implements a process P is denoted by $P \sqsubseteq_X Q$, where X denotes the model in which we are working. (T denotes the traces model, F the stable failures model and FD the failures divergences model.) $P \sqsubseteq_T Q$ if and only if $\tau Q \subseteq \tau P$. $P \sqsubseteq_F Q$ if and only if $\tau Q \subseteq \tau P$ and $\phi Q \subseteq \phi P$. $P \sqsubseteq_{FD} Q$ if and only if $\delta Q \subseteq \delta P$ and $\phi_\perp Q \subseteq \phi_\perp P$. Note that the alphabet component plays no role here.

It is the case that, for any process term P, $\phi_\perp P = \phi P \cup \{(t, R) \mid t \in \delta P \ \wedge \ R \subseteq \alpha P\}$. In addition, we define $\tau_\perp P \triangleq \{t \mid (t, R) \in \phi_\perp P\}$ and observe that $\tau_\perp P = \tau P \cup \delta P$, where τP is the meaning of

P in the traces model. Finally, we define $min\delta(P) \triangleq \{t \mid t \in \delta P \ \wedge \ (\forall u < t)\ u \notin \delta P\}$ and have that $\tau P = min\delta(P) \cup \{t \mid (t, R) \in \phi P\}$ [15].

$[[P]]_X$ denotes the semantic meaning of P in the model $X \in \{T, F, FD\}$, while $P =_X Q$ if and only if $[[P]]_X = [[Q]]_X$. From the above definitions of the failures divergences and stable failures models respectively, it is easy to see that if $\delta P = \emptyset$ then $\phi P = \phi_\bot P$ and $\tau_\bot P = \tau P = \{t \mid (t, R) \in \phi P)\}$. We use DIV to denote the immediately diverging process and observe that $[[DIV]]_F = (\{\langle\rangle\}, \emptyset)$ (note that we have omitted the alphabet component here) [15]. In FDR2, deadlock freedom is checked in the stable failures model. A process P is *deadlock-free* if and only if, for every $(t, R) \in \phi P$, R is a proper subset of αP.

All inputs to FDR2 must be supplied in the machine-readable syntax of CSP. (See [15] or the FDR2 manual ([7]) for details.) Although we will define processes syntactically in this paper using the standard CSP syntax, all of the operators used have a direct equivalent in the machine-readable syntax. Note, however, that the operator in the machine-readable syntax which represents parallel composition must be provided explicitly with the events on which synchronization is to occur: to mimic the parallel composition operator used here it is simply necessary to provide the set of events in the intersection of the alphabets of the two processes to be composed.

3 Extraction patterns and the implementation relation

3.1 Extraction patterns

In this section, we first explain the basic mechanism behind our modelling of behaviour abstraction — extraction patterns — and then provide a formal definition of these objects. The example given here, which is used as a running example in the remainder of the paper, is deliberately *very simple*, in order to better convey the basic ideas, rather than to demonstrate a wider applicability of the approach.

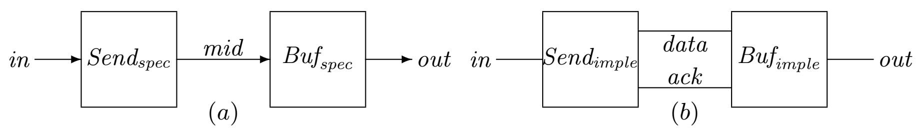

Fig. 1. A simple specification network and its implementation

Consider a pair of specification or *base* processes, $Send_{spec}$ and Buf_{spec}, shown in figure 1(a). $Send_{spec}$ generates an infinite sequence of 0s or an infinite sequence of 1s, depending on the signal (0 or 1) received on its input channel, *in*, at the very beginning of its execution. Buf_{spec} is a buffer process of capacity one, forwarding signals received on its input channel, *mid*. In terms of CSP, we have:

$$Send_{spec} \triangleq \Box_{i\in\{0,1\}}\ in.i \to S_i \qquad \text{and} \qquad Buf_{spec} \triangleq \Box_{i\in\{0,1\}}\ mid.i \to B_i$$

where $S_i \triangleq mid.i \to S_i$ and $B_i \triangleq out.i \to Buf_{spec}$, for $i = 0, 1$.

Suppose that the communication on *mid* is liable to fail and has therefore been implemented using two channels, *data* and *ack*, where *data* is a data channel, and *ack* is a feedback channel used to pass acknowledgements. It is assumed that a given message is sent at most *twice* since a re-transmission always succeeds. This leads to a simple protocol which can be incorporated into suitably modified original processes. The resulting implementation processes shown in figure 1(b), $Send_{imple}$ and Buf_{imple}, are given by:

$$Send_{imple} \triangleq \Box_{i\in\{0,1\}}\ in.i \to S_i$$

$$Buf_{imple} \triangleq \Box_{i\in\{0,1\}}\ data.i \to (ack.yes \to B'_i \sqcap ack.no \to B)$$

where B, S_i and B'_i $(i = 0, 1)$ are auxiliary processes defined thus:

$$S_i \triangleq data.i \rightarrow (ack.yes \rightarrow S_i \;\square\; ack.no \rightarrow data.i \rightarrow S_i)$$
$$B \triangleq \square_{i \in \{0,1\}}\, data.i \rightarrow B'_i$$
$$B'_i \triangleq out.i \rightarrow Buf_{imple}$$

Suppose we wish to show that Buf_{imple} is a valid implementation of Buf_{spec}. We need some way of relating behaviour over channels *data* and *ack* to that over *mid*. Firstly, we must relate traces and do this using a mapping *extr*, which in this case, for example, would need to relate traces over *data* and *ack* to those over *mid*. For example,

$$\begin{aligned}
\langle\,\rangle &\mapsto \langle\,\rangle \\
\langle data.v\rangle &\mapsto \langle\,\rangle \\
\langle data.v, ack.yes\rangle &\mapsto \langle mid.v\rangle \\
\langle data.v, ack.no\rangle &\mapsto \langle\,\rangle \\
\langle data.v, ack.no, data.v\rangle &\mapsto \langle mid.v\rangle
\end{aligned}$$

Note that the mapping *extr* has a well-defined domain here: it is only defined for traces where message transmission was successful in the first place or where retransmission *must* succeed We call this domain *Dom*. Observe also that some of the traces in *Dom* may be regarded as *incomplete*. For example, $\langle data.v\rangle$ is such a trace since we know that the event *data.v* must be followed by an event on channel *ack*. The set of all other traces in *Dom* — i.e. those which in principle may be *complete* — will be denoted by *dom*.[1] For our example, *dom* will contain all traces in *Dom* where we know that the last value communicated on channel *data* has been transmitted (or retransmitted) successfully.

We also need a device to relate the refusals (and so failures) of Buf_{imple} to those of Buf_{spec}. This comes in the form of another mapping, *ref*, constraining the possible refusals a process can exhibit, after a given trace $t \in Dom$[2], on channels which will be hidden when the final implementation network is composed. More precisely, a sender process can admit a refusal disallowed by $ref(t)$ only if the extracted trace $extr(t)$ admits in the specification process the refusal of all communication on the corresponding channel and, moreover, the trace t itself is complete, i.e., $t \in dom$. For example, in the process described above, if $Send_{imple}$ has just communicated *data.v* then its behaviour cannot be complete and so it cannot refuse anything on the channel *ack*: i.e. it must be ready to receive either a positive or a negative acknowledgement. And if behaviour is complete — i.e. $t \in dom$ — in Buf_{imple}, it can, for example, refuse an event on channel *data* only if Buf_{spec} can refuse everything on channel *mid* after performing $extr(t)$.

The refusal bounds given by *ref* may be thought of as ensuring a kind of liveness or progress condition on sets of channels upon which composition will occur when implementation components are composed to build a full implementation system Q_{Net} (introduced in section 1). Since these channels are to be composed upon and so hidden, the progress enforced manifests itself in the final system as the occurrence of an invisible transition and states in which an invisible transition is enabled do not contribute a (stable) failure of Q_{Net}. Conversely, if we may not enforce progress after a *complete* behaviour, then it is possible that the relevant state reached *will* contribute to a failure, (t, R), of Q_{Net}. Since, in the stable failures model of CSP[3], if Q_{Net} 'implements' P_{Net} then $\phi Q_{Net} \subseteq \phi P_{Net}$, we must ensure that the relevant failure, (t, R), also occurs in P_{Net}. We do this by ensuring that progress will *not* be possible on the

[1] In general, $Dom = Pref(dom)$, meaning that each interpretable trace has, at least in theory, a chance of being completed.

[2] In general, we will only be interested in traces belonging to *Dom*.

[3] We are able to work only in the stable failures model since we assume the divergence freeness of any specification component and also, with a slight qualification, require the divergence freeness of the implementation component.

corresponding channel in the specification component. Here, lack of progress on internal channels leads to the fact that the relevant state *will* give rise to a failure of P_{Net}.

Finally, it should be stressed that $ref(t)$ gives a refusal bound on the sender side (more precisely, the process which implements the sender specification process). But this is enough since, if we want to rule out a deadlock in communication between the sender and receiver (on a localised set of channels), it is now possible to stipulate on the receiver side that no refusal is such that, when combined with any refusal allowed by $ref(t)$ on the sender side, it can yield the whole alphabet of the channels used for transmission.

3.2 Formal definition

The notion of extraction pattern relates behaviour on a set of channels in an implementation process to that on a channel in a specification process. An *extraction pattern* is a tuple $ep = (B, b, dom, extr, ref)$ satisfying the following:

EP1 B is a non-empty set of channels, called *sources* and b is a channel, called *target*.
EP2 dom is a non-empty set of traces over the sources; its prefix-closure is denoted by Dom.
EP3 $extr$ is a strict, monotonic mapping defined for traces in Dom; for every t, $extr(t)$ is a trace over the target.[4]
EP4 ref is a mapping defined for traces in Dom such that, for every $t \in Dom$, $ref(t)$ is a non-empty subset-closed family of proper subsets of αB and, if $a \in \alpha B$ and $t \circ \langle a \rangle \notin Dom$, then $R \cup \{a\} \in ref(t)$, for all $R \in ref(t)$.

As already mentioned, the mapping $extr$ interprets a trace over the source channels B (in the implementation process) in terms of a trace over a channel b (in the base or specification process) and defines functionally correct (i.e., in terms of traces) behaviour over those source channels by way of its domain Dom. The mapping ref is used to define correct behaviour in terms of failures as it gives bounds on refusals after execution of a particular trace sequence over the source channels. dom contains those traces in Dom for which the communication over B may be regarded as complete; the constraint on refusals given by ref is only allowed to be violated for such traces. The intuition behind this requirement is that we cannot regard as correct a situation where deadlock occurs in the implementation process when behaviour is incomplete, since, if regarded as correct behaviour, this would imply that the specification process could in some sense deadlock while in the middle of executing a single (atomic) action.

The extraction mapping is monotonic as receiving more information cannot decrease the current knowledge about the transmission. $\alpha B \notin ref(t)$ will be useful in that for an unfinished communication t we do not allow the sender to refuse all possible transmission. The second condition in EP4 is a rendering in terms of extraction patterns of a condition imposed on CSP processes that impossible events can always be refused (see SF4 and FD3 in section A in the appendix).

We note that not all channels require all components of an extraction pattern to interpret their behaviour. We shall denote these channels *uninterpreted* or *identity* channels and their extraction patterns *identity extraction patterns*. Intuitively, these channels have the same interface in both implementation and specification. In particular, all channels which remain visible in the final compositions of the implementation and specification components respectively — for example Q_{Net} and P_{Net} — are of this type. For an extraction pattern ep, let $b = c$ be an uninterpreted channel. Then $B = \{c\}$ and $Dom = dom \triangleq (\alpha c)^*$. If $t \in Dom$, then $extr(t) = t$. That is, the extraction mapping for such channels is the identity mapping: i.e. behaviour over them does not require further interpretation. Finally, no ref component is specified for such an extraction pattern.

[4] For the purposes of this report, we assume that $|extr(t \circ \langle a \rangle)| \leq |extr(t)| + 1$.

Twice extraction pattern For the example given here, in order to demonstrate that $Send_{imple}$ and Buf_{imple} are implementations of respectively $Send_{spec}$ and Buf_{spec}, we will need an extraction pattern ep_{twice}. We also observe that the channels *in* and *out* are uninterpreted.

For the ep_{twice} extraction pattern, $B = \{data, ack\}$ are the source channels and $b = mid$ is the target channel; moreover $\mu mid = \mu data = \{0, 1\}$ and $\mu ack = \{yes, no\}$. The remaining components of ep_{twice} are defined in the following way, where $t \in dom$ and $t \circ u \in Dom$:

$$dom \triangleq \{\langle data.0, ack.yes\rangle, \langle data.0, ack.no, data.0\rangle, \langle data.1, ack.yes\rangle, \langle data.1, ack.no, data.1\rangle\}^*$$

$$extr(t \circ u) \triangleq \begin{cases} \langle\,\rangle & \text{if } t \circ u = \langle\,\rangle \\ extr(t) \circ \langle mid.v\rangle & \text{if } u = \langle data.v, ack.yes\rangle \\ & \text{or } u = \langle data.v, ack.no, data.v\rangle \\ extr(t) & \text{if } u = \langle data.v\rangle \text{ or } u = \langle data.v, ack.no\rangle \end{cases}$$

$$ref(t \circ u) \triangleq \begin{cases} 2^{\alpha data} & \text{if } u = \langle data.v\rangle \\ \{R \in 2^{\alpha data \cup \alpha ack} \mid \alpha data \not\subseteq R\} & \text{if } u = \langle\rangle \\ \{R \in 2^{\alpha data \cup \alpha ack} \mid data.v \notin R\} & \text{if } u = \langle data.v, ack.no\rangle\ . \end{cases}$$

Here, intuitively, we can extract $\langle mid.0\rangle$ from two sequences of communications: $\langle data.0, ack.yes\rangle$ and $\langle data.0, ack.no, data.0\rangle$ (and similarly for $\langle mid.1\rangle$). A valid trace in Dom is one which is a concatenation of a series of 'complete' segments of this kind, possibly followed by an initial fragment of one of them. Any trace for which the latter is true is *incomplete* and belongs to $Dom \setminus dom$; otherwise it belongs to dom.

3.3 Additional notations

The various components of the extraction patterns can be annotated (e.g. subscripted) to avoid ambiguity. In what follows, different extraction patterns will have disjoint sources and distinct targets.

For notational convenience, we lift some of the notions introduced to finite sets of extraction patterns. Let $ep = \{ep_1, \ldots, ep_n\}$ be a non-empty set of extraction patterns, where $ep_i = (B_i, b_i, dom_i, extr_i, ref_i)$. Moreover, let $B = B_1 \cup \ldots \cup B_n$. Then:

EP5 $dom_{ep} = \{t \in \alpha B^* \mid \forall i \leq n : t \upharpoonright B_i \in dom_i\}$.
EP6 $Dom_{ep} = \{t \in \alpha B^* \mid \forall i \leq n : t \upharpoonright B_i \in Dom_i\}$.
EP7 $extr_{ep}(\langle\,\rangle) = \langle\,\rangle$ and, for every $t \circ \langle a\rangle \in Dom_{ep}$ with $a \in \alpha B_i$, $extr_{ep}(t \circ \langle a\rangle) = extr_{ep}(t) \circ u$ where u is a (possibly empty) trace such that $extr_i(t \upharpoonright B_i \circ \langle a\rangle) = extr_i(t \upharpoonright B_i) \circ u$.

Note that u in EP7 is well defined since $extr_i$ is monotonic.

3.4 The implementation relation

We present here the relation from [6], which is a slightly modified version of that in [5]. All proofs of results in the remainder of this section can be found in [6].

Suppose that we intend to implement a base process P using another process Q with a possibly different communication interface (in what follows, P and Q denote process expressions). The correctness of the implementation Q will be expressed in terms of two sets of extraction patterns, ep and ep'. The former (with sources $in\,Q$ and targets $in\,P$) will be used to relate the communication on the input channels of P and Q, while the latter (with sources $out\,Q$ and targets $out\,P$) will serve a similar purpose for the output channels.

Let P be a base process as in figure 2, and $ep_i = (B_i, b_i, dom_i, extr_i, ref_i)$ be an extraction pattern, for every $i \leq m + n$. We assume that the B_i's are mutually disjoint channel sets, and denote $ep = \{ep_1, \ldots, ep_m\}$ and $ep' = \{ep_{m+1}, \ldots, ep_{m+n}\}$. We then take a process Q such that $in\,Q = B_1 \cup \ldots \cup B_m$

Fig. 2. Base process P and its implementation Q.

and $out\,Q = B_{m+1} \cup \ldots \cup B_{m+n}$, as shown in figure 2, and denote by $\tau_{Dom}Q$ the set of all $t \in \tau_\perp Q$ which belong to $Dom_{ep\cup ep'}$. Similarly, $\phi_{Dom}Q$ and $\phi_{dom}Q$ will be the sets of those failures in $\phi_\perp Q$ in which the trace component belongs to $Dom_{ep\cup ep'}$ and $dom_{ep\cup ep'}$ respectively. Intuitively, $\tau_{Dom}Q$ — which is subsequently referred to as the *domain* of Q — is the set of $t \in \tau_\perp Q$ which are of actual interest and, consequently, $\phi_{Dom}Q$ is the set of $(t, R) \in \phi_\perp Q$ of actual interest too.

In what follows, we denote the uninterpreted channels of P as b_{id}. Note that these are also the uninterpreted channels of Q. The interpreted channels of P are denoted b_{nid} and the interpreted channels of Q are denoted B_{nid}. This means that $\alpha Q = \alpha B_{nid} \cup \alpha b_{id}$ and $\alpha P = \alpha b_{nid} \cup \alpha b_{id}$. We define also $I \triangleq \{i \mid b_i \in b_{nid}\}$ — intuitively, I means "interpreted" — and then $ep_I \triangleq \{ep_i \mid i \in I\}$. We then write $extr_I$ for $extr_{ep_I}$, Dom_I for Dom_{ep_I} and so on.

We will say that a channel b_i of P is *blocked* at a failure (t, R) in Q if either b_i is an input channel and $\alpha B_i - R \in ref_i(t \upharpoonright B_i)$, or b_i is an output channel and $\alpha B_i \cap R \notin ref_i(t \upharpoonright B_i)$ (blocking is not defined for any channel $c \in b_{id}$). Note that in both cases this signifies that the refusal bound imposed by the ref_i has been breached.

We then call Q an *implementation* of P w.r.t. extraction patterns ep and ep', denoted $Q \preceq^{ep}_{ep'} P$, if the following six conditions are satisfied.

- DP If $t \in \tau_\perp Q$ is such that $t \upharpoonright in\,Q \in Dom_{ep}$, then $t \in \tau_{Dom}Q$.
- DF $\tau_{Dom}Q \cap \delta Q = \emptyset$.
- TE $extr_{ep\cup ep'}(\tau_{Dom}Q) \subseteq \tau_\perp P$.
- GE If $\ldots, t_i, \ldots$ is an ω-sequence in $\tau_{Dom}Q$, then $\ldots, extr_{ep\cup ep'}(t_i), \ldots$ is also an ω-sequence.
- LC If $b_i \in b_{nid}$ and b_i is a channel of P blocked at $(t, R) \in \phi_{Dom}Q$, then $t \upharpoonright B_i \in dom_i$.
- RE If $(t, R) \in \phi_{dom}Q$ then $(extr_{ep\cup ep'}(t), \alpha B \cup (R \cap \alpha b_{id})) \in \phi_\perp P$, where $B \subseteq b_{nid}$ is the set of all channels of P blocked at (t, R).

We interpret the above conditions in the following way. DP expresses the *domain preservation* property, which says that if a trace of Q projected on the input channels can be interpreted by ep, then it must be possible to interpret the projection on the output channels by ep'. Note that such a condition is a simple *rely/guarantee* property in the sense of [8]. DF can be interpreted as *divergence freedom* within the domain of Q (recall that CSP divergences signify totally unacceptable behaviour). TE simply states that within the domain of Q we insist on generating P's *traces after extraction*. GE states that an unboundedly growing sequence of traces in the domain of Q is a sequence of traces unboundedly *growing after extraction* (notice that we place no restriction on the relative growth of the ω-sequences $\ldots, t_i, \ldots$ and $\ldots, extr_{ep\cup ep'}(t_i), \ldots$). LC means that going outside the bounds of allowed refusals indicates that the communication on a given (*interpreted*) channel may be considered as *locally completed*. Finally, RE states a condition for *refusal extraction*, which means that if a trace is locally completed on all channels, any local blocking of an (interpreted) channel of P in Q is transformed into the refusal of its whole alphabet in P; moreover, the refusals on the uninterpreted channels in Q should be matched in P.

A direct comparison of an implementation process Q with the corresponding base process P is only possible if there is no difference in the communication interfaces. This corresponds to the situation that all of the channels of Q (and of P) are uninterpreted. In such a case, we simply denote $Q \preceq P$ and then we can attempt to directly compare the semantics of the two processes in question.

Theorem 1. *If $Q \preceq P$ then $P \sqsubseteq_{FD} Q$ (i.e. $\delta Q \subseteq \delta P$ and $\phi_\perp Q \subseteq \phi_\perp P$).*
N.B. The latter implies that $\tau_\perp Q \subseteq \tau_\perp P$ as it suffices to take $R = \emptyset$.

The next result states that the implementation relation is compositional in the sense that it is preserved by the network composition operation. Taken with the previous result, we have met both of the restrictions stated in section 1 and so have a means of compositional verification in the event that corresponding specification and implementation component processes have different interfaces.

Theorem 2. *Let K and L be two base processes whose composition is non-diverging, as in figure 3, and let c, d, e, f, g and h be sets of extraction patterns whose targets are respectively the channel sets C, D, E, F, G and H. If $M \preceq^{c \cup h}_{d \cup e} K$ and $N \preceq^{d \cup f}_{g \cup h} L$ then $M \otimes N \preceq^{c \cup f}_{e \cup g} K \otimes L$.*

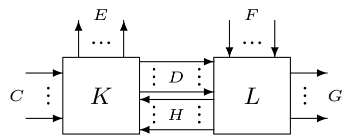

Fig. 3. Base processes used in the formulation of the compositionality theorem.

Hence the implementation relation is preserved through network composition and the only restriction is that the network of the base processes should be designed in a divergence-free way. This means, of course, that the base processes themselves must be divergence-free and so we assume $\delta P = \emptyset$ for the base process P described above. However, this is a standard requirement in the CSP approach (recall again that divergences are regarded as totally unacceptable).

3.5 Remainder of the report

Each of the subsequent sections deals with checking the conditions of the implementation relation given here, showing how this can be carried out using a standard notion of what it means for one process to implement another in CSP. As a result, this allows us to use FDR2 to verify the implementation relation. Each section defines various syntactic terms which are used either directly, or as components in larger terms, to define the inputs supplied to FDR2. Where appropriate, the example given in this section is used to provide a concrete example of how those syntactic terms would be used in practice: this gives both an illustration of the method at work and also allows us to verify automatically and compositionally that $(Send_{spec} \otimes Buf_{spec}) \sqsubseteq_{FD} (Send_{imple} \otimes Buf_{imple})$.

Although we will define such syntactic terms using the standard CSP syntax, all of the operators used have a direct equivalent in the machine-readable syntax. Also, as a final point, it is worth noting that the immediately diverging process does not have a distinguished syntactic representation in the way that, for example, the immediately deadlocking process does (i.e. $STOP$). As a result, we define the auxiliary process $X \triangleq a \rightarrow X$ and define the immediately diverging process $DIV \triangleq X \setminus \{a\}$.

4 Preprocessing the implementation process

Note that throughout this section we work within the full failures divergences model. Since none of the various components of the implementation relation we are to verify are interested in traces which, when projected on the input channels of Q, are not found in the set of valid traces over the input channels as defined by Dom_{ep}, it is necessary to remove them before proceeding with the verification proper. For example, imagine that $t \in \tau Q$ but $t \upharpoonright in\, Q \notin Dom_{ep}$. Condition DP places no restrictions on t and, in subsequent conditions, we take no notice of it since we only interest ourselves in traces in $\tau_{Dom} Q$. However, if we do not remove such traces from our implementation process, they will remain when we come to carry out refinement checks using FDR2 and may cause a particular refinement check to fail, despite the fact that the relevant condition has been met.

As a result, we remove these traces from Q, creating a process $\widehat{Q}$, such that $t \in \tau Q \wedge t \upharpoonright in\, Q \in Dom_{ep}$ if and only if $t \in \tau\widehat{Q}$ (we refer here to τQ rather than $\tau_\perp Q$ since we are not interested in traces which are only present due to the presence of divergence). Although this preprocessing affects the failures of Q, it does not do so in such a way that the answer to the verification question is different for Q than it is for $\widehat{Q}$ (see theorem 6).

In order to carry out this preprocessing on Q, we simply compose it in parallel with a component process for each of the extraction patterns which deal with an *interpreted* input channel (i.e. the extraction pattern is not an identity extraction pattern). Each of these processes is capable of performing exactly the traces from the *Dom* component associated with the relevant extraction pattern. Note that it is unnecessary to include identity extraction patterns in our preprocessing, since Dom_i for such an extraction pattern ep_i simply allows the execution of any trace over αb_i.

We first define a mapping $Next_i : Dom_i \to 2^{\alpha B_i}$ such that $Next_i(t) \triangleq \{a \mid t \circ \langle a \rangle \in Dom_i\}$. This mapping is used throughout the report and gives the possible extensions to a trace such that the resulting trace remains a member of the domain Dom_i. We also define the complement of $Next_i$ as $NotNext_i$, where $NotNext_i : Dom_i \to 2^{\alpha B_i}$ and $NotNext_i(t) \triangleq \{a \mid t \circ \langle a \rangle \notin Dom_i\}$. Let $\Gamma \triangleq \{i \mid ep_i \in ep \ \wedge\ b_i \in b_{nid}\}$. We then define $NotNext_\Gamma \triangleq \bigcup_{i \in \Gamma} NotNext_i$. We then have that $PP_i \triangleq PP_i(\langle\rangle)$ and[5]

$$PP_i(t) \triangleq \Box_{a \in Next_i(t)} a \to PP_i(t \circ \langle a \rangle)$$

where PP_i is a component process to be composed in parallel with Q during preprocessing. We finally define

$$PP_\Gamma \triangleq \ \|_{i \in \Gamma}\ PP_i \quad \text{and} \quad \widehat{Q} \triangleq Q \parallel PP_\Gamma$$

where $\widehat{Q}$ is Q after preprocessing has been applied.

Lemma 3 *The following hold of* $\widehat{Q}$*:*

1. $\alpha\widehat{Q} = \alpha Q$.
2. $\delta\widehat{Q} = \{t \circ u \mid t \in min\delta Q \ \wedge\ t \upharpoonright in\, Q \in Dom_{ep} \ \wedge\ u \in (\alpha\widehat{Q})^*\}$.
3. $\phi_\perp\widehat{Q} = \phi\widehat{Q} \ \cup\ \{(t, R) \mid t \in \delta\widehat{Q}\}$, *where* $\phi\widehat{Q} = \{(t, R) \mid (\exists (t, X) \in \phi Q, Y \subseteq NotNext_\Gamma(t))\ t \upharpoonright in\, Q \in Dom_{ep} \ \wedge\ R \subseteq X \cup Y\}$.

Proof. (1) The proof is immediate since $\alpha PP_\Gamma = \bigcup_{i \in \Gamma} \alpha B_i \subseteq \alpha Q$ (we assume that, if a process can execute exactly the traces of Dom_i, then its alphabet is αB_i).

(2) We first observe that $\delta PP_i = \emptyset$ and so $\delta PP_\Gamma = \emptyset$. The proof follows from the above, the fact that $t \upharpoonright B_i \in Dom_i$ is met trivially if ep_i is an identity extraction pattern, and the divergence semantics of the parallel composition operator. With respect to the latter, note that, since $\delta PP_\Gamma = \emptyset$ and $\alpha PP_\Gamma \subseteq \alpha Q$, if $w \in \delta\widehat{Q}$ there exists $v \in min\delta\widehat{Q}$, $v \leq w$, such that $v \upharpoonright in\, Q \in Dom_{ep}$ and $v \upharpoonright \alpha Q = v \in \delta Q$. It follows that $v \in min\delta Q$, since otherwise $v \notin min\delta\widehat{Q}$.

(3) We first observe that $\phi_\perp PP_i = \phi PP_i = \{(t, R) \mid t \in Dom_i \ \wedge\ R \subseteq NotNext_i(t)\}$ and that $\phi_\perp PP_\Gamma = \phi PP_\Gamma = \{(t, R) \mid R \subseteq NotNext_\Gamma(t) \ \wedge\ (\forall i \in \Gamma)\ t \upharpoonright B_i \in Dom_i\}$. The proof follows from the above and the fact that $Dom_i = (\alpha B_i)^*$ if $b_i \in b_{id}$. □

---

[5] Processes supplied to FDR2 as input must be such that they can be represented operationally by a finite (variant of a) labelled transition system. Although generic process definitions in this and subsequent sections — such as $PP_i(t)$ — may appear to be parameterized by labels from possibly infinite sets, we assume that there is a finite equivalence relation over these labels, such that if two processes are parameterized by equivalent labels then they are semantically equivalent. In fact, the concrete inputs provided to FDR2 will not be parameterized in such a way — i.e. they will not be parameterized by a particular trace — since the user will define a single distinct process name to represent the set of processes parameterized by equivalent labels.

Corollary 4 *The following hold:*

1. $\tau_\perp \widehat{Q} \subseteq \tau_\perp Q$.
2. $\delta\widehat{Q} \subseteq \delta Q$.
3. *If* $t \in (\tau_\perp Q \setminus \tau_\perp \widehat{Q})$ *then* $t \upharpoonright in\, Q \notin Dom_{ep}$.
4. *If* $(t, R) \in \phi Q$ *and* $t \upharpoonright in\, Q \in Dom_{ep}$, *then* $(t, R) \in \phi\widehat{Q}$.
5. *If* $(t, R) \in \phi\widehat{Q}$ *then there exists* $(t, X) \in \phi Q$ *such that:*
   (a) $X \subseteq R$.
   (b) $R \cap \alpha out\, Q = X \cap \alpha out\, Q$.
   (c) $(R \cap \alpha in\, Q) \setminus (X \cap \alpha in\, Q) \subseteq NotNext_\Gamma(t)$.
   (d) $R \cap \alpha b_{id} = X \cap \alpha b_{id}$.

We now give a supporting result before giving the main result of this section, that Q meets all of the conditions of the implementation relation if and only if $\widehat{Q}$ does.

Lemma 5 *Let* (t, R) *and* (t, X) *be as characterised in corollary 4(5). If* $b_i \in b_{nid}$ *is blocked at* (t, R) *then it is blocked at* (t, X).

Proof. Let b_i be blocked at (t, R). We consider each of two cases in turn.

Case 1: $b_i \in out\,\widehat{Q}$. By corollary 4(5)(b) we know that $R \cap out\,\widehat{Q} = X \cap out\, Q$. Since we know that $\alpha B_i \cap R \notin ref_i(t \upharpoonright B_i)$ it follows that $\alpha B_i \cap X \notin ref_i(t \upharpoonright B_i)$ and so b_i is blocked at (t, X).

Case 2: $b_i \in in\,\widehat{Q}$. In this case, if $a \in (R \cap \alpha B_i) \setminus (X \cap \alpha B_i)$ then $t \upharpoonright B_i \circ \langle a \rangle \notin Dom_i$ by corollary 4(5)(c) and the definition of $NotNext_\Gamma$. By EP4, we know that if $Z \in ref_i(t \upharpoonright B_i)$ then $Z \cup \{a\} \in ref_i(t \upharpoonright B_i)$. Since b_i is blocked at (t, R), we know that $\alpha B_i \setminus R \in ref_i(t \upharpoonright B_i)$. It follows from the above that $\alpha B_i \setminus X \in ref_i(t \upharpoonright B_i)$ and so b_i is blocked at (t, X). □

Theorem 6. *Let* P *be a specification process. Then* $Q \preceq^{ep}_{ep'} P$ *if and only if* $\widehat{Q} \preceq^{ep}_{ep'} P$.

Proof. We first note that if $(t, R) \in \phi_\perp \widehat{Q}$ and $(t, X) \in \phi_\perp Q$, $(t, R) \in \phi_{Dom}\widehat{Q}$ if and only if $(t, X) \in \phi_{Dom} Q$. Likewise, $(t, R) \in \phi_{dom}\widehat{Q}$ if and only if $(t, X) \in \phi_{dom} Q$.

($\Longrightarrow$) Let $Q \preceq^{ep}_{ep'} P$. We establish that $\widehat{Q} \preceq^{ep}_{ep'} P$ by considering each of the conditions in turn.
DP,GE,TE: The proof follows from corollary 4(1).
DF: The proof follows from corollary 4(2). It follows that $\phi_{Dom}\widehat{Q} \subseteq \phi\widehat{Q}$ and also $\phi_{dom}\widehat{Q} \subseteq \phi\widehat{Q}$. This is important because corollary 4(5) only refers to stable failures of $\widehat{Q}$.
LC: We consider each $b_i \in b_{nid}$ in turn and show that if condition LC is not violated in Q by b_i then neither is that condition violated by b_i in $\widehat{Q}$. Let $(t, R) \in \phi_{Dom}\widehat{Q}$, $(t, X) \in \phi_{Dom} Q$ be as characterised in corollary 4(5) and let $b_i \in b_{nid}$. We consider each of two cases in turn.

Case 1: b_i is not blocked at (t, X) in Q. It follows from the contrapositive of lemma 5 that b_i is not blocked at (t, R) in $\widehat{Q}$.

Case 2: b_i is blocked at (t, X) in Q. In this case it follows that $t \upharpoonright B_i \in dom_i$ and the proof is immediate.
RE: Let $(t, R) \in \phi_{dom}\widehat{Q}$. We know that $(t, R) \in \phi\widehat{Q}$. Let $(t, X) \in \phi Q$ be as characterised in corollary 4(5). We also have that $(t, X) \in \phi_{dom} Q$. By definition of condition RE, it suffices to show that any channel $b_i \in b_{nid}$ is blocked at (t, X) in Q if it is blocked at (t, R) in $\widehat{Q}$ and also that $R \cap \alpha b_{id} = X \cap \alpha b_{id}$. That the former requirement is met follows from lemma 5. The latter requirement is met due to corollary 4(5)(d).

($\Longleftarrow$) Let $\widehat{Q} \preceq^{ep}_{ep'} P$. We establish that $Q \preceq^{ep}_{ep'} P$ by considering each of the conditions in turn.
DP,GE,TE: The proof follows from corollary 4(1) and 4(3).
DF: The proof follows from corollary 4(2) and 4(3). It follows that $\phi_{Dom} Q \subseteq \phi Q$ and also $\phi_{dom} Q \subseteq \phi Q$. This is important because corollary 4(4) only refers to stable failures of Q.
LC,RE: We prove a contradiction. Assume that $(t, R) \in \phi_{Dom} Q$ ($(t, R) \in \phi_{dom} Q$) is a failure at which condition LC (RE) is violated in Q. We then have that $(t, R) \in \phi Q$. By corollary 4(4), we have that $(t, R) \in \phi\widehat{Q}$ and so $(t, R) \in \phi_{Dom}\widehat{Q}$ ($(t, R) \in \phi_{dom}\widehat{Q}$). As a result, condition LC (RE) would be violated at (t, R) in $\widehat{Q}$ and so we have a contradiction. □

As a result of the above theorem, we are able to verify that Q is a valid implementation of P according to the scheme presented here by instead verifying that $\widehat{Q}$ is a valid implementation of P.

Although in the proofs above we have assumed that all of the PP_i are composed in parallel before being combined with Q to generate $\widehat{Q}$, in practice we would compose each PP_i with Q in turn, which is possible due to the associativity of the parallel composition operator. This is desirable since it avoids the state explosion which might result from composing the (disjoint) PP_i in parallel first.

5 Checking conditions DP and DF

We are now in a position to begin checking the first two conditions, DP and DF, which are as follows (according to theorem 6, we may substitute $\widehat{Q}$ for Q):

DF $\delta Q \cap \tau_{Dom} Q = \emptyset$.
DP If $t \in \tau_\perp Q$ and $t \upharpoonright in\, Q \in Dom_{ep}$ then $t \in \tau_{Dom} Q$.

Although the checking of subsequent conditions will allow us to *abstract* the behaviour of $\widehat{Q}$ and interpret it in terms of the interface of the specification process P, conditions DP and DF are checked at the level of abstraction at which $\widehat{Q}$ is expressed. (Condition LC is similar since it, too, makes no reference to the behaviour of $\widehat{Q}$ after extraction.)

Two checks are carried out to ensure that $\widehat{Q}$ meets conditions DP and DF. The first is a divergence-freeness check on $\widehat{Q}$, obviously carried out in the failures divergences model. The other check needs to ensure that $\tau_\perp \widehat{Q} \subseteq Dom_{ep \cup ep'}$. However, since we know by this point that $\delta\widehat{Q} = \emptyset$, we are able to simply check that $\tau\widehat{Q} \subseteq Dom_{ep\cup ep'}$. As we will construct the processes to represent $Dom_{ep\cup ep'}$ to be divergence-free, we can carry out the relevant check in the traces model.

Rather than define a process with the traces of $Dom_{ep\cup ep'}$ and use it as the specification process in a refinement check with $\widehat{Q}$, we take the following approach, based on the way in which $Dom_{ep\cup ep'}$ is defined in EP6 according to each Dom_i. As a result, we carry out a refinement check for each relevant extraction pattern in turn, rather than carrying out a single refinement check. (Note that we need only carry out the refinement check for non-identity extraction patterns referring to *output* channels.)

Theorem 7. *$\widehat{Q}$ meets conditions DP and DF if and only if $\delta\widehat{Q} = \emptyset$ and, for every $t \in \tau\widehat{Q}$ and every $ep_i \in ep'$ such that $b_i \in b_{nid}$, $t \upharpoonright B_i \in Dom_i$.*

Proof. ($\Longrightarrow$) Since $\widehat{Q}$ meets conditions DP and DF, we have immediately that if $t \in \tau_\perp\widehat{Q}$ and $t \upharpoonright in\,\widehat{Q} \in Dom_{ep}$, then $t \notin \delta\widehat{Q}$. By lemma 3(2), it follows that $\delta\widehat{Q} = \emptyset$. Since $\delta\widehat{Q} = \emptyset$, $\tau\widehat{Q} = \tau_\perp\widehat{Q}$ and it follows by lemma 3(3) that if $t \in \tau\widehat{Q}$, then $t \upharpoonright in\,\widehat{Q} \in Dom_{ep}$. Since $\widehat{Q}$ meets condition DP, it follows that if $t \in \tau\widehat{Q}$ then $t \in \tau_{Dom}\widehat{Q}$. By EP6, we have that if $t \in \tau\widehat{Q}$, then, for every $ep_i \in ep'$, $t \upharpoonright B_i \in Dom_i$. The proof follows from the above.

($\Longleftarrow$) Since $\delta\widehat{Q} = \emptyset$, it follows trivially that $\widehat{Q}$ meets condition DF. It also follows that $\tau\widehat{Q} = \tau_\perp\widehat{Q}$. By this fact, EP6 and the fact that $Dom_i = (\alpha B_i)^*$ if $b_i \in b_{id}$, we have that if $t \in \tau_\perp\widehat{Q}$, then $t \upharpoonright out\,\widehat{Q} \in Dom_{ep'}$. It follows by EP6 and the definition of DP that $\widehat{Q}$ meets condition DP. □

Corollary 8 *If $\widehat{Q}$ meets conditions DP and DF, then $\tau\widehat{Q} = \tau_\perp\widehat{Q}$.*

We define a separate specification process for each $ep_i \in ep'$ such that $b_i \in b_{nid}$ and carry out a refinement check in the traces model between that specification and $\widehat{Q}$ with all events hidden which are not in αB_i.

We denote the specification process for each ep_i as DP_i and define it as $DP_i \triangleq DP_i(\langle\rangle)$, where $DP_i(t) \triangleq \Box_{a\in Next(t)} a \to DP_i(t \circ \langle a\rangle)$. The following result then lets us check conditions DF and DP using FDR2.

Theorem 9. *$\widehat{Q}$ meets conditions DP and DF if and only if $\delta\widehat{Q} = \emptyset$ and, for every $ep_i \in ep'$ such that $b_i \in b_{nid}$, $DP_i \sqsubseteq_T \widehat{Q} \setminus (\alpha\widehat{Q} \setminus \alpha B_i)$.*

Proof. We first observe that $\tau DP_i = Dom_i$ by a straightforward induction on the traces of DP_i (respectively Dom_i). The proof follows from the above, theorem 7 and the fact that, in the traces model, $\{w \upharpoonright B_i \mid w \in \tau\widehat{Q}\} = \tau(\widehat{Q} \setminus (\alpha\widehat{Q} \setminus \alpha B_i))$. □

6 Checking condition TE

The condition to be checked is as follows:

TE $extr_{ep \cup ep'}(\tau_{Dom} Q) \subseteq \tau_\perp P$.

When checking this condition, we first assume that $\widehat{Q}$ has already successfully passed the verification checks for conditions DP and DF. This allows us to derive the following results, appealed to in the checking of condition TE.

Lemma 10 *Assume that $\widehat{Q}$ meets conditions DP and DF. Then $\tau_\perp\widehat{Q} = \tau\widehat{Q} = \tau_{Dom}\widehat{Q}$.*

Proof. By theorem 7, $\delta\widehat{Q} = \emptyset$ and so $\tau_\perp\widehat{Q} = \tau\widehat{Q}$. The proof follows from lemma 3(3) and the fact that $\widehat{Q}$ meets condition DP. □

Corollary 11 *$\widehat{Q}$ meets condition TE if and only if $extr_{ep \cup ep}(\tau\widehat{Q}) \subseteq \tau P$.*

Remember that we assume $\delta P = \emptyset$ and so $\tau_\perp P = \tau P$. Corollary 11 allows us to check condition TE working only in the traces model and so all results and proofs in the remainder of this section assume we are working in that model. By virtue of that same corollary, we must generate an implementation process to be supplied as an input to a refinement check in FDR2 such that it has precisely the extracted traces of $\widehat{Q}$. This means that we must generate a process context in which to place $\widehat{Q}$ such that the context encodes the extraction function *extr* (a function from traces to traces). We show exactly how to do this below.

Intuitively, the approach is similar to that employed to *extract* traces in the algorithms given in [4]. For each extraction pattern ep_i, we define a process TE_i, such that, in a sense, each event in the alphabet of TE_i is a *pair*. One of the events of the pair comes from αB_i — i.e. the events from the implementation alphabet which are interpreted by ep_i — and the other is either $\langle\rangle$ or an event from αb_i — i.e. the events from the specification which ep_i can interpret as having occurred after extraction. If we project the traces from TE_i — using renaming — onto the left-hand event of each pair, then we get the traces of Dom_i. If we project — using hiding and renaming — onto the right-hand event of each pair then we get the traces representing $extr_i(Dom_i)$. (Note that the notions of left-hand and right-hand event are used purely for ease of expression and have no other significance.)

Since CSP has no operator itself which allows direct definition of a function from traces to traces, then the user must define the process TE_i explicitly, rather than simply applying some operator to a process which encodes Dom_i (where $b_i \in b_{nid}$): that is, the user has to encode the extraction function explicitly.

6.1 Extraction pattern construction

We now show how to construct the process TE_i. The main problem to address is the nature of the events that will be used — necessarily within the syntax of machine-readable CSP — to represent the pairs of events described informally above. In machine-readable CSP, events are of the form *channelname.dataval*, where *dataval* is a value of type *datatype* and *datatype* is the type of the channel *channelname*. Note that *datatype* may be an empty data type — leaving the channel name as a simple event (in this case, *dataval* is not used) — a simple data type or a tuple combining a number of data types. In the event that the right-hand event of a pair is $\langle\,\rangle$ then we can simply leave the event as it was originally. However, if this is not the case, a *pair* of events have to be encoded by a *single* event occurring on a single channel. As a result, we are required to define a number of new channels, with corresponding new data type extensions: the new data type will represent the data values which may be sent on the original channel, along with both the channel name and data type of events which may form the right-hand component of a pair. In general, the approach to be taken will be as follows.

We take a channel from B_i on which an event occurs that forms the left-hand event of a pair as described above. Assume that this channel has the form *chan.dat*, where *chan* is the *name* of the channel and *dat* is the *type* of the data that it carries. The event which is the right-hand event of the pair will occur on channel b_i and we assume that this event has the form $b_i.specDat$ (again, b_i is the name of the channel and *specDat* is the type of the data that it carries). Then we create a new channel, called $extract_{chan}$, where the name of the original channel may be derived from the subscript of the new name. The data type of this new channel is *dat.name.specDat* — a data tuple is defined in FDR2 using dot notation — where *name* is a data type containing a single value, namely the label of the channel b_i.[6]

If we consider the running example and ep_{twice} introduced in section 3, the trace $\langle data.0, ack.yes\rangle$ extracts to $\langle mid.0\rangle$, where b_i is the channel *mid*. If it were possible to use the notion of event pairs directly, we could encode this using the trace $\langle(data.0, \langle\,\rangle), (ack.yes, mid.0)\rangle$. Since we cannot use such pairs directly, in the representation of the extraction pattern this trace would become $\langle data.0,$ $extract_{ack}.yes.mid.0\rangle$. (Notice that *data*.0 remains as the original event since the right-hand component of the pair of which it was the left-hand component is simply $\langle\,\rangle$.) As another example, consider the trace $\langle data.0, ack.no, data.0\rangle$, also extracting to *mid*.0. In this case, using event pairs, we would have $\langle(data.0, \langle\,\rangle), (ack.no, \langle\,\rangle), (data.0, mid.0)\rangle$ In the extraction pattern representation the trace would become $\langle data.0, ack.no,\ extract_{data}.0.mid.0\rangle$. Note that we have both an occurrence of the unchanged *data*.0 and also an occurrence of *data*.0 modified to allow the extraction function to be encoded. Note also that in machine-readable CSP we cannot have channel names containing subscripts: we use the device here for the purposes of presentation and, in practice, would have to define channels such as, for example, *extractchan* or *extractdata*.

Renaming functions We now give the renaming functions[7] which can be used to reclaim Dom_i and $extr_i(Dom_i)$ respectively from τTE_i. The renaming $domain_i$ will return Dom_i and $extract_i$ will return $extr_i(Dom_i)$. The functions $domain_i$ and $extract_i$ are both partial, defined only for those events which have a non-empty right-hand component. In what follows, we assume that x has the form $extract_{chan}.data.b_i.val$, where *data* and *val* are specific instances of data values; moreover, $chan \in B_i$. The type of $extract_i$ is $extract_i : \alpha TE_i \rightharpoonup \alpha b_i$ and it is defined as $extract_i(x) \triangleq b_i.val$. The type of $domain_i$ is $domain_i : \alpha TE_i \rightharpoonup \alpha B_i$ and it is defined as $domain_i(x) \triangleq chan.data$. (Note that these functions are defined only for values of the form of x and not for values in αB_i. However, as with all relations, we assume that partial functions are extended by the identity mapping for all elements of the domain for which the function is undefined.)

Defining TE_i For each ep_i such that $b_i \in b_{nid}$ we encode the extraction function as a process TE_i, where $TE_i \triangleq TE_i(\langle\rangle)$ and

$$TE_i(t) \triangleq \Box_{a\in\pi_i(t)} a \rightarrow TE_i(t \circ domain_i(a)).$$

[6] Note that the textual representations of the channel name b_i as a value of type *name* and as a channel identifier are the same; whenever the label is used in what follows, the type of value which it denotes will be clear from the context.

[7] Although, in the general case, we use renaming relations, it happens that these are functions.

For ease of expression, the function π_i is used here to encode the modifications that must be made to the events of αB_i, although its effects must be implemented directly in any input supplied to FDR2, since it cannot be encoded as such in CSP (see the example below in section 6.2). We have $\pi_i(t) \triangleq \{\lambda_i(a,t) \mid a \in Next_i(t)\}$, where

$$\lambda_i(a,t) \triangleq \begin{cases} a & \text{if } extr_i(t) = extr_i(t \circ \langle a \rangle) \\ \\ extract_{chan}.data.b_i.val & \text{if } extr_i(t) \circ \langle e \rangle = extr_i(t \circ \langle a \rangle), \text{ where} \\ & \quad a = chan.data \text{ and } e = b_i.val. \end{cases}$$

Observe that $\lambda_i(a,t)$ simply returns a if the extraction of t is identical to the extraction of $t \circ \langle a \rangle$. In this case, the relevant event pair in τTE_i would have a as the left-hand component and $\langle\,\rangle$ as the right-hand component. In the other case — namely that the extraction of t is a strict prefix of the extraction of $t \circ \langle a \rangle$ — we are effectively encoding the fact that a is the left-hand component of the event pair and e is the right-hand component.

We now give a lemma which formalises the fact that $\tau(TE_i[domain_i]) = Dom_i$ and $\tau((TE_i[extract_i]) \setminus \alpha B_i) = extr_i(Dom_i)$.

Lemma 12 *The following results hold:*

1. *If $w \in \tau TE_i$ then $domain_i(w) \in Dom_i$.*
2. *If $t \in Dom_i$, there exists $w \in \tau TE_i$ such that $domain_i(w) = t$.*
3. *If $w \in \tau TE_i$, then $(extract_i(w)) \setminus \alpha B_i = extr_i(domain_i(w))$.*

Proof. (1) The proof is a straightforward induction on the length of w using the definitions of π_i and $domain_i$.

(2) The proof in this case is a straightforward induction on the length of t using the definitions of π_i and $domain_i$.

(3) The proof is once more a straightforward induction on the length of w, this time using the definitions of π_i, $domain_i$ and $extract_i$. In addition, $extr_i(domain_i(w))$ is well-defined by part (1) of the lemma. □

We now define renaming functions *domain* and *extract* as follows:

$$domain \triangleq \bigcup_{i \in I} domain_i \qquad \text{and} \qquad extract \triangleq \bigcup_{i \in I} extract_i.$$

Note that *domain* and *extract* are well-defined by the disjointness of αB_i and αB_j when $i \neq j$. It is unnecessary to define processes TE_i for the identity extraction patterns ep_i, where $b_i \in b_{id}$, since for traces over events in $\alpha B_i = \alpha b_i$, the extraction mapping, $extr_i$, is simply the identity mapping. We thus define TE_I as:

$$TE_I \triangleq \|_{i \in I}\ TE_i.$$

Although it may not be immediately obvious, the alphabets of distinct TE_i are disjoint. This disjointness — which is due to the disjointness of the αB_i and also the disjointness of the αb_i — can be easily observed from the following lemma.

Lemma 13 *$\alpha TE_i = A_1 \cup A_2$, where:*

- $A_1 \triangleq \{a \mid a \in \alpha B_i \ \wedge\ ((\exists t)\ t \circ \langle a \rangle \in Dom_i \ \wedge\ extr_i(t) = extr_i(t \circ \langle a \rangle))\}$ *and*
- $A_2 \triangleq \{extract_{chan}.data.b_i.val \mid (\exists a = chan.data \in \alpha B_i, e = b_i.val \in \alpha b_i, t \in Dom_i)\ t \circ \langle a \rangle \in Dom_i \ \wedge\ extr_i(t) \circ e = extr_i(t \circ \langle a \rangle)\}$.

Proof. The proof is immediate from the definitions above of π_i and λ_i. □

Corollary 14 *If $i \neq j$ then $\alpha TE_i \cap \alpha TE_j = \emptyset$.*

Extracting the traces of $\widehat{Q}$ Once TE_I has been defined, we wish to compose it in parallel with $\widehat{Q}$ before applying the hiding and renaming which will mimic the application of the extraction mapping. In order for $\widehat{Q}$ to synchronize with TE_I, we have to rename its events: each event of $\widehat{Q}$ is renamed to all those events in αTE_I of which it might form the left-hand component. If an event may only form the left-hand component of an event pair where the right-hand component is the empty trace, then we do not need to rename that event at all.

We now define the renaming, $prep : \alpha B_I \times \alpha TE_I$, which is applied to $\widehat{Q}$ (see section 6.2 for the renaming *prep* used when verifying condition TE for the processes which appear in the running example). If $a \in \alpha B_i$:

$$prep(a) \triangleq \{extract_{chan}.data.b_i.val \mid a = chan.data \ \wedge \ extract_{chan}.data.b_i.val \in \alpha TE_i\} \cup (\{a\} \cap \alpha TE_i).$$

The final clause of the above definition is simply to ensure that any event a is also renamed to itself where necessary.

We now give the definition of the process, *INTERP*, which, in the traces model, has exactly the traces of $\widehat{Q}$ after extraction:

$$INTERP \triangleq ((\widehat{Q}[prep] \parallel TE_I)[extract]) \setminus \alpha B_{nid}.$$

Lemma 15 $(\widehat{Q}[prep] \parallel TE_I)[domain] =_T \widehat{Q}$.

Proof. ($\subseteq$) Let $t \in \tau(\widehat{Q}[prep] \parallel TE_I)[domain]$. Then there exists $w \in \tau(\widehat{Q}[prep] \parallel TE_I)$ such that $domain(w) = t$. Since $\alpha TE_I \subseteq \alpha\widehat{Q}[prep]$, we have that $w \in \tau\widehat{Q}[prep]$. It follows by definition of *prep* that $domain(w) = t \in \tau\widehat{Q}$.

($\supseteq$) Let $t \in \tau\widehat{Q}$. We know that $t \in Dom_{ep\cup ep'}$ by lemma 10. By lemma 12(2) and EP6, there exists $w \in \tau TE_I$ such that $domain(w) = t \upharpoonright B_{nid}$. By definition of *prep*, it follows that there exists $u \in \tau(\widehat{Q}[prep] \parallel TE_I)$ such that $domain(u) = t$. □

Lemma 16 *Let* $w \in \tau(\widehat{Q}[prep] \parallel TE_I)$ *and* $domain(w) = t$. *Then* $extract(w) \setminus \alpha B_{nid} = extr_{ep\cup ep'}(t)$.

Proof. By lemma 12(3), $extract_i(w \upharpoonright \alpha TE_i) \setminus \alpha B_i = extr_i(domain_i(w \upharpoonright \alpha TE_i))$, for each $i \in I$. By definition of αTE_i and *domain*, we observe that $domain(\alpha TE_i) = \alpha B_i$. From the fact that $domain_i$ and $domain_j$ have disjoint ranges for $i \neq j$, we then have $domain(w \upharpoonright \alpha TE_i) = domain(w) \upharpoonright domain(\alpha TE_i) = domain(w) \upharpoonright \alpha B_i = t \upharpoonright \alpha B_i$. It follows that $extract_i(w \upharpoonright \alpha TE_i) \setminus \alpha B_i = extr_i(t \upharpoonright \alpha B_i)$. Moreover, for all $1 \leq j \leq m+n$ such that $j \notin I$ — i.e. $b_j \in b_{id}$ — we know that $extr_j(w \upharpoonright \alpha B_j) = w \upharpoonright \alpha B_j$ by definition of the extraction mapping for uninterpreted channels and also that $(extract(w) \setminus \alpha B_{nid}) \upharpoonright \alpha B_j = w \upharpoonright \alpha B_j$. The proof follows from the above and EP7. □

We are now able to give the final theorem showing how we may check condition TE using FDR2.

Theorem 17. $\widehat{Q}$ *meets condition TE if and only if* $P \sqsubseteq_T INTERP$.

Proof. By corollary 11 we need only prove that $extr_{ep\cup ep'}(\tau\widehat{Q}) \subseteq \tau P$ if and only if $P \sqsubseteq_T INTERP$.

($\Longrightarrow$) Assume that $extr_{ep\cup ep'}(\tau\widehat{Q}) \subseteq \tau P$. Let $t \in \tau INTERP$. Then there exists $w \in \tau(\widehat{Q}[prep] \parallel TE_I)$ such that $extract(w) \setminus \alpha B_{nid} = t$. By lemma 15, we have that $domain(w) = u \in \tau\widehat{Q}$. By lemma 16, $t = extr_{ep\cup ep'}(u)$. It follows that $t \in \tau P$ and so $P \sqsubseteq_T INTERP$.

($\Longleftarrow$) Assume that $P \sqsubseteq_T INTERP$. Let $t \in \tau\widehat{Q}$. By lemma 15, we know there exists $w \in \tau(\widehat{Q}[prep] \parallel TE_I)$ such that $domain(w) = t$. By lemma 16, we know that $extract(w) \setminus \alpha B_{nid} = extr_{ep\cup ep'}(t)$ and so $extr_{ep\cup ep'}(t) \in \tau INTERP$. Since $P \sqsubseteq_T INTERP$ we have that $extr_{ep\cup ep'}(t) \in \tau P$ and so $extr_{ep\cup ep'}(\tau\widehat{Q}) \subseteq \tau P$. □

For ease of expression and manipulation, the above results imply that we would compose the extraction pattern representations TE_i such that $i \in I$ in parallel and then compose the result with $\widehat{Q}$. This would be inefficient, since the composition of the extraction pattern processes, all of which are disjoint, could be very large indeed. As a result, we would compose each such process with $\widehat{Q}$ individually. We are able to do this since parallel composition is associative.

6.2 Example

Here we show how to apply the results in this section to verify that Buf_{imple} meets condition TE with respect to Buf_{spec}. (Note that the components defined here could be used equally well without modification to verify that $Send_{imple}$ meets condition TE with respect to $Send_{spec}$, since condition TE is not concerned with whether channel b_i for extraction pattern ep_i is an input or an output channel. This contrasts with condition LC, for example, which does distinguish between input and output channels.) Let $\widehat{Q}$ be Buf_{imple} after the application of the necessary preprocessing. Let ep_1 be ep_{twice}. (Note that $B_1 = \{data, ack\}$.) We define the process TE_1 as follows:

$$TE_1 \triangleq \Box_{x\in\{0,1\}}\, data.x \rightarrow ((extract_{ack}.yes.mid.x \rightarrow TE_1) \Box (ack.no \rightarrow extract_{data}.x.mid.x \rightarrow TE_1)).$$

We then give the concrete renamings $prep$ and $extract_1$ used in this example:

$$prep \triangleq \begin{cases} \{(ack.yes, extract_{ack}.yes.mid.0), (ack.yes, extract_{ack}.yes.mid.1), \\ (data.0, data.0), (data.1, data.1), (ack.no, ack.no), \\ (data.0, extract_{data}.0.mid.0), (data.1, extract_{data}.1.mid.1)\} \end{cases}$$

$$extract_1 \triangleq \begin{cases} \{(extract_{ack}.yes.mid.0, mid.0), (extract_{ack}.yes.mid.1, mid.1), \\ (extract_{data}.0.mid.0, mid.0), (extract_{data}.1.mid.1, mid.1)\} \end{cases}$$

Note that here $extract = extract_1$ and $B_{nid} = B_1$. Using the process expressions defined here, we were able to define $INTERP$ as above and verify automatically using FDR2 that Buf_{imple} meets condition TE with respect to Buf_{spec}.

7 Checking condition GE

The condition to be checked is:

GE If $\ldots, t_i, \ldots$ is an ω-sequence in $\tau_{Dom}Q$, then $\ldots, extr_{ep\cup ep'}(t_i), \ldots$ is also an ω-sequence.

Once again we assume that we have successfully verified conditions DP and DF. Checking condition GE simply requires that we check $INTERP$ for divergence-freeness (and so obviously we work in the failures divergences model).

Lemma 18 *$\delta TE_I = \emptyset$, where TE_I is as defined in section 6.*

Proof. We first have that $\delta TE_i = \emptyset$ for $i \in I$ since neither the deterministic choice operator nor the prefix operator can introduce divergence. The proof then follows from the above and the fact that the parallel composition operator cannot introduce divergence. □

Theorem 19. *$\widehat{Q}$ meets condition GE if and only if $\delta INTERP = \emptyset$.*

Proof. In what follows, we extend the renaming relations *domain* and *extract* and the mapping *extr* to infinite traces in the usual way. (Remember that we assume our input processes are finite state and so finitely non-deterministic.)

($\Longrightarrow$) We prove a contradiction. Let $t \in min\delta(INTERP)$. Since $\delta\widehat{Q} = \emptyset$ by theorem 7, $\delta TE_I = \emptyset$ by lemma 18 and neither renaming nor parallel composition can introduce divergence, it follows that $\delta((\widehat{Q}[prep] \parallel TE_I)[extract]) = \emptyset$. It follows that there exists $w \in \tau(\widehat{Q}[prep] \parallel TE_I)$ such that $w = u \circ v$ and

$extract(v) = v \in (\alpha B_{nid})^\omega$ (note that $domain(w) \in \tau\widehat{Q}$ by lemma 15). As a result, $domain(w) \in (\alpha\widehat{Q})^\omega$ while $extract(w) \setminus \alpha B_{nid}$ is finite and so $extr_{ep \cup ep}(domain(w))$ is finite by lemma 16. This means that condition GE is not met by $\widehat{Q}$ and so we have a contradiction.

($\Longleftarrow$) Since $\delta INTERP = \emptyset$, we know that there does not exist a trace $w \in \tau(\widehat{Q}[prep] \parallel TE_I)$ such that $w = u \circ v$ and $extract(v) = v \in (\alpha B_{nid})^\omega$. It follows from the above and lemmas 15 and 16 that for every $t \in (\alpha\widehat{Q})^\omega \cap \tau\widehat{Q}$, $extr_{ep \cup ep'}(t)$ is of infinite length. □

8 Checking condition LC

The condition to be checked is as follows:

LC If $b_i \in b_{nid}$ and b_i is a channel of P blocked at $(t, R) \in \phi_{Dom}Q$, then $t \upharpoonright B_i \in dom_i$.

We again assume that conditions DP and DF have been verified successfully by this point. From this and theorem 7 we can deduce that $\delta\widehat{Q} = \emptyset$ and so $\phi\widehat{Q} = \phi_\perp\widehat{Q}$. Below, we transform the check for condition LC into a check for deadlock freedom on a process derived from $\widehat{Q}$. Since deadlock freedom is checked in the stable failures model, we are only able to effect this transformation due to the fact that $\phi\widehat{Q} = \phi_\perp\widehat{Q}$ (that is, we do not lose any information on failures by only working in the stable failures model).

The means used to check condition LC is derived directly from the rationale behind the condition and is similar in some respects to the use of tester processes in [1], where the question of whether one process implements another is transformed into a question of deadlock-freedom of the implementation composed in parallel with the tester process. We first explain what it means that condition LC is met by a process or processes.

If we take two (implementation) processes M and N which meet condition LC with respect to a channel b_i and compose them in parallel on the channel set B_i, then at any state reachable in the resulting composed process by a trace t such that $t \upharpoonright B_i \notin dom_i$, at least one event from B_i will be synchronized upon and so enabled, meaning that deadlock will not result on that channel set. This is achieved in the following way. A set of refusal bounds are defined for the process M which is notionally a sender process with respect to the channels B_i. When behaviour over the channels B_i is incomplete, then the events refused by M must be contained within one of a set of refusal bounds (the ref_i component of the extraction pattern ep_i). Or, to put it another way, there are sets of events to be offered and M must offer all the events in at least one of the sets. The receiver process N must then guarantee that it offers at least one event from each of the sets of events one of which must be offered in its entirety by M. This guarantees that the sender (M) and receiver (N) processes (remember that these notions of sender and receiver are defined here only with respect to the channel set B_i) will synchronize on at least one event and deadlock will not arise across B_i.

Conversely, if either M or N fails to meet condition LC with respect to the channel set B_i, then deadlock *may* arise across B_i when the two processes are composed in parallel. We therefore create a tester process such that, if our implementation process fails to meet condition LC, then deadlock *will* arise across the channel set B_i, in the parallel composition of the tester and the implementation, at some point where behaviour is incomplete with respect to dom_i. Finally, in order that we may transform the checking of condition LC into a check for deadlock freedom across a process as a whole, we must isolate the refusals of the implementation process with respect to the channel set B_i. This latter fact means that we check the condition for each extraction pattern in turn — although once more we need not check the condition for identity extraction patterns — which allows for greater compression since we hide all events not on the channels B_i.

Transforming the implementation process We now show how to transform the implementation process $\widehat{Q}$ such that its failures are projected onto αB_i (that is, if $(t, R) \in \phi\widehat{Q}$, then $(t \upharpoonright B_i, R \cap \alpha B_i)$ is a failure of the transformed process). Since we may work in the stable failures model, we are able to use the process *DIV*, the immediately diverging process, to obscure failures in which we are not interested without adding behaviours to the resulting process. We are able to do this in the following way.

Let P be a process expression. Then $\phi(P \,\Box\, \mathit{DIV}) = \{(t, R) \in \phi P \mid t \neq \langle\rangle\}$. That this is the case follows from the definition of $\Box$ in the stable failures model and the fact that the meaning of *DIV* in this model is $(\{\langle\rangle\}, \emptyset)$ (the alphabet component is omitted here). In particular, we are able to obscure failures derived from states where behaviour is complete over the channel set B_i, since condition LC places no restriction on what may be refused in those situations. This aproach of working in the stable failures model and (crucially) using *DIV* is also used in the checking of condition RE.

The following process, PROC_i, is such that after any trace we may non-deterministically arrive at a state which offers all events in $\alpha\widehat{Q}$ but which does not contribute to a failure of PROC_i due to the appearance of *DIV* in that state. Alternatively, we may arrive at a state which offers all events in αB_i and then takes us to a state that is simply immediately divergent whichever event from αB_i we follow.

$$\mathit{PROC}_i \triangleq ((\Box_{a \in \alpha\widehat{Q}} a \to \mathit{PROC}_i) \,\Box\, \mathit{DIV}) \;\sqcap\; (\Box_{a \in \alpha B_i} a \to \mathit{DIV}).$$

We then compose $\widehat{Q}$ in parallel with PROC_i, with the result that, for every failure $(t, R) \in \phi\widehat{Q}$, the refusal R has $\alpha\widehat{Q} \setminus \alpha B_i$ added to it. This means that the refusal $R \cap \alpha B_i$ will survive the hiding of the events in $\alpha\widehat{Q} \setminus \alpha B_i$. This gives us the following process:

$$\widehat{Q}_i \triangleq (\widehat{Q} \parallel \mathit{PROC}_i) \setminus (\alpha\widehat{Q} \setminus \alpha B_i).$$

Note in part (2) of the following lemma that $(w, X) \in \phi_\perp\widehat{Q}$, rather than simply $\phi\widehat{Q}$. This means that we can look at the meaning of the process expression $\widehat{Q}$ in the stable failures model rather than in the failures divergences model and yet we will not lose any failures information by doing so (recall that the conditions of the implementation relation itself are defined in terms of the failures divergences model). This is the fact that allows us to work only in the stable failures model and thus to transform the checking of condition LC into a check for deadlock-freedom.

Lemma 20 *The following hold of* $\widehat{Q}_i$*:*

1. $\alpha\widehat{Q}_i = \alpha B_i$.
2. $\phi\widehat{Q}_i = \{(t, R) \mid (\exists (w, X) \in \phi_\perp\widehat{Q})\ t = w \upharpoonright B_i \ \wedge\ R \subseteq \alpha B_i \cap X\}$.
3. $\tau\widehat{Q}_i = \{t \mid w \in \tau\widehat{Q} \ \wedge\ t = w \upharpoonright B_i\}$.

Proof. (1) We observe that $\alpha\mathit{PROC}_i = \alpha\widehat{Q}$ and the proof of this part follows immediately.

(2) By theorem 7 and the fact that conditions DP and DF have been met, $\delta\widehat{Q} = \emptyset$ and so $\phi\widehat{Q} = \phi_\perp\widehat{Q}$. We observe that $\phi\mathit{PROC}_i = \{(t, R) \mid R \subseteq \alpha\widehat{Q} \setminus \alpha B_i\}$. From this we have that $\phi(\widehat{Q} \parallel \mathit{PROC}_i) = \{(t, R) \mid (\exists (t, X) \in \phi_\perp\widehat{Q})\ R \subseteq (\alpha\widehat{Q} \setminus \alpha B_i) \cup (\alpha B_i \cap X)\}$. The proof of this part follows from the above.

(3) We first observe that $\tau\mathit{PROC}_i = (\alpha\widehat{Q})^*$ and from this we have that $\tau(\widehat{Q} \parallel \mathit{PROC}_i) = \tau\widehat{Q}$. The proof of this part follows from the above. $\square$

Before going on to define the tester process which will be composed with $\widehat{Q}_i$ to give a process which will deadlock if and only if $\widehat{Q}$ does not meet condition LC with respect to channel b_i, we first give some preliminary definitions and results.

8.1 Preliminary definitions and results

The following results and definitions will be used in giving the semantic characterisation of the process to be composed with $\widehat{Q}_i$ and to show that it does indeed encode exactly the properties we wish. These definitions will also be used when we come to consider condition RE. In what follows, the set of maximal refusal bounds for a particular extraction pattern ep_i and trace $t \in Dom_i$ is denoted as $RM^i_t \triangleq \{R \mid R \in ref_i(t) \ \wedge \ (\forall R' \in ref_i(t)) \ R \subseteq R' \Rightarrow R = R'\}$.

The mapping $Off_i : Dom_i \to 2^{\alpha B_i}$ is used to define the possible sets of events one of which must be offered in its entirety by a receiver process on the channel set B_i after a trace t, $t \upharpoonright B_i \notin dom_i$, if it is to meet condition LC with respect to channel b_i. That is, it gives all possible sets of events which include a single event from each set of events one of which must be offered by a sender process over B_i when its refusals are constrained by ref_i. We define

$$Off_i(t) \triangleq \{Y \mid Y \subseteq \alpha B_i \ \wedge \ (\forall R \in RM^i_t)(\exists a \in Y) \ a \notin R\}.$$

and impose the restriction that all sets in $Off_i(t)$ are minimal under the subset ordering.

The mapping $RefSets_i : Dom_i \to 2^{\alpha B_i}$ gives the set of events which the tester process to be composed with $\widehat{Q}_i$ may refuse after the trace t. Essentially, the set of sets of events returned if b_i is an input channel gives all possible valid refusals for a *sender* implementation process on the channels B_i after the trace t has been executed on those channels (where $t \notin dom_i$) if that implementation process is to meet condition LC for the channel b_i. Likewise, the set of sets of events returned if b_i is an output channel gives all possible valid refusals for a *receiver* implementation process. We define

$$RefSets_i(t) \triangleq \begin{cases} ref_i(t) & \text{if } b_i \in in\,P \\ \{X \mid (\exists Y \in Off_i(t)) \ X \subseteq \alpha B_i \setminus Y\} & \text{if } b_i \in out\,P \end{cases}$$

The reason for defining $RefSets_i$ in this way is that, if one composes $\widehat{Q}_i$ in parallel with a process with all the traces of Dom_i and all possible refusals defined by $RefSets_i$, then the composition will deadlock if and only if $\widehat{Q}$ fails to meet condition LC with respect to channel b_i. Essentially, we check to see if condition LC is met by placing $\widehat{Q}_i$ in parallel with a process which refuses as much on the channel set B_i as may any valid process which meets LC and then see if the circumstances have arisen which condition LC is specifically designed to prevent: i.e. we see if deadlock has occurred on a channel set B_i when behaviour over those channels is incomplete.

The following lemma formalises the points from the previous paragraph.

Lemma 21 *Let b_i be such that $b_i \in b_{nid}$. Then b_i is blocked at the failure (w, X) if and only if there exists $Z \in RefSets_i(w \upharpoonright B_i)$ such that $(\alpha B_i \cap X) \cup Z = \alpha B_i$.*

Proof. In what follows, let $t = w \upharpoonright B_i$.

($\Longrightarrow$) Let b_i be blocked at (w, X).

Case 1: $b_i \in out\,P$. We know that $\alpha B_i \cap X \notin ref_i(t)$. Therefore, for every $S \in ref_i(t)$ there exists $a \in \alpha B_i \cap X$ such that $a \notin S$. It follows that there exists $Y \in Off_i(t)$ such that $Y \subseteq \alpha B_i \cap X$. By definition of $RefSets_i$ we know that $Z \in RefSets_i(t)$, where $Z = \alpha B_i \setminus (\alpha B_i \cap X) \subseteq \alpha B_i \setminus Y$. It follows that $Z \cup (\alpha B_i \cap X) = \alpha B_i$.

Case 2: $b_i \in in\,P$. We know that $\alpha B_i \setminus X \in ref_i(t)$ and so $Z \in RefSets_i(t)$ where $Z = \alpha B_i \setminus X$. It follows that $Z \cup (\alpha B_i \cap X) = \alpha B_i$.

($\Longleftarrow$) Suppose that there exists $Z \in RefSets_i(t)$ such that $(\alpha B_i \cap X) \cup Z = \alpha B_i$. We therefore have that $\alpha B_i \setminus X \subseteq Z$.

Case 1: $b_i \in out\,P$. Z is such that $Z \subseteq \alpha B_i \setminus Y$ for some $Y \in Off_i(t)$. It follows that $\alpha B_i \setminus X \subseteq \alpha B_i \setminus Y$ and so $Y \subseteq \alpha B_i \cap X$. By definition of Off_i we have that $\alpha B_i \cap X \notin ref_i(t)$ and so b_i is blocked at (w, X).

Case 2: $b_i \in in\,P$. It follows by definition of $RefSets_i(t)$ that $Z \in ref_i(t)$ and so $\alpha B_i \setminus X \in ref_i(t)$, meaning that b_i is blocked at (w, X). □

The next two results are used to relate the failures of the tester process defined in section 8.2 to $RefSets_i$.

Lemma 22 *Let $b_i \in in\,P$. Then $\bigcup_{R \in RM^i_t} \{R' \mid R' \subseteq R\} = \{S \mid S \in RefSets_i(t)\}$.*

Proof. The proof follows from the the definition of $\mathit{RefSets}_i(t)$ and the fact that $\mathit{ref}_i(t)$ is the subset-closure of RM^i_t. $\square$

Lemma 23 *Let $b_i \in out\,P$. Then* $\bigcap_{R\in RM^i_t}(\bigcup_{a\in\alpha B_i\setminus R}\{X\subseteq\alpha B_i \mid a\notin X\}) = \{S \mid S\in \mathit{RefSets}_i(t)\}$.

Proof. ($\subseteq$) Let $X\in\bigcap_{R\in RM^i_t}(\bigcup_{a\in\alpha B_i\setminus R}\{X \mid a\notin X\})$. It follows that for every $R\in RM^i_t$, there exists $a\in\alpha B_i\setminus R$ such that $a\notin X$. As a result, there exists $Y\in \mathit{Off}_i(t)$ such that $Y\cap X=\emptyset$. It follows that $X\subseteq\alpha B_i\setminus Y$ and so $X\in \mathit{RefSets}_i(t)$.

($\supseteq$) Let $S\in \mathit{RefSets}_i(t)$. We have that $S\subseteq\alpha B_i\setminus Y$ for some $Y\in \mathit{Off}_i(t)$. We therefore have that $S\cap Y=\emptyset$ and so for every $R\in RM^i_t$, there exists $a\notin R$ such that $a\in Y$ and so $a\notin S$. It follows that for every $R\in RM^i_t$ there exists $a\in\alpha B_i\setminus R$ such that $a\notin S$. The proof in this direction follows from the above. $\square$

8.2 The tester process for channel set B_i

We now define the tester process, LCEP_i, which has exactly the traces of Dom_i and exactly the refusals allowed by $\mathit{RefSets}_i$. After a trace $t\notin \mathit{dom}_i$ in LCEP_i we may arrive non-deterministically at one of two types of state. At the first are enabled all events which are valid extensions of t with respect to Dom_i. This ensures that $\tau \mathit{LCEP}_i$ contains all of the traces of Dom_i. That this type of state does not contribute to a failure of LCEP_i is ensured by the appearance of DIV as an argument to the deterministic choice operator. The other type of state is used to generate the refusals which are allowed by $\mathit{RefSets}_i(t)$; after an event has been executed from one of these second type of state, we proceed to a state which is equivalent to the immediately diverging process. This means that the second type of state contributes to a refusal of LCEP_i after t but that it contributes no more than that (recall the meaning of DIV in the stable failures model).

In the event that b_i is an input channel, it is relatively straightforward to generate the necessary refusal sets after $t\notin \mathit{dom}_i$: to generate a *single, maximal* refusal set we simply *offer* the *complement* of a maximal set from $\mathit{ref}_i(t)$. We then use the non-deterministic choice operator indexed over the set of maximal sets from $\mathit{ref}_i(t)$ to make sure that *all* of the necessary refusal sets are generated.

In the event that b_i is an output channel, we wish to offer, at each relevant state, an event from the complement of *each* maximal set in $\mathit{ref}_i(t)$: as a result, we index the deterministic choice operator with the set of maximal sets from $\mathit{ref}_i(t)$ and then use the non-deterministic choice operator to pick an event from the complement of a maximal set.

We first give the definition of LCEP_i when b_i is an input channel. In this case, $\mathit{LCEP}_i \triangleq \mathit{LCEP}^{in}_i(\langle\rangle)$.

$$\mathit{LCEP}^{in}_i(t) \triangleq \begin{cases} (\square_{a\in Next_i(t)} a \to \mathit{LCEP}^{in}_i(t\circ\langle a\rangle)) \;\square\; \mathit{DIV} & \text{if } t\in \mathit{dom}_i \\ ((\square_{a\in Next_i(t)} a \to \mathit{LCEP}^{in}_i(t\circ\langle a\rangle)) \;\square\; \mathit{DIV}) \sqcap & \text{if } t\in \mathit{Dom}_i\setminus \mathit{dom}_i \\ (\sqcap_{R\in RM^i_t}(\square_{a\in(\alpha B_i\setminus R)} a\to \mathit{DIV})) & \end{cases}$$

We now give the the definition of LCEP_i when b_i is an output channel. In this case, $\mathit{LCEP}_i \triangleq \mathit{LCEP}^{out}_i(\langle\rangle)$.

$$\mathit{LCEP}^{out}_i(t) \triangleq \begin{cases} (\square_{a\in Next_i(t)} a \to \mathit{LCEP}^{out}_i(t\circ\langle a\rangle)) \;\square\; \mathit{DIV} & \text{if } t\in \mathit{dom}_i \\ ((\square_{a\in Next_i(t)} a \to \mathit{LCEP}^{out}_i(t\circ\langle a\rangle)) \;\square\; \mathit{DIV}) \sqcap & \text{if } t\in \mathit{Dom}_i\setminus \mathit{dom}_i \\ (\square_{R\in RM^i_t}(\sqcap_{a\in(\alpha B_i\setminus R)} a\to \mathit{DIV})) & \end{cases}$$

Lemma 24 *The following hold of LCEP_i:*

1. $\alpha \mathit{LCEP}_i = \alpha B_i$.
2. $\phi \mathit{LCEP}_i = \{(t,R) \mid t\in \mathit{Dom}_i\setminus \mathit{dom}_i \;\wedge\; R\in \mathit{RefSets}_i(t)\}$.

3. $\tau LCEP_i = Dom_i$.

Proof. (1) The proof is immediate.

(2) We consider each of two cases in turn.

Case 1: $b_i \in in\,P$.

Case 1a: $t \in dom_i$. In this case,
$\phi LCEP_i^{in}(t) = \{(\langle a \rangle \circ s, X) \mid a \in Next_i(t) \ \wedge \ (s, X) \in \phi LCEP_i^{in}(t \circ \langle a \rangle)\}$.

Case 1b: $t \in Dom_i \setminus dom_i$. In this case,
$\phi LCEP_i^{in}(t) = \{(\langle a \rangle \circ s, X) \mid a \in Next_i(t) \wedge (s, X) \in \phi LCEP_i^{in}(t \circ \langle a \rangle)\} \cup \bigcup_{R \in RM_t^i} \{(\langle \rangle, R') \mid R' \subseteq R\}$.

Case 2: $b_i \in out\,P$.

Case 2a: $t \in dom_i$. In this case,
$\phi LCEP_i^{out}(t) = \{(\langle a \rangle \circ s, X) \mid a \in Next_i(t) \ \wedge \ (s, X) \in \phi LCEP_i^{out}(t \circ \langle a \rangle)\}$.

Case 2b: $t \in Dom_i \setminus dom_i$. In this case,
$\phi LCEP_i^{out}(t) = \{(\langle a \rangle \circ s, X) \mid a \in Next_i(t) \ \wedge \ (s, X) \in \phi LCEP_i^{out}(t \circ \langle a \rangle)\} \cup$
$\bigcap_{R \in RM_t^i} (\bigcup_{a \in \alpha B_i \setminus R} \{(\langle \rangle, X) \mid a \notin X\})$.

The proof of this part follows from the above by a straightforward induction on the length of traces using the definition of $Next_i$, and lemmas 22 and 23.

(3) We consider each of two cases in turn.

Case 1: $b_i \in in\,P$.

Case 1a: $t \in dom_i$. In this case, we have
$\tau LCEP_i^{in}(t) = \{\langle \rangle\} \cup \{\langle a \rangle \circ s \mid a \in Next_i(t) \ \wedge \ s \in \tau LCEP_i^{in}(t \circ \langle a \rangle)\}$.

Case 1b: $t \in Dom_i \setminus dom_i$. In this case,
$\tau LCEP_i^{in}(t) = \{\langle \rangle\} \cup \{\langle a \rangle \circ s \mid a \in Next_i(t) \ \wedge \ s \in \tau LCEP_i^{in}(t \circ \langle a \rangle)\} \cup \{\langle a \rangle \mid (\exists R \in RM_t^i) \ a \notin R\}$.

Case 2: $b_i \in out\,P$.

Case 2a: $t \in dom_i$. In this case we have
$\tau LCEP_i^{out}(t) = \{\langle \rangle\} \cup \{\langle a \rangle \circ s \mid a \in Next_i(t) \ \wedge \ s \in \tau LCEP_i^{out}(t \circ \langle a \rangle)\}$.

Case 2b: $t \in Dom_i \setminus dom_i$. In this case,
$\tau LCEP_i^{out}(t) = \{\langle \rangle\} \cup \{\langle a \rangle \circ s \mid a \in Next_i(t) \ \wedge \ s \in \tau LCEP_i^{out}(t \circ \langle a \rangle)\} \cup \{\langle a \rangle \mid (\exists R \in RM_t^i) \ a \notin R\}$.

The proof of this part follows from the above by a straightforward induction on the length of traces using the definition of $Next_i$, the definition of RM_t^i and EP4. □

Note in the above result that we have no stable failure with a trace component t such that $t \in dom_i$. This is because condition LC is not interested in what is refused when behaviour over a channel set B_i is complete.

We are now in a position to define the process, $FINALIMPLE_i$, upon which the check for deadlock-freedom will be carried out.

$$FINALIMPLE_i \triangleq \widehat{Q}_i \parallel LCEP_i.$$

Theorem 25. *$\widehat{Q}$ meets condition LC with respect to channel b_i if and only if $FINALIMPLE_i$ is deadlock-free.*

Proof. By lemma 24(2) and lemma 20(2) we have that $\phi FINALIMPLE_i = \{(t, R) \mid (\exists (w, X) \in \phi_\perp \widehat{Q}, Y \in RefSets_i(t)) \ t = w \upharpoonright B_i \wedge \ t \notin dom_i \ \wedge \ R \subseteq (\alpha B_i \cap X) \cup Y\}$. The proof follows immediately from the above and lemma 21. □

Corollary 26 *$\widehat{Q}$ meets condition LC if and only if, for every ep_i such that $b_i \in b_{nid}$, $FINALIMPLE_i$ is deadlock-free.*

The above result allows us to proceed to automatic verification of condition LC.

8.3 Example

We now show how to define the relevant process expressions used to verify that Buf_{imple} and $Send_{imple}$ respectively meet condition LC. Note that, in both cases, we need only check the condition with respect to channel mid. We assume ep_1 is ep_{twice} (it follows that $B_1 = \{data, ack\}$).

Verifying Buf_{imple}: In this case, b_1 is an *input* channel. Assume that $\widehat{Q}$ is Buf_{imple} after the application of preprocessing as described in section 4 and so $\alpha\widehat{Q} = \alpha data \ \cup \ \alpha ack \ \cup \ \alpha out$. Assume that $PROC_1$ is as defined above, where $\alpha B_1 = \alpha data \cup \alpha ack$ and $\alpha\widehat{Q} = \alpha data \ \cup \ \alpha ack \ \cup \ \alpha out$. We then have that $\widehat{Q}_1 \triangleq (\widehat{Q} \parallel PROC_1) \setminus \alpha out$. We then define the tester process $LCEP_1$ for the extraction pattern $ep_1 = ep_{twice}$ in terms of two auxiliary processes $LCEP'_1(x)$ and $LCEP''_1(x)$:

$$\begin{aligned}
LCEP_1 &\triangleq (\Box_{x\in\{0,1\}}(data.x \to LCEP'_1(x)) \ \Box \ DIV \\
LCEP'_1(x) &\triangleq ((ack.yes \to LCEP_1 \ \Box \ ack.no \to LCEP''_1(x)) \ \Box \ DIV) \ \sqcap \\
& \ (\sqcap_{R\in\{\alpha data\}}(\Box_{y\in(\alpha B_1\setminus R)} \ y \to DIV)) \\
LCEP''_1(x) &\triangleq ((data.x \to LCEP_1) \ \Box \ DIV) \ \sqcap \ (\sqcap_{R\in\{\alpha B_1\setminus data.x\}}(\Box_{y\in(\alpha B_1\setminus R)} \ y \to DIV)).
\end{aligned}$$

From these process expressions, we were able to define $FINALIMPLE_1$ for $ep_1 = ep_{twice}$ when b_1 is an input channel and check it for deadlock freedom using FDR2, as a result of which we have checked condition LC for Buf_{imple}.

Verifying $Send_{imple}$: In this case, b_1 is an *output* channel. Assume that $\widehat{Q}$ is $Send_{imple}$ after preprocessing and so $\alpha\widehat{Q} = \alpha data \ \cup \ \alpha ack \ \cup \ \alpha in$. Assume that $PROC_1$ is as defined above, where $\alpha B_1 = \alpha data \cup \alpha ack$ and $\alpha\widehat{Q} = \alpha data \ \cup \ \alpha ack \ \cup \ \alpha in$. We then have that $\widehat{Q}_1 \triangleq (\widehat{Q} \parallel PROC_1) \setminus \alpha in$. We define the tester process $LCEP_1$ for this extraction pattern in terms of two auxiliary processes $LCEP'_1(x)$ and $LCEP''_1(x)$:

$$\begin{aligned}
LCEP_1 &\triangleq (\Box_{x\in\{0,1\}}(data.x \to LCEP'_1(x)) \ \Box \ DIV \\
LCEP'_1(x) &\triangleq ((ack.yes \to LCEP_1 \ \Box \ ack.no \to LCEP''_1(x)) \ \Box \ DIV) \ \sqcap \\
& \ (\Box_{R\in\{\alpha data\}}(\sqcap_{y\in(\alpha B_1\setminus R)} \ y \to DIV)) \\
LCEP''_1(x) &\triangleq ((data.x \to LCEP_1) \ \Box \ DIV) \ \sqcap \ (\Box_{R\in\{\alpha B_1\setminus data.x\}}(\sqcap_{y\in(\alpha B_1\setminus R)} \ y \to DIV)).
\end{aligned}$$

From these process expressions, we were able to define $FINALIMPLE_1$ for $ep_1 = ep_{twice}$ when b_1 is an output channel and check it for deadlock freedom using FDR2, as a result of which we have checked condition LC for $Send_{imple}$.

9 Checking condition RE

Finally we consider how to check condition RE. We assume that, by this stage, all other conditions of the implementation relation have been successfully verified for $\widehat{Q}$. The condition to be met is as follows:

RE If $(t, R) \in \phi_{dom}Q$ then $(extr_{ep\cup ep'}(t), \alpha B \cup (R \cap \alpha b_{id})) \in \phi_{\perp}P$, where $B \subseteq b_{nid}$ is the set of all channels of P blocked at (t, R).

From theorem 7 and the fact that conditions DP and DF have been met, we can deduce that $\delta\widehat{Q} = \emptyset$ and so $\phi\widehat{Q} = \phi_\perp\widehat{Q}$. Since we require $\delta P = \emptyset$, then we also have that $\phi_\perp P = \phi P$. These two facts allow us to check condition RE while working only in the stable failures model: they guarantee that we do not lose any information on the failures of either P or Q by working in that model as opposed to working in the failures divergences model (remember that the implementation relation itself — and so condition RE — is defined in the failures divergences model).

According to this condition, there are three major things which we need to do to the implementation process: we need to extract its traces, identify in some way the channels that are blocked and, finally, preserve refusals on channels which are uninterpreted. As before, we will convert blocking of a channel b_i into a local deadlock on channel set B_i. This time, however, we cannot isolate the refusals on B_i and the resulting deadlock will remain local: i.e. it will be a deadlock across a restricted channel set rather than a deadlock of a process as a whole. Processes similar to the $LCEP_i$ defined in section 8 will be used to convert blocking of channels in P into (local) deadlock across channel sets in $\widehat{Q}$, although there are two main differences. The first is that the new processes defined here incorporate the features necessary to *extract* the traces of $\widehat{Q}$: to do this, they use the approach of section 6 and the processes TE_i.

The second difference is to do with making sure that too much is not refused if a channel b_i is not blocked. The following should explain this a little more. We are transforming the process $\widehat{Q}$ so that it is expressed at the level of abstraction of the specification process P and wish the resulting abstracted process to refine P (in the stable failures model) if and only if $\widehat{Q}$ meets condition RE. Since RE stipulates no requirements for refusals on channels $b_i \in b_{nid}$ if b_i is not blocked in $\widehat{Q}$, we need to make sure that our abstracted implementation process refuses as little as the specification might refuse on b_i after any particular trace. This means that the abstracted implementation process needs to offer all events which are valid extensions according to τP of the extracted trace which brought us to this point (in the abstracted implementation). To do this would be rather difficult and there is a much easier way to proceed, which is taken here.

We first modify the specification process P — yielding $\widehat{P}'$ — using the approach of section 8 of combining DIV with the non-deterministic choice operator to give two types of state, one to give the traces of the new process $\widehat{P}'$ and the other to give the failures. At each state contributing to the failures of the process, we simply need to know whether or not everything is refused on any channel b_i such that $b_i \in b_{nid}$. We therefore rename any event offered on such a b_i to a distinguished event, d_i. (We define $d_I \triangleq \{d_i \mid i \in I\}$.) As a result, rather than offering or refusing events on b_i, the process simply refuses or accepts the event d_i. However, in order for this renaming to d_i to be carried out without interfering with the events which contribute to the traces of the new specification process, the events from αb_i which give the *refusals* of the specification are *primed* and it is these primed events which are renamed.

As a consequence of this, rather than having to ensure that our (modified) implementation process either refuses everything on B_i if b_i is blocked or otherwise offers as much on b_i as is consistent with the traces of the specification, we either refuse the event d_i if b_i is blocked or offer d_i otherwise. This also means that we must prime events in the implementation in the same way as in the specification.

In addition, we make sure that every $t \in \tau P$ can be extended by any event d_i, since otherwise the traces of our constructed implementation process may fail to be contained in those of our constructed specification: this is because if a channel b_i is not blocked in $\widehat{Q}$ then it is ignored by condition RE, while this fact of non-blocking will give rise to the occurrence of an event d_i in the constructed implementation process.

9.1 Preprocessing the specification

We first show how the specification process P must be preprocessed in order that it has the necessary semantic characterisation. To do this, we define two renaming relations which will be used to "prime" events, along with one to rename *primed* events on either αB_i or αb_i such that $b_i \in b_{nid}$ to d_i.

The relations p and *prime* are defined for events a such that $a \in \alpha B_i$ or $a \in \alpha b_i$, where $b_i \in b_{nid}$.

$$prime(a) \triangleq a' \qquad \text{and} \qquad p(a) \triangleq \{a, a'\}.$$

If $a \in prime(\alpha B_i)$ or $a \in prime(\alpha b_i)$, where $b_i \in b_{nid}$:

$$m(a) \triangleq d_i.$$

The act of priming an event cannot be done directly in (machine-readable) CSP and so the approach taken is as follows. We take the event to be primed and define a new channel with the same type as the original and whose name is a concatenation of the original name and some other "reserved" word, such as prime. The new event will then occur on the new channel, whilst communicating the same data value as the old event. For example, if we were to "prime" the event *data*.0, the result could be $data_{prime}.0$ (see section 9.5 for further examples). Note, of course, that we would not be able to use channel names containing subscripts in machine-readable CSP: they are used here simply for the purposes of presentation and, in practice, we would use something like *dataprime*.

The specification process is redefined as $\widehat{P}$ using p:

$$\widehat{P} \triangleq P[p].$$

The process *PROC* is then defined which will be composed in parallel with $\widehat{P}$ in order to separate its states into those which contribute traces and those which contribute failures in the stable failures model.

$$PROC \triangleq ((\Box_{a \in \alpha b_{nid}} a \rightarrow PROC)) \Box DIV) \sqcap (\Box_{a \in prime(\alpha b_{nid})} a \rightarrow DIV)$$

$$\sqcap ((\Box_{y \in d_I} y \rightarrow DIV) \Box DIV).$$

PROC is then composed in parallel with $\widehat{P}$, with the result having the renaming function m applied to it to give the process *NEWSPEC*. This process will be used as a specification process in the refinement check used to check condition RE.

$$\widehat{P}' \triangleq \widehat{P} \parallel PROC \qquad \text{and} \qquad NEWSPEC \triangleq \widehat{P}'[m].$$

For X such that $(t, X) \in \phi_\perp P$ for some t, we define $D(X) \triangleq \{d_i \in d_I \mid \alpha b_i \subseteq X\}$. Note by definition of d_I that if $d_i \in D(X)$ then $b_i \in b_{nid}$.

Lemma 27 *The following hold of NEWSPEC:*

1. $\alpha NEWSPEC = \alpha P \cup d_I$.
2. $\phi NEWSPEC = \{(t, R) \mid (\exists (t, X) \in \phi_\perp P)\ R \subseteq (X \cap \alpha b_{id}) \cup D(X) \cup \alpha b_{nid}\}$.
3. $\tau NEWSPEC = \tau P \cup \{t \circ \langle d_i \rangle \mid t \in \tau P \ \wedge \ d_i \in d_I\}$.

Proof. (1) We have that $\alpha\widehat{P} = \alpha P \cup prime(\alpha b_{nid})$ and $\alpha PROC = \alpha b_{nid} \cup prime(\alpha b_{nid}) \cup d_I$. From this we have that $\alpha\widehat{P}' = \alpha P \cup prime(\alpha b_{nid}) \cup d_I$ and the proof of this part follows from the definition of m.

(2) We first have that
$\phi\widehat{P} = \{(t, R) \mid (\exists\ (w, X) \in \phi P)\ p^{-1}(t) = w \ \wedge \ R \subseteq X \cup prime(X \cap \alpha b_{nid})\}$. We then observe that

- $\phi((\Box_{a \in \alpha b_{nid}} a \rightarrow PROC) \Box DIV) = \{(\langle a \rangle \circ s, R) \mid a \in \alpha b_{nid} \ \wedge \ (s, R) \in \phi PROC\}$.
- $\phi(\Box_{a \in prime(\alpha b_{nid})} a \rightarrow DIV) = \{(\langle\,\rangle, R) \mid R \subseteq \alpha b_{nid} \cup d_I\}$. (In order to determine the events which are *refused* here, we assume that $\alpha(\Box_{a \in prime(\alpha b_{nid})} a \rightarrow DIV) = \alpha PROC$.)
- $\phi((\Box_{y \in d_I} y \rightarrow DIV) \Box DIV) = \emptyset$.

It follows that $\phi PROC = \{(t, R) \mid t \in (\alpha b_{nid})^* \ \wedge \ R \subseteq \alpha b_{nid} \cup d_I\}$. From the above we have that $\phi\widehat{P}' = \{(t, R) \mid (\exists (t, X) \in \phi P) \ R \subseteq (X \cap \alpha b_{id}) \cup prime(X \cap \alpha b_{nid}) \cup \alpha b_{nid} \cup d_I\}$. The proof of this part follows by the definition of m, the semantic definition of the renaming operator (see section A in the appendix and recall that $m^{-1}(d_i) = prime(\alpha B_i) \cup prime(\alpha b_i))$ and the fact that $\phi_\perp P = \phi P$ since $\delta P = \emptyset$.

(3) We first have that $\tau\widehat{P} = \{t \mid p^{-1}(t) \in \tau P\}$. We then observe that

- $\tau((\Box_{a \in \alpha b_{nid}} a \to PROC) \ \Box \ DIV) = \{\langle\rangle\} \cup \{\langle a\rangle \circ s \mid a \in \alpha b_{nid} \ \wedge \ s \in \tau PROC\}$.
- $\tau(\Box_{a \in prime(\alpha b_{nid})} a \to DIV) = \{\langle\rangle\} \cup \{\langle a\rangle \mid a \in prime(\alpha b_{nid})\}$.
- $\tau((\Box_{y \in d_I} y \to DIV) \ \Box \ DIV)) = \{\langle\rangle\} \cup \{\langle a\rangle \mid a \in d_I\}$.

It follows that
$\tau PROC = (\alpha b_{nid})^* \cup \{t \circ \langle a\rangle \mid t \in (\alpha b_{nid})^* \ \wedge \ a \in prime(\alpha b_{nid})\}$
$\qquad \cup \{t \circ \langle d_i\rangle \mid t \in (\alpha b_{nid})^* \ \wedge \ d_i \in d_I\}$.
From the above we have that
$\tau\widehat{P}' = \tau P \cup \{t \circ \langle a\rangle \mid t \circ \langle b\rangle \in \tau P \ \wedge \ b \in \alpha b_{nid} \ \wedge \ a = prime(b)\}$
$\qquad \cup \{t \circ \langle d_i\rangle \mid t \in \tau P \ \wedge \ d_i \in d_I\}$.
The proof of this part follows by the definition of m. □

9.2 Preprocessing the implementation

It is also necessary to preprocess the implementation $\widehat{Q}$, firstly so that the events in $\alpha\widehat{Q}$ are primed as necessary and secondly so that they are renamed — i.e. with *prep* defined in section 6 — in preparation for the syntactic manipulation that allows the extraction function to be encoded. We therefore define:

$$\widehat{Q}' \triangleq \widehat{Q}[p] \quad \text{and} \quad \widehat{Q}'' \triangleq \widehat{Q}'[prep].$$

9.3 Extracting traces and detecting blocking

We now define a process RE_i for each non-identity extraction pattern ep_i, the set of which processes will be combined with $\widehat{Q}''$ — the renaming of the implementation process for the purpose of carrying out the extraction of traces — in order to extract the traces of the implementation and to convert the blocking, after any trace $t \in dom_i$, of any channel $b_i \in b_{nid}$ to a refusal of the event d_i.

In the first case, we give the definition of the process RE_i when $b_i \in in\,P$. In this case, $RE_i = RE_i^{in}(\langle\rangle)$:

$$RE_i^{in}(t) = \begin{cases} (\Box_{x \in \pi_i(t)} x \to RE_i^{in}(t \circ domain(x))) \ \Box \ DIV & \text{if } t \in Dom_i \setminus dom_i \\ ((\Box_{x \in \pi_i(t)} x \to RE_i^{in}(t \circ domain(x))) \ \Box \ DIV) \ \sqcap & \text{if } t \in dom_i \\ (\sqcap_{R \in RM_t^i} (\Box_{x \in prime_i(\alpha B_i \setminus R)} x \to DIV)) & \end{cases}$$

We now give the definition of the process RE_i when $b_i \in out\,P$. In this case, $RE_i = RE_i^{out}(\langle\rangle)$:

$$RE_i^{out}(t) = \begin{cases} (\Box_{x \in \pi_i(t)} x \to RE_i^{out}(t \circ domain(x))) \ \Box \ DIV & \text{if } t \in Dom_i \setminus dom_i \\ ((\Box_{x \in \pi_i(t)} x \to RE_i^{out}(t \circ domain(x))) \ \Box \ DIV) \ \sqcap & \text{if } t \in dom_i \\ (\Box_{R \in RM_t^i} (\sqcap_{x \in prime_i(\alpha B_i \setminus R)} x \to DIV)) & \end{cases}$$

We now define the composition in parallel of all of the processes RE_i such that $b_i \in b_{nid}$:

$$RE_I \triangleq \|_{i \in I} \ RE_i.$$

Lemma 28 *The following hold of RE_I:*

1. $\alpha RE_I = prime(\alpha B_{nid}) \cup prep(\alpha B_{nid})$.
2. $\phi RE_I = \{(t, R) \mid domain(t) \in dom_I \;\wedge\; extract(t) \setminus \alpha B_{nid} = extr_I(domain(t))$
   $\wedge\; ((\forall i \in I)(\exists R' \in RefSets_i(domain(t) \upharpoonright B_i))\; R \cap prime(\alpha B_i) = prime(R')\}$.
3. $\tau RE_I = \{t \mid domain(t) \in Dom_I \;\wedge\; extract(t) \setminus \alpha B_{nid} = extr_I(domain(t))\} \cup$
   $\{t \circ prime(a) \mid (\exists b_i \in b_{nid})\; extract(t) \setminus \alpha B_{nid} = extr_I(domain(t))$
   $\wedge\; a \in \alpha B_i \;\wedge\; domain(t) \upharpoonright B_i \in dom_i \;\wedge\; domain(t) \circ \langle a \rangle \in Dom_I\}$.

Proof. (1) The proof follows from the fact that $\alpha RE_i = prime_i(\alpha B_i) \cup prep(\alpha B_i)$.

(2) We begin by considering each of two cases in turn.

Case 1: $b_i \in in\,P$.

Case 1a: $t \in Dom_i \setminus dom_i$. In this case, we have that

$\phi RE_i^{in}(t) = \{(\langle a \rangle \circ s, X) \mid a \in \pi_i(t) \;\wedge\; (s, X) \in \phi RE_i^{in}(t \circ domain(a))\}$.

Case 1b: $t \in dom_i$. In this case we have that

$\phi RE_i^{in}(t) = \{(\langle a \rangle \circ s, X) \mid a \in \pi_i(t) \;\wedge\; (s, X) \in \phi RE_i^{in}(t \circ domain(a))\} \cup$
$\bigcup_{R \in RM_t^i} \{(\langle\,\rangle, R') \mid R' \subseteq prep(\alpha B_i) \cup prime_i(R)\}$.

Case 2: $b_i \in out\,P$.

Case 2a: $t \in Dom_i \setminus dom_i$. In this case, we have that

$\phi RE_i^{out}(t) = \{(\langle a \rangle \circ s, X) \mid a \in \pi_i(t) \;\wedge\; (s, X) \in \phi RE_i^{out}(t \circ domain(a))\}$.

Case 2b: $t \in dom_i$. In this case, we have that

$\phi RE_i^{out}(t) = \{(\langle a \rangle \circ s, X) \mid a \in \pi_i(t) \;\wedge\; (s, X) \in \; \phi RE_i^{out}(t \circ domain(a))\} \cup$
$\bigcap_{R \in RM_t^i} (\bigcup_{a \in prime_i(\alpha B_i \setminus R)} \{(\langle\,\rangle, X) \mid a \notin X \;\wedge\; X \subseteq prep(\alpha B_i) \cup prime_i(\alpha B_i)\})$.

From a straightforward induction on the length of traces using the above two cases and the definitions of $extract_i$ and π_i, and lemmas 22 and 23, we have that

$\phi RE_i = \{(t, R) \mid domain(t) \in dom_i \;\wedge\; extract(t) \setminus \alpha B_i = extr_i(domain(t))$
$\wedge\; (\exists R' \in RefSets_i(domain(t)))\; R \subseteq prep(\alpha B_i) \cup prime(R')\}$.

The proof of this part follows from the above and EP5, EP6 and EP7.

(3) We begin by considering each of two cases in turn.

Case 1: $b_i \in in\,P$.

Case 1a: $t \in Dom_i \setminus dom_i$. In this case, we have that

$\tau RE_i^{in}(t) = \{\langle\,\rangle\} \cup \{\langle a \rangle \circ s \mid a \in \pi_i(t) \;\wedge\; s \in \tau RE_i^{in}(t \circ domain(a))\}$.

Case 1b: $t \in dom_i$. In this case, we have that

$\tau RE_i^{in}(t) = \{\langle\,\rangle\} \cup \{\langle a \rangle \circ s \mid a \in \pi_i(t) \;\wedge\; s \in \tau RE_i^{in}(t \circ domain(a))\} \cup$
$\{\langle prime_i(a) \rangle \mid (\exists R \in RM_t^i)\; a \notin R\}$.

Case 2: $b_i \in out\,P$.

Case 2a: $t \in Dom_i \setminus dom_i$. In this case we have that

$\tau RE_i^{out}(t) = \{\langle\,\rangle\} \cup \{\langle a \rangle \circ s \mid a \in \pi_i(t) \;\wedge\; s \in \tau RE_i^{out}(t \circ domain(a))\}$.

Case 2b: $t \in dom_i$. In this case we have that

$\tau RE_i^{out}(t) = \{\langle\,\rangle\} \cup \{\langle a \rangle \circ s \mid a \in \pi_i(t) \;\wedge\; s \in \tau RE_i^{out}(t \circ domain(a))\} \cup$
$\{\langle prime_i(a) \rangle \mid \exists R \in RM_t^i . a \notin R\}$.

From a straightforward induction on the length of traces using the above two cases and the definitions of $extract_i$ and π_i, along with the third clause given in cases 1b and 2b respectively and EP4, we have that

$\tau RE_i = \{t \mid domain(t) \in Dom_i \;\wedge\; extract_i(t) \setminus \alpha B_i = extr_i(domain(t))\} \cup$
$\{t \circ prime(a) \mid domain(t) \in dom_i \;\wedge\; extract_i(t) \setminus \alpha B_i = extr_i(domain(t))$
$\wedge\; domain(t) \circ \langle a \rangle \in Dom_i\}$.

The proof of this part follows from the above and EP5, EP6 and EP7. □

We now define the process *FINALIMPLE* which will constitute the implementation process supplied to the refinement check in FDR2 which will check whether or not $\widehat{Q}$ meets condition RE with respect to P. It is defined in terms of an auxiliary process $PREIMPLE$ simply to make clearer the presentation of proofs below and also for the purposes of the readability of the definition itself:

$$PREIMPLE \triangleq ((\widehat{Q}'' \parallel RE_I)[extract]) \setminus \alpha B_{nid}.$$

$$FINALIMPLE \triangleq PREIMPLE[m].$$

We define $block(w, X) \triangleq \{d_i \in d_I \mid b_i \text{ is blocked at } (w, X)\}$. Remember that if $d_i \in d_I$ then $b_i \in b_{nid}$.

Lemma 29 *The following hold of FINALIMPLE:*

1. $\alpha FINALIMPLE = \alpha P \cup d_I$.
2. $\phi FINALIMPLE = \{(t, R) \mid (\exists (w, X) \in \phi_{dom}\widehat{Q})\ extr_{ep\cup ep'}(w) = t$
   $\wedge\ R \subseteq (X \cap \alpha b_{id}) \cup block(w, X) \cup \alpha b_{nid}\}$.
3. $\tau FINALIMPLE = extr_{ep\cup ep'}(\tau\widehat{Q}) \cup$
   $\{t \circ \langle d_i\rangle \mid (\exists w \circ \langle a\rangle \in \tau\widehat{Q}, b_i \in b_{nid})\ extr_{ep\cup ep'}(w) = t$
   $\wedge\ a \in \alpha B_i\ \wedge\ w \upharpoonright B_i \in dom_i\}$.

Proof. (1) We observe that $\alpha\widehat{Q}' = \alpha Q \cup prime(\alpha B_{nid})$, from which we have that $\alpha\widehat{Q}'' = \alpha b_{id} \cup prime(\alpha B_{nid}) \cup prep(\alpha B_{nid})$. From this and lemma 28(1), we have that $\alpha(\widehat{Q}'' \parallel RE_I) = \alpha b_{id} \cup prime(\alpha B_{nid}) \cup prep(\alpha B_{nid})$. From the above and by definition of *extract* we have that $\alpha PREIMPLE = \alpha P \cup prime(\alpha B_{nid})$. The proof of this part follows from the above by definition of m.

(2) We first observe that
$\phi\widehat{Q}' = \{(t, R) \mid (\exists (w, X) \in \phi_\perp\widehat{Q})\ p^{-1}(t) = w\ \wedge\ R \subseteq X \cup prime(X \cap \alpha B_{nid})\}$
(remember that $\phi\widehat{Q} = \phi_\perp\widehat{Q}$ since $\delta\widehat{Q} = \emptyset$). We then have that
$\phi\widehat{Q}'' = \{(t, R) \mid (\exists (w, X) \in \phi_\perp\widehat{Q})\ p^{-1}(domain(t)) = w$
$\wedge\ R \subseteq (X \cap \alpha b_{id}) \cup prime(X \cap \alpha B_{nid}) \cup prep(X \cap \alpha B_{nid})\}$.
From the above, the fact that the extraction mapping is the identity mapping and $dom_i = (\alpha b_i)^*$ if ep_i is such that $b_i \in b_{nid}$, and lemma 28(2), we have that

$$\begin{aligned}\phi(\widehat{Q}'' \parallel RE_I) = \{(t, R) \mid\ & (\exists (w, X) \in \phi_\perp\widehat{Q}, Z \subset \alpha B_{nid})\\ & domain(t) = w \in dom_{ep\cup ep}\\ & \wedge\ extract(t) \setminus \alpha B_{nid} = extr_{ep\cup ep'}(w)\\ & \wedge\ ((\forall i \in I)\ Z \cap \alpha B_i \in RefSets_i(w \upharpoonright B_i))\\ & \wedge\ R \subseteq (X \cap \alpha b_{id}) \cup prime(X \cap \alpha B_{nid}) \cup prep(\alpha B_{nid}) \cup prime(Z)\}.\end{aligned}$$

From the above and by definition of *extract* we have that

$$\begin{aligned}\phi PREIMPLE = \{(t, R) \mid\ & (\exists (w, X) \in \phi_\perp\widehat{Q}, Z \subset \alpha B_{nid})\\ & w \in dom_{ep\cup ep'}\\ & \wedge\ t = extr_{ep\cup ep'}(w)\\ & \wedge\ ((\forall i \in I)\ Z \cap \alpha B_i \in RefSets_i(w \upharpoonright B_i))\\ & \wedge\ R \subseteq (X \cap \alpha b_{id}) \cup prime(X \cap \alpha B_{nid}) \cup \alpha b_{nid} \cup prime(Z)\}.\end{aligned}$$

By lemma 21 and the definition of $prime_i$, we have that, if $b_i \in b_{nid}$, b_i is blocked at (w, X) if and only if there exists $Z \in RefSets_i(w \upharpoonright B_i)$ such that $prime_i(X \cap \alpha B_i) \cup prime_i(Z) = prime_i(\alpha B_i)$. The proof of this part follows from the above, the definition of m and the semantic definition of the renaming operator (see section A in the appendix and recall that $m^{-1}(d_i) = prime(\alpha b_i) \cup prime(\alpha B_i)$).

(3) We first observe that $\tau\widehat{Q}' = \{t \mid p^{-1}(t) \in \tau\widehat{Q}\}$. From this we have that $\tau\widehat{Q}'' = \{t \mid p^{-1}(domain(t)) \in \tau\widehat{Q}\}$. From the above, the fact that the extraction mapping is the identity mapping if ep_i is such that $b_i \in b_{nid}$ and lemma 28(3) we have that

$$\begin{aligned}\tau(\widehat{Q}'' \parallel RE_I) = \ & \{t \mid domain(t) \in \tau\widehat{Q}\ \wedge\ extract(t) \setminus \alpha B_{nid} = extr_{ep\cup ep'}(domain(t))\} \cup\\ & \{t \circ prime(a) \mid (\exists w \circ \langle a\rangle \in \tau\widehat{Q}, b_i \in b_{nid})\ domain(t) = w\\ & \wedge\ extract(t) \setminus \alpha B_{nid} = extr_{ep\cup ep'}(w)\\ & \wedge\ a \in \alpha B_i\ \wedge\ w \upharpoonright B_i \in dom_i\}.\end{aligned}$$

From the above we have that
$\tau PREIMPLE = extr_{ep \cup ep'}(\tau \widehat{Q}) \cup$
$\{t \circ prime(a) \mid (\exists w \circ \langle a \rangle \in \tau \widehat{Q}, b_i \in b_{nid})\ extr_{ep \cup ep'}(w) = t$
$\wedge\ a \in \alpha B_i\ \wedge\ w \upharpoonright B_i \in dom_i\}$.
The proof of this part follows from the above. □

Note from the above result that, if $(t, R) \in \phi FINALIMPLE$, then $t = extr_{ep \cup ep'}(w)$ for some trace w of $\widehat{Q}$ such that $w \in dom_{ep \cup ep'}$. This is because, due to the definition of condition RE, we are not interested in what is refused in $\widehat{Q}$ after traces which are not complete.

We now give the final result which lets us check condition RE using FDR2.

Theorem 30. *$\widehat{Q}$ meets condition RE if and only if $NEWSPEC \sqsubseteq_F FINALIMPLE$.*

Proof. ($\Longrightarrow$) We assume that $\widehat{Q}$ has been shown to meet condition RE and prove that $NEWSPEC \sqsubseteq_F FINALIMPLE$ under this assumption. Since we know that $\widehat{Q}$ meets condition TE, we know that $extr_{ep \cup ep'}(\tau \widehat{Q}) \subseteq \tau P$ and so it follows from lemma 27(3) and lemma 29(3) that $\tau FINALIMPLE \subseteq \tau NEWSPEC$. We simply have to show, therefore, that $\phi FINALIMPLE \subseteq \phi NEWSPEC$.

Let $(t, R) \in \phi FINALIMPLE$. By lemma 29(2), there exists $(w, X) \in \phi_{dom} \widehat{Q}$ such that $t = extr_{ep \cup ep'}(w)$, $R \cap \alpha b_{id} \subseteq X \cap \alpha b_{id}$ and if $d_i \in R$ then $b_i \in b_{nid}$ is blocked at (w, X). Since $\widehat{Q}$ meets condition RE, we know that there exists $(t, Y) \in \phi_\perp P$ such that $Y \cap \alpha b_{id} = X \cap \alpha b_{id}$ — that is, $R \cap \alpha b_{id} \subseteq Y \cap \alpha b_{id}$ — and if $b_i \in b_{nid}$ is blocked at (w, X), then $\alpha b_i \subseteq Y$. By lemma 27(2), there exists $(t, Z) \in \phi NEWSPEC$ such that $R \cap \alpha b_{id} \subseteq Y \cap \alpha b_{id} = Z \cap \alpha b_{id}$, $d_i \in Z$ if $\alpha b_i \subseteq Y$ (that is, if b_i is blocked at (w, X) then $d_i \in Z$) and $\alpha b_{nid} \subseteq Z$. It follows from the above and SF3 (see section A in the appendix) that $(t, R) \in \phi NEWSPEC$.

($\Longleftarrow$) We assume that $NEWSPEC \sqsubseteq_F FINALIMPLE$ and prove that $\widehat{Q}$ meets condition RE under this assumption. Let $(w, X) \in \phi_{dom} \widehat{Q}$. By lemma 29(2) there exists $(t, R) \in \phi FINALIMPLE$ such that $t = extr_{ep \cup ep'}(w)$, $R \cap \alpha b_{id} = X \cap \alpha b_{id}$ and $d_i \in R$ if $b_i \in b_{nid}$ is blocked at (w, X). Since we assume that $\phi FINALIMPLE \subseteq \phi NEWSPEC$, we have that $(t, R) \in \phi NEWSPEC$. It follows from lemma 27(2) that there exists $(t, Y) \in \phi_\perp P$ such that $X \cap \alpha b_{id} = R \cap \alpha b_{id} \subseteq Y \cap \alpha b_{id}$ and if $d_i \in R$ — that is, if $b_i \in b_{nid}$ is blocked at (w, X) — then $\alpha b_i \subseteq Y$. The proof in this direction follows from the above and SF3 (see section A in the appendix). □

The above result allows us to proceed to automatic verification of condition RE.

9.4 Verification in practice

It is possible to construct the process *FINALIMPLE* in such a way in practice that we avoid the state explosion which may arise from first composing in parallel the disjoint processes RE_i to give RE_I and then composing RE_I in parallel with $\widehat{Q}''$ (we use this order of composition above for simplicity of reasoning). By associativity of the parallel composition operator, we have that $\widehat{Q}'' \parallel RE_I = (\ldots((\widehat{Q}'' \parallel RE_1) \parallel RE_2) \parallel \ldots) \parallel RE_{|I|}$ (where processes, channel sets and renamings are subscripted, it is assumed that the subscript is taken from the set I and $|I|$ is used to denote the cardinality of I). Due to the way *extract* is defined in terms of $extract_i$ and since $extract_i$ refers only to events in RE_i, we may push the $extract_i$ components inwards as far as possible over the parallel composition. We may do the same with the hiding of αB_{nid} and the renaming using m for the same reason and thus have the following result:

$$(((\widehat{Q}'' \parallel RE_I)[extract]) \setminus \alpha B_{nid})[m] =$$

$$((((\ldots(((((((\widehat{Q}'' \parallel RE_1)[extract_1]) \setminus \alpha B_1)[m]) \parallel RE_2)[extract_2]) \setminus \alpha B_2)[m]) \parallel \ldots)$$
$$\parallel RE_{|I|})[extract_{|I|}]) \setminus \alpha B_{|I|})[m]$$

This new order of composition of the necessary component processes means that we keep down the size of the intermediate processes constructed by FDR2 as *FINALIMPLE* itself is constructed.

9.5 Example

We now show how the results given above can be used to define inputs to FDR2 to verify automatically that Buf_{imple} (respectively $Send_{imple}$) meets condition RE with respect to Buf_{spec} (respectively $Send_{spec}$).

In both cases, let ep_1 be ep_{twice}. This is the only non-identity extraction pattern and $d_I = \{d_1\}$. Also, $B_{nid} = B_1 = \{data, ack\}$. We define new channels $data_{prime}$, ack_{prime} and mid_{prime} with the same types as $data$,ack and mid respectively on which will occur the primed events. Note that, in the definition of $PROC$ (which is the same whichever of the two processes we are checking here), the deterministic choice operator is indexed by $x \in prime(\alpha mid)$: although, in this syntactic definition, the set of events is given and then primed, in practice we would index the operator directly with $x \in \alpha mid_{prime}$ to give an equivalent effect. The renaming $prime$ is defined as:

$$
\begin{aligned}
prime \triangleq \{&(mid.0, mid_{prime}.0), (mid.1, mid_{prime}.1), (data.0, data_{prime}.0),\\
&(data.1, data_{prime}.1), (ack.yes, ack_{prime}.yes), (ack.no, ack_{prime}.no)\}.
\end{aligned}
$$

The renaming p is defined as:

$$
\begin{aligned}
p \triangleq \{&(mid.0, mid.0), (mid.0, mid_{prime}.0), (mid.1, mid.1),\\
&(mid.1, mid_{prime}.1), (data.0, data.0)(data.0, data_{prime}.0),\\
&(data.1, data.1), (data.1, data_{prime}.1), (ack.yes, ack.yes),\\
&(ack.yes, ack_{prime}.yes), (ack.no, ack.no), (ack.no, ack_{prime}.no)\}.
\end{aligned}
$$

The renaming m is defined as:

$$
\begin{aligned}
m \triangleq \{&(mid_{prime}.0, d_1), (mid_{prime}.1, d_1), (data_{prime}.0, d_1),\\
&(data_{prime}.1, d_1), (ack_{prime}.yes, d_1), (ack_{prime}.no, d_1)\}.
\end{aligned}
$$

Using the detail above, $NEWSPEC$ could be constructed for both Buf_{spec} and $Send_{spec}$.

In the definition of RE_i in section 9.3, a choice operator is indexed by $R \in RM_t^i$ and then a subsequent choice operator is indexed by $x \in prime(\alpha B_i \setminus R)$. In practice, as shown below, we actually index the first operator by $prime(R)$ such that $R \in RM_t^i$ and the subsequent choice operator is indexed by $x \in prime(\alpha B_i) \setminus R$. This is not important as the two ways of proceeding are equivalent: the former is used in definitions for ease of expression whilst the latter is used in practice since, in machine-readable CSP, we cannot actually apply the renaming operator to a set (which is being used to index a choice operator).

Below, let $X \triangleq \alpha ack_{prime} \cup \{data_{prime}.0\}$ and $Y \triangleq \alpha ack_{prime} \cup \{data_{prime}.1\}$.

Verifying Buf_{imple}: In this case, b_1 is an input channel and RE_1 for the extraction pattern $ep_1 = ep_{twice}$ is defined in terms of two auxiliary processes $RE_1'(x)$ and $RE_1''(x)$:

$$
\begin{aligned}
RE_1 \quad &\triangleq ((\Box_{x\in\{0,1\}}\, data.x \rightarrow RE_1'(x)) \,\Box\, DIV) \sqcap\\
&\quad (\sqcap_{R\in\{X,Y\}}(\Box_{y\in((\alpha data_{prime}\cup\alpha ack_{prime})\setminus R)}\ y \rightarrow DIV)).
\end{aligned}
$$

$$RE_1'(x) \triangleq (extract_{ack}.yes.mid.x \rightarrow RE_1 \,\Box\, ack.no \rightarrow RE_1''(x)) \,\Box\, DIV.$$

$$RE_1''(x) \triangleq (extract_{data}.x.mid.x \rightarrow RE_1) \,\Box\, DIV.$$

Verifying $Send_{imple}$: In this case, b_1 is an output channel and RE_1 for the extraction pattern $ep_1 = ep_{twice}$ is defined in terms of the following two auxiliary processes $RE_1'(x)$ and $RE_1''(x)$:

$$RE_1 \triangleq ((\Box_{x\in\{0,1\}} data.x \to RE'_1(x)) \Box DIV) \sqcap (\Box_{R\in\{X,Y\}} (\sqcap_{y\in((\alpha data_{prime} \cup \alpha ack_{prime})\setminus R)} y \to DIV)).$$

$$RE'_1(x) \triangleq (extract_{ack}.yes.mid.x \to RE_1 \Box ack.no \to RE''_1(x)) \Box DIV.$$

$$RE''_1(x) \triangleq (extract_{data}.x.mid.x \to RE_1 \Box DIV).$$

Using the components defined above and the renamings *extract* and *prep* as described in section 6.2, we were able to define the necessary process expressions. By supplying them as inputs to FDR2, we were then able to verify automatically that Buf_{imple} (respectively $Send_{imple}$) meets condition RE with respect to Buf_{spec} (respectively $Send_{spec}$).

10 Concluding remarks

We have developed a verification method that allows for automatic compositional verification of networks of CSP processes in the event that corresponding specification and implementation processes have different interfaces. Most importantly, we have built that method of verification on top of an existing industrial strength tool, with all the benefits which that accrues.

Although the approach may seem somewhat unwieldy if a user is required to provide explicitly the various components used to produce the inputs to FDR2, it is relatively straightforward to automate production of these inputs from the input which the user must *necessarily* supply. In other words, the user supplies the original specification and implementation components, the necessary extraction pattern representations (not incorporating refusal bound information), the refusal bounds as separate sets labelled with the state at which they are applied and perhaps an explicit statement of the alphabets of the various B_i; from these the necessary inputs to FDR2 can be generated automatically. This can be done by simple processsing of text files, since the inputs to FDR2 are text files. As an example, to check condition LC, if we are given a process encoding Dom_i, along with refusal bound information, we may effectively lay the template provided by process $LCEP_i$ over the top of this process encoding Dom_i and substitute for any RM^i_t in (the syntactic definition of) $LCEP_i$ the relevant set of sets denoting the refusal bounds.

The method presented in this report is currently being used to verify the correctness of asynchronous communication mechanisms (ACMs) (see, for example, [16]) and, in particular, is being used to help derive abstract specifications for such mechanisms. Such specifications are useful in reasoning about systems implemented using ACMs but traditionally only exist in vague forms. Since the method presented here is compositional, we can use it to verify the correctness of a particular ACM independently of any context in which it is to be placed, while still basing our notion of correctness on the fact that the meaning of a process is its meaning when placed in context. This is especially relevant when, as is the case with ACMs, none of the behaviour of the component will be directly visible once it has been placed in a system and all internal communication hidden.

Acknowledgements This work was supported by EPSRC studentship no. 99306462. Thanks are also due to Maciej Koutny for reading numerous drafts of this report and suggesting many useful improvements.

References

1. E. Brinksma: A Theory for the Derivation of Tests. In: *Protocol Specification, Verification and Testing, VIII*, S. Aggarwal and K. Sabnani(Eds.). North-Holland (1988) 63–74.
2. S. D. Brookes, C. A. R. Hoare and A. W. Roscoe: A Theory of Communicating Sequential Process. *Journal of ACM* 31 (1984) 560–599.

3. S. D. Brookes and A. W. Roscoe: An Improved Failures Model for Communicating Sequential Processes. Proc. of *Seminar on Concurrency*, S. D. Brookes, A. W. Roscoe and G. Winskel (Eds.). Springer-Verlag, Lecture Notes in Computer Science 197 (1985) 281–305.
4. J. Burton, M. Koutny and G. Pappalardo: Verifying Implementation Relations in the Event of Interface Difference. Proc. of *FME 2001: Formal Methods for Increasing Software Productivity*, J. N. Oliveira and P. Zave (Eds.). LNCS 2021 (2001) 364–383.
5. J. Burton, M. Koutny and G. Pappalardo: Implementing Communicating Processes in the Event of Interface Difference. Proc. of *ACSD 2001: Second International Conference on Application of Concurrency to System Design*, A. Valmari and A. Yakovlev (Eds.). IEEE Computer Society (2001) 87–96.
6. J. Burton, M. Koutny and G. Pappalardo: Compositional Verification of a Network of CSP Processes. Technical Report 757, Dept. of Computing Science, University of Newcastle upon Tyne (2002).
7. *FDR2 User Manual* : Available at: *http://www.formal.demon.co.uk/fdr2manual/fdr2manual_toc.html*
8. P. Collette and C. B. Jones: Enhancing the Tractability of Rely/Guarantee Specifications in the Development of Interfering Operations. Technical Report CUMCS-95-10-3, Department of Computing Science, Manchester University (1995).
9. M. C. B. Hennessy: *Algebraic Theory of Processes.* MIT Press (1988).
10. C. A. R. Hoare: *Communicating Sequential Processes.* Prentice Hall (1985).
11. M. Koutny, L. Mancini and G. Pappalardo: Two Implementation Relations and the Correctness of Communicated Replicated Processing. *Formal Aspects of Computing* 9 (1997) 119–148.
12. R. Milner: *Communication and Concurrency.* Prentice Hall (1989).
13. A. W. Roscoe: Model-Checking CSP. In: *A Classical Mind, Essays in Honour of C.A.R. Hoare.* Prentice-Hall (1994) 353–378.
14. A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson and J. B. Scattergood: Hierarchical Compression for Model-checking CSP, *or* How to Check 10^{20} Dining Philosophers for Deadlock. Proc. of *Workshop on Tools and Algorithms for The Construction and Analysis of Systems, TACAS*, U. H. Engberg, K. G. Larsen and A. Skou (Eds.). BRICS Notes Series NS-95-2 (1995) 187–200.
15. A. W. Roscoe: *The Theory and Practice of Concurrency.* Prentice-Hall (1998).
16. H. R. Simpson: Four-slot Fully Asynchronous Communication Mechanism. *IEE Proceedings* 137, Pt E(1) (January 1990) 17–30.

A Operator definitions

We first show how the alphabet of a process may be derived from the alphabets of its components.

$$
\begin{aligned}
\alpha(a \to P) &= \{a\} \cup \alpha P.\\
\alpha(P \;\Box\; Q) &= \alpha P \cup \alpha Q.\\
\alpha(P \;\sqcap\; Q) &= \alpha P \cup \alpha Q.\\
\alpha(P \parallel Q) &= \alpha P \cup \alpha Q.\\
\alpha(P \setminus A) &= \alpha P \setminus A.\\
\alpha(P[R]) &= R(\alpha P).
\end{aligned}
$$

The traces model In order for a set of execution sequences to be considered a valid process in the traces model, they must meet a consistency condition:

T1 τP is non-empty and prefix-closed.

The semantics of a process term in the traces model may be derived recursively from the semantics of its components and the semantic definition accorded to the various operators used to combine those components.

$$\begin{array}{lcl}
\tau(a \to P) & = & \{\langle\rangle\} \cup \{\langle a\rangle \circ s \mid s \in \tau P\}. \\
\tau(P \;\Box\; Q) & = & \tau P \cup \tau Q. \\
\tau(P \;\sqcap\; Q) & = & \tau P \cup \tau Q. \\
\tau(P \parallel Q) & = & \{t \mid t \upharpoonright \alpha P \in \tau P \;\wedge\; t \upharpoonright \alpha Q \in \tau Q\}. \\
\tau(P \setminus A) & = & \{t \setminus A \mid t \in \tau P\}. \\
\tau(P[R]) & = & \{t \mid (\exists u \in \tau P)\; uR^{*}t\}.
\end{array}$$

The stable failures model The set of failures and traces representing a process in the stable failures model have to meet the following consistency conditions:

SF1 τP is non-empty and prefix-closed.
SF2 $(t, R) \in \phi P \Rightarrow t \in \tau P$.
SF3 $(t, R) \in \phi P \;\wedge\; S \subseteq R \Rightarrow (t, S) \in \phi P$.
SF4 $(t, R) \in \phi P \;\wedge\; t \circ \langle a\rangle \notin \tau P \Rightarrow (t, R \cup \{a\}) \in \phi P$.

The necessary semantic definitions of the various operators in the stable failures model are as follows (note that the effect of the various operators on the trace component of the model is as described for the traces model above).

$$\begin{array}{lcl}
\phi(a \to P) & = & \{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a\rangle \circ s, X) \mid (s, X) \in \phi P\}. \\
\phi(P \;\sqcap\; Q) & = & \phi P \cup \phi Q. \\
\phi(P \;\Box\; Q) & = & \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \phi P \cap \phi Q\} \cup \{(s, X) \mid (s, X) \in \phi P \cup \phi Q \;\wedge\; s \neq \langle\rangle\}. \\
\phi(P \parallel Q) & = & \{(u, Y \cup Z) \mid (\exists s, t)\; (s, Y) \in \phi P \;\wedge\; (t, Z) \in \phi Q \;\wedge\; s = u \upharpoonright \alpha P \;\wedge\; t = u \upharpoonright \alpha Q\}. \\
\phi(P \setminus X) & = & \{(t \setminus X, Y) \mid (t, Y \cup X) \in \phi P\}. \\
\phi(P[R]) & = & \{(s', X) \mid (\exists s)\; sR^{*}s' \;\wedge\; (s, R^{-1}(X)) \in \phi P\}.
\end{array}$$

The failures divergences model The set of failures and divergences representing a process in this model have to meet the following consistency conditions:

FD1 $\tau_{\perp} P$ is non-empty and prefix-closed.
FD2 $(t, R) \in \phi_{\perp} P \;\wedge\; S \subseteq R \Rightarrow (t, S) \in \phi_{\perp} P$.
FD3 $(t, R) \in \phi_{\perp} P \;\wedge\; t \circ \langle a\rangle \notin \tau_{\perp} P \Rightarrow (t, R \cup \{a\}) \in \phi_{\perp} P$.
FD4 $s \in \delta P \;\wedge\; t \in (\alpha P)^{*} \Rightarrow s \circ t \in \delta P$.
FD5 $t \in \delta P \Rightarrow (t, R) \in \phi_{\perp} P$.

The definitions of the operators in the failures divergences model are as follows:

$$\begin{array}{lcl}
\phi_{\perp}(a \to P) & = & \{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a\rangle \circ s, X) \mid (s, X) \in \phi_{\perp} P\}. \\
\delta(a \to P) & = & \{\langle a\rangle \circ s \mid s \in \delta P\}. \\
\phi_{\perp}(P \;\sqcap\; Q) & = & \phi_{\perp} P \cup \phi_{\perp} Q. \\
\delta(P \;\sqcap\; Q) & = & \delta P \cup \delta Q. \\
\phi_{\perp}(P \;\Box\; Q) & = & \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \phi_{\perp} P \cap \phi_{\perp} Q\} \cup \{(s, X) \mid (s, X) \in \phi_{\perp} P \cup \phi_{\perp} Q \;\wedge\; s \neq \langle\rangle\} \\
 & & \cup \{(\langle\rangle, X) \mid \langle\rangle \in \delta P \cup \delta Q\}. \\
\delta(P \;\Box\; Q) & = & \delta P \cup \delta Q. \\
\phi_{\perp}(P \parallel Q) & = & \{(u, Y \cup Z) \mid (\exists s, t)(s, Y) \in \phi_{\perp} P \;\wedge\; (t, Z) \in \phi_{\perp} Q \;\wedge\; s = u \upharpoonright \alpha P \;\wedge\; t = u \upharpoonright \alpha Q\} \\
 & & \cup \{(u, Y) \mid u \in \delta(P \parallel Q)\}. \\
\delta(P \parallel Q) & = & \{u \circ v \mid (\exists s \in \tau_{\perp} P, t \in \tau_{\perp} Q)\; u \upharpoonright \alpha P = s \;\wedge\; u \upharpoonright \alpha Q = t \;\wedge\; (s \in \delta P \;\vee\; t \in \delta Q)\}. \\
\phi_{\perp}(P \setminus X) & = & \{(s \setminus X, Y) \mid (s, Y \cup X) \in \phi_{\perp} P\} \cup \{(s, X) \mid s \in \delta(P \setminus X)\}. \\
\delta(P \setminus X) & = & \{(s \setminus X) \circ t \mid s \in \delta P\} \cup \{(u \setminus X) \circ t \mid u \in (\alpha P)^{\omega} \;\wedge\; (u \setminus X) \text{ is finite} \\
 & & \wedge\; (\forall s < u)\;\; s \in \tau_{\perp} P\}. \\
\phi_{\perp}(P[R]) & = & \{(s', X) \mid (\exists s)\; sR^{*}s' \;\wedge\; (s, R^{-1}(X)) \in \phi_{\perp} P\} \cup \{(s, X) \mid s \in \delta P[R]\}. \\
\delta(P[R]) & = & \{s' \circ t \mid (\exists s \in \delta P)\; sR^{*}s'\}.
\end{array}$$