Stepwise Refinement of Processes

Steve Reeves [2]

Department of Computer Science, University of Waikato, Hamilton, New Zealand

David Streader [1]

Department of Computer Science, University of Waikato, Hamilton, New Zealand

Abstract

Industry is looking to create a market in reliable "plug-and-play" components. To model components in a modular style it would be useful to combine event-based and state-based reasoning. One of the first steps in building an event-based model is to decide upon a set of atomic actions. This choice will depend on the formalism used, and may restrict in quite unexpected ways what we are able to formalise. In this paper we illustrate some limits to developing real world processes using existing formalisms, and we define a new notion of refinement, *vertical refinement*, which addresses some of these limitations. We show that using vertical refinement we can rewrite a specification into a different formalism, allowing us to move between handshake processes, broadcast processes and abstract data types.

Key words: components, process, vertical refinement

1 Introduction

Industry is looking to create a market in reliable "plug-and-play" components. It has been noted [25] that to model components in a modular style it would be useful to combine the event-based reasoning of process algebra with state-based reasoning. But it has been commented [24] that in order for process algebras to become of greater use in practice there is a need for a more well-defined methodology. Here we will take a familiar state-based methodology and apply it to a specification of an event-based process. This can be seen as an improved, or at least novel, event-based methodology or as the first step towards a methodology for specifying components with both state-based and event-based features.

This paper is event-based other than that we will be applying *use case* specifications, a common part of state-based methodologies, to event-based processes.

[1] Email: dstr@cs.waikato.ac.nz
[2] Email: stever@cs.waikato.ac.nz

We will show a very simple example of an event-based process where the atomic actions we have chosen are perfectly adequate. But, when we attempt to extend the implementation by specifying an additional use case the specification cannot be satisfied using process algebras such as CSP or CCS. From this we conclude that the actions we have chosen are not at an adequate level of abstraction. It is usual to have to informally rewrite the specification using a different set of atomic actions. Here we define *vertical refinement* of processes that allows us to formally rewrite the specification using a different set of atomic actions.

Anyone building a process model must decide on the formalism to be used and on a set of atomic actions. It is tempting to think that the atomic actions used in the formalism correspond quite naturally to real "atomic" actions. In practice it is more likely that some thought is required to decide what aspects of the real world can be modelled by an atomic action. To make such a choice requires a detailed knowledge both about the world to be modelled and the formalism in which it is going to be modelled. Moreover the choice will depend on the formalism used, and this may restrict in quite unexpected ways what we are able to formalise.

In the paper we will illustrate some limits to developing real world processes using existing formalisms, and we define vertical refinement which addresses some of these limitations.

Many definitions of process refinement, e.g. failure refinement [3] and trace refinement [23], relate processes with the same alphabet or set of atomic actions. We will refer to processes with the same alphabet as being in the same *layer* and refinement within a layer as *horizontal refinement*. Our definition of vertical refinement between layers is based on a refinement function and an abstraction function. The refinement function $[\![\_]\!]$ maps high-level processes, defined over a set of high-level atomic actions, to low-level, more detailed processes, defined over a set of low-level atomic actions. The abstraction mapping vA moves us from the low-level back to the high-level.

Each refinement step formalises a design decision, e.g. a failure (trace) refinement step is a design decision to remove some failures (traces) from the set of failures (traces) of the process. A sequence of refinement steps is *well behaved* when no design decisions can undo a previous decision, e.g. after a failure (trace) is removed from the set of failures (traces) of the process no subsequent refinement step can replace it.

To the best of our knowledge our definition of vertical refinement is the first that can be used to relate layers where the actions on different layers can be of a different kind, e.g. handshake, broadcast or abstract data types.

In this paper we will use two sets of observable actions: $\mathtt{a}!$ actions that can be thought of as active and $\mathtt{a}?$ actions that can be thought of as passive; and we will model their synchronisation as the unobservable τ. In the handshake layer we use these two sets of actions in the same way that CCS uses names and co-names.

The example that follows demonstrates that it is possible to use a set of handshake actions to formally describe how a simple process must interact, but to model the process we need to extend process algebras such as CSP or CCS.

1.1 Example - Use case specification

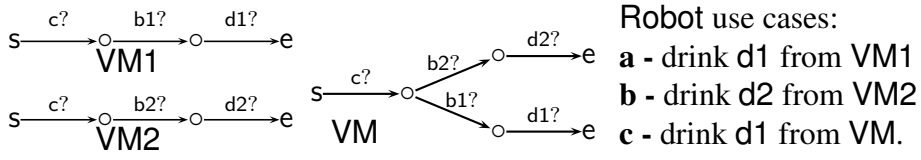
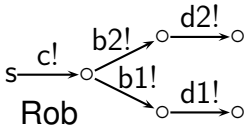


Fig. 1. Robot specification

In Fig. 1 we give three simple use cases to specify what drinks Robot obtains from three vending machines. VM is easy to understand as a machine that accepts a coin (c?) and then reacts to either button one (b1?) or button two (b2?) being pushed and subsequently enables the removal of drink one (d1?) or drink two (d2?). The vending machines VM1 and VM2 are self-explanatory. Using the chosen atomic actions process algebras can model these simple vending machines but can they model the specified Robot ?

Use case **a**, the robot must obtain drink d1 from VM1, can obviously be satisfied by R1 $\stackrel{\text{def}}{=}$ c!b1!d1!. But this fails to satisfy use case **b**.



We can refine R1 to Rob and establish that it satisfies use case **b**, that the robot must obtain drink d2 from VM2. Although for such a simple process we can see that it also satisfies the initial use case "obtaining drink d1 from VM1", for more complex processes we would establish a formal refinement relation from R1 to Rob.

At this point in the development VM1, VM2 and Rob are all viewed as being defined at an adequate level of abstraction and the first two use cases have been satisfied. But we are unable to satisfy the third use case **c**, the robot must obtain drink d1 from VM. From this we conclude that the actions we have chosen are not at an adequate level of abstraction.

Using existing formalisms we would have to start again with a new formalisation that is only informally related to the original formal specification. For our simple example, not much work would be lost. However, for large processes, changing the formal specification could entail a huge amount of work. Moreover, as we see from Fig. 1, there is no simple way to tell that our specification is in any way unsatisfactory.

The same problem also occurs with feature addition. Imagine that we implement Rob to obtain the required drinks from VM1 and VM2 with our high-level atomic actions. Later, however, we are required to add a new feature to the robot, namely to obtain drink d1 from VM. Normally the formal specification is thrown away and a new specification based on a new set of "atomic actions" is written. We, instead, propose to formally refine the specification by applying a vertical refinement.

In Section 5 we give a solution to the example that works if we interpret the low-level processes as broadcast processes, i.e. processes with local control of output, rather than handshake processes.

2 Framework for a Single layer

Let $Names$ be a finite set of names, $Act^! \stackrel{\text{def}}{=} \{\mathsf{a}!|\mathsf{a} \in Names\}$ be a set of *active* actions, $Act^? \stackrel{\text{def}}{=} \{\mathsf{a}?|\mathsf{a} \in Names\}$ be a set of *passive* actions, $Act \stackrel{\text{def}}{=} Act^! \cup Act^?$ the set of observable actions and $Act^\tau \stackrel{\text{def}}{=} Act \cup \{\tau\}$. Our handshake formalism, like CCS, splits observable actions into two sets and like CCS the only use we will make of this distinction, in our handshake layer, is to define point-to-point synchronisation. Note, following what Hoare and He say [17][p.198] "The main distinguishing feature of CSP is to define a hiding operator that succeeds in total concealment of internal actions", our observational semantics totally conceals internal actions.

Definition 1 *LTS—labelled transition systems. Let N_A be a finite set of nodes and s_A the start node. Labelled transition system* $\mathsf{A} \stackrel{\text{def}}{=} (N_\mathsf{A}, s_\mathsf{A}, T_\mathsf{A})$ *where* $s_\mathsf{A} \in N_\mathsf{A}$, *and* $T_\mathsf{A} \subseteq \{(n, a, m)|n, m \in N_\mathsf{A} \wedge a \in Act^\tau\}$. •

A *path* is a sequence of states and actions and the set of paths generated by the LTS A is: $Path_\mathsf{A} \stackrel{\text{def}}{=} \{s_\mathsf{A}, \rho_1^\alpha, n_2, \rho_2^\alpha, \ldots | (n_1, \rho_1^\alpha, n_2), (n_2, \rho_2^\alpha, n_3), \ldots \in T_\mathsf{A}\}$. We write $|\rho|$ for the number of actions in (i.e. length of) a path and ρ^α for the sequence of actions $\rho_1^\alpha, \rho_2^\alpha \ldots$ in path $\rho = s_\mathsf{A}, \rho_1^\alpha, n_2, \rho_2^\alpha \ldots$. For finite paths $\rho = s_\mathsf{A}, \rho_1^\alpha, n_2, \rho_2^\alpha, \ldots n_i$ define $last(\rho) \stackrel{\text{def}}{=} n_i$.

We will write ϵ for the empty sequence of actions, hence $\{s_\mathsf{A}\}^\alpha = \epsilon$.

We write $x \xrightarrow{\mathsf{a}} y$ for $(x, \mathsf{a}, y) \in T_\mathsf{A}$ where A is obvious from context, $n \xrightarrow{\mathsf{a}}$ for $\exists m.(n, \mathsf{a}, m) \in T_\mathsf{A}$, $s_\mathsf{A} \xrightarrow{\rho^\alpha}$ when $\rho \in Path_\mathsf{A}$ and $s_\mathsf{A} \xrightarrow{\rho^\alpha} n$ when $\rho \in Path_\mathsf{A} \wedge last(\rho) = n$. Also, $\alpha(\mathsf{A}) \stackrel{\text{def}}{=} \{\mathsf{a}|n \xrightarrow{\mathsf{a}} m \in T_\mathsf{A}\}$, $\pi(s) \stackrel{\text{def}}{=} \{\mathsf{a}|s \xrightarrow{\mathsf{a}}\}$

The complete traces of A are:

$Tr^c(\mathsf{A}) \stackrel{\text{def}}{=} \{\rho^\alpha|(s_\mathsf{A} \xrightarrow{\rho^\alpha} n \wedge \pi(n) = \emptyset) \vee (s_\mathsf{A} \xrightarrow{\rho^\alpha} \wedge |\rho| = \infty)\}$.

A process diverges when it engages in an infinite sequence of τ actions. Divergence has been treated in at least three distinct ways in the literature. Divergence as chaos in [3], chaos free divergence in [6] and the fair interpretation found in [9,18]. We believe our approach would work with any of these interpretations as long as divergence is defined in the same way on both handshake and broadcast processes. We have chosen a fair interpretation of divergence as this can be found in both broadcast [23] and handshake [9,18] semantics.

We use *strong fairness*: a path is fair if, whenever something can occur infinitely often it does occur infinitely often. Thus if a process is offered the ability to perform $s \xrightarrow{\mathsf{b}} \circ$ infinitely often then the action must ultimately be taken. The fair traces of A are:

$Tr^f(\mathsf{A}) \stackrel{\text{def}}{=} \{\rho^\alpha|\rho \in Path(\mathsf{A}) \wedge$
$\forall \mathsf{a} \forall n.(|\{i|n_i \xrightarrow{\mathsf{a}} \wedge n_i = n\}| = \infty \Rightarrow |\{i|n_i \xrightarrow{\mathsf{a}} n_{i+1} \wedge n_i = n\}| = \infty)\}$.

The complete fair traces of A are: $Tr^{cf}(\mathsf{A}) \stackrel{\text{def}}{=} Tr^f(\mathsf{A}) \cap Tr^c(\mathsf{A})$

Our definition of parallel composition models point-to-point synchronisation and is closer to CCS parallel composition than CSP parallel composition. We avoid

CSP-style parallel composition as its ability to model multi-way synchronisation would force us to use a more complicated definition of vertical refinement.

Definition 2 *Parallel composition of LTS* A *and* B*: let* $N \subseteq Names$.

$N_{\mathsf{A}\|_N\mathsf{B}} \stackrel{\text{def}}{=} N_{\mathsf{A}} \times N_{\mathsf{B}}$, $s_{\mathsf{A}\|_N\mathsf{B}} = (s_{\mathsf{A}}, s_{\mathsf{B}})$ *and* $T_{\mathsf{A}\|_N\mathsf{B}}$ *is defined as follows.*

Let $\mathsf{x} \in Act^\tau$ *and* $name(\mathsf{a?}) \stackrel{\text{def}}{=} \mathsf{a}, name(\mathsf{a!}) \stackrel{\text{def}}{=} \mathsf{a}$ *and* $name(\tau) \stackrel{\text{def}}{=} \tau$

$$\frac{n \xrightarrow{\mathsf{x}}_{\mathsf{A}} l, name(\mathsf{x}) \notin N}{(n,m) \xrightarrow{\mathsf{x}}_{\mathsf{A}\|_N\mathsf{B}} (l,m)} \qquad \frac{n \xrightarrow{\mathsf{x}}_{\mathsf{B}} l, name(\mathsf{x}) \notin N}{(m,n) \xrightarrow{\mathsf{x}}_{\mathsf{A}\|_N\mathsf{B}} (m,l)} \qquad \frac{n \xrightarrow{\mathsf{a?}}_{\mathsf{A}} l, m \xrightarrow{\mathsf{a!}}_{\mathsf{B}} k, \mathsf{a} \in N}{(n,m) \xrightarrow{\tau}_{\mathsf{A}\|_N\mathsf{B}} (l,k)}$$

We will write $\|$ *as short for* $\|_{Act}$*. Note this parallel composition does not allow any actions to be concurrent, all must be synchronised.* •

We define refusals: $Ref(\rho, \mathsf{C}) \stackrel{\text{def}}{=} \{\{\mathsf{a} | n \xrightarrow{\mathsf{a}} \not\rightarrow\} | s_{\mathsf{C}} \xrightarrow{\rho} n\}$ and failure refinement [3]: $\mathsf{A} \sqsubseteq_F \mathsf{C} \stackrel{\text{def}}{=} \forall \rho . Ref(\rho, \mathsf{C}) \subseteq Ref(\rho, \mathsf{A})$ where ρ is a sequence of actions and C and A are processes.

The LTS in Definition 1 takes no account of τ actions being unobservable, so we would call it a **strong semantics** ($\rightarrow$) and based on it we have a **strong equivalence** ($=_X$) and a **strong refinement** ($\sqsubseteq_X$) .

We will define, in Section 2.1, a strong semantics of actions and then, in Section 2.2, quite separately give a meaning to unobservable τ actions by defining how to abstract these actions to build an observational semantics. This "separation of concerns" is more common in operational models [18,2] than denotational models [3]. The reason we do this is that we want the observational semantics to be the same on both layers of handshake processes and layers of broadcast processes.

2.1 Refinement, meaning and strong semantics

We should think of the LTS semantics of a process as defining some underlying machinery on which strong equality and strong refinement are built. A single LTS can be used to mean different processes and the different meanings can be formalised using different equalities. By taking the meaning of a specification to be the set of implementations that it can be refined into we can give specifications different meanings by applying different refinements.

Let LTS be the set of all LTSs. Our definition of refinement is parameterised by:

(i) the set of contexts we can use, $\Xi \subseteq \{(\_ \parallel x) | x \in LTS\}$

(ii) $Obs : LTS \rightarrow 2^{Ob}$ a function from LTS to a set of observations. Ob is the set containing any observation of any process.

Definition 3 $\mathsf{A} \sqsubseteq_{(\Xi, Obs)} \mathsf{C} \stackrel{\text{def}}{=} \forall [\_]_x \in \Xi . Obs([\mathsf{C}]_x) \subseteq Obs([\mathsf{A}]_x)$ •

This definition is derived from the generalised testing semantics in [22].

By the explicit use of contexts this definition of refinement can be applied to different **kinds of things**. Contexts for handshake process are $\{(\_ \parallel x) | x \in LTS\}$ [1,22] , contexts for abstract data types are traces [7,22] $\{(\_ \parallel x) | x \in (Act^\tau)^*\}$ and

contexts for broadcast have local control of output [23].

From any definition of refinement we have an equality:

$$\mathsf{A} =_Y \mathsf{C} \stackrel{\text{def}}{=} \mathsf{A} \sqsubseteq_Y \mathsf{C} \wedge \mathsf{C} \sqsubseteq_Y \mathsf{A}$$

As the empty trace is considered an observation the empty set of observations is not in the range of Obs. Hence when $Obs([\mathsf{A}]_x)$ is a singleton set of observations then so is $Obs([\mathsf{C}]_x)$, i.e. our refinement preserves uniqueness of observation.

2.2 *Abstraction*

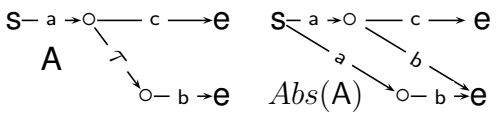
Our definition of observational semantics is quite separate from the definition of strong equality/refinement. This allows us to use the same observational semantics on a layer of handshake processes and a layer of broadcast processes.

Definition 4 *Observational semantics* $\Longrightarrow$*:*

$$s \stackrel{\tau}{\Longrightarrow} t \stackrel{\text{def}}{=} s \stackrel{\tau}{\longrightarrow} s_1, s_1 \stackrel{\tau}{\longrightarrow} s_2, \ldots s_{n-1} \stackrel{\tau}{\longrightarrow} t$$

$$n \stackrel{\mathsf{a}}{\Longrightarrow} m \stackrel{\text{def}}{=} n \stackrel{\tau}{\Longrightarrow} n', n' \stackrel{\mathsf{a}}{\longrightarrow} m', m' \stackrel{\tau}{\Longrightarrow} m \wedge \mathsf{a} \in Act$$

$$Abs(\mathsf{A}) \stackrel{\text{def}}{=} \langle N_{\mathsf{A}}, s_{\mathsf{A}}, \{n \stackrel{x}{\longrightarrow} m | n \stackrel{x}{\Longrightarrow} m\} \rangle. \quad \bullet$$



Our observational semantics is not the same as in CCS [18] as we, like CSP, use failure semantics and thus $Abs(\_)$ removes all τ actions (see example to the right).

Definition 4, or an equivalent definition, has appeared in [9,6,20], and see [20] for a comparison with the literature and a discussion about stability.

Our definition of abstraction, like the definitions in CCS and ACP [2], formalises a "fairness" assumption, i.e. τ loops that can be exited must be exited after a finite number of times around the loop.

From the definition of an **observational semantics** ($\Rightarrow$) we have defined an abstraction function Abs which we now use to define an **observational refinement** $\sqsubseteq_{aX}$ from a strong refinement $\sqsubseteq_X$:

$$\mathsf{A} \sqsubseteq_{aX} \mathsf{C} \stackrel{\text{def}}{=} Abs(\mathsf{A}) \sqsubseteq_X Abs(\mathsf{C})$$

An **observational equivalence** $=_{aX}$ can be defined in the obvious way, the point being that $\sqsubseteq_X$ could be failure refinement $\sqsubseteq_F$ or a trace refinement $\sqsubseteq_{Tr}$.

2.3 *Layers*

Both things and contexts are modelled using LTSs. A layer consists of a set of things, a set of contexts and a refinement relation.

Definition 5 *A layer* X *is* $(\mathsf{T}_\mathsf{X}, \Xi_\mathsf{X}, \sqsubseteq_\mathsf{X})$ *where* T_X *is a set of LTSs used to represent* **things** *,* Ξ_X *a set* **contexts** *and* $\sqsubseteq_\mathsf{X} \subseteq \mathsf{T}_\mathsf{X} \times \mathsf{T}_\mathsf{X}$ *is a refinement relation on things.*

The things represented in the layer are equivalence classes of $=_\mathsf{X}$. Where not confusing we will misuse terminology and refer to an individual LTS as one of the things in the layer. Importantly, different layers can represent different **kinds of things** (Section 2.1).

3 Vertical refinement

We use a function $[\![\_]\!]$ to embed, or *interpret*, high-level LTS as low-level LTS. But not all the low-level LTS are in the range of $[\![\_]\!]$. We use a function vA to embed, or *interpret*, low-level LTS as high-level LTS.

We apply $[\![\_]\!]$ to LTS representations of both high-level things $\mathsf{T_H}$ and contexts Ξ_H. Similarly vA can be applied to LTS representations of both low-level things $\mathsf{T_L}$ and contexts Ξ_L.

Let $\mathsf{P_H} \in \mathsf{T_H}$ be the high-level things, and $\mathsf{X_H} \in \Xi_\mathsf{H}$ be the high-level contexts. Similarly let $\mathsf{P_L} \in \mathsf{T_L}$ be the low-level things, and $\mathsf{X_L} \in \Xi_\mathsf{L}$ be the low-level contexts.

The term $[\mathsf{A_H}]_{\mathsf{X_H}}$ models the interaction of the high-level process $\mathsf{A_H}$ with its high-level context $\mathsf{X_H}$ and this can be seen as an abstract **specification** of the desired behaviour. A more concrete specification (or implementation) is the low-level behaviour. These interactions are represented by $[[\![\mathsf{A_H}]\!]]_{[\![\mathsf{X_H}]\!]}$. Next we discuss what properties we want interpretations $[\![\_]\!]$ and vA to have in order that they constitute a vertical refinement.

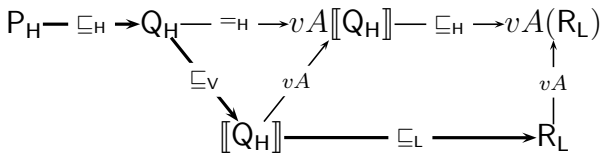


Fig. 2. Stepwise Refinement

Vertical refinement $\sqsubseteq_\mathsf{V}$ may be preceded by some high-level refinement steps and may itself precede low-level refinement steps (Fig. 2). Ideally we would require this sequence of refinements to be well behaved but whether a design decision at one layer is preserved by a process at another layer is a matter of *interpretation*. Consequently we call this sequence of refinements well behaved if: refinements within a layer are well behaved and both vertical refinements and low-level refinements can be interpreted as high-level refinements (see Fig. 2).

Definition 6 *Functions* $[\![\_]\!]$ *and* vA *define a* **vertical refinement** *between a high-level layer* $(\mathsf{T_H}, \Xi_\mathsf{H}, \sqsubseteq_\mathsf{H})$ *and a low-level layer* $(\mathsf{T_L}, \Xi_\mathsf{L}, \sqsubseteq_\mathsf{L})$ *when*

$\mathsf{P_H} \sqsubseteq_H \mathsf{Q_H} \sqsubseteq_V [\![\mathsf{Q_H}]\!] \sqsubseteq_L \mathsf{R_L}$ *implies* $\mathsf{P_H} \sqsubseteq_H \mathsf{Q_H} =_H vA[\![\mathsf{Q_H}]\!] \sqsubseteq_H vA(\mathsf{R_L})$ •

In practice we use LTS to represent things and thus will have no problem applying $[\![\_]\!]$ a function from LTS to LTS. As the things at any layer are actually equivalence classes of LTSs it would be desirable that $[\![\_]\!]$ was a monotonic functions and hence could be lifted to a function between equivalence classes, i.e. between things. The vertical refinement we will apply in the stepwise development of our example specification will use $[\![\_]\!]$ that is not monotonic. Although monotonisity may be regained by restricting handshake processes to the constructable processes of [21] we leave this to future work.

In our example the high-level process R1 is refined into Rob that satisfies the first two conditions, i.e. that in context VM1 drink d1 is obtained and in context

VM2 drink d2 is obtained. We will construct a vertical refinement $[\![\mathsf{Rob}]\!]$ that can be refined into $\mathsf{Robot_L}$ such that $\mathsf{Robot_L}$ satisfies the third and final part of the specification, i.e. in context VM obtain drink d1.

Because $[\mathsf{Rob}]_{VM1}$ obtains drink d1 and $[\mathsf{Rob}]_{VM2}$ drink d2 then when $[\![\_]\!]$ and vA are a vertical refinement and $[\![\mathsf{Rob}]\!] \sqsubseteq_{\mathsf{L}} \mathsf{Robot_L}$ we are able to conclude that $[vA(\mathsf{Robot_L})]_{\mathsf{VM1}}$ obtains drink d1 and $[vA(\mathsf{Robot_L})]_{\mathsf{VM2}}$ drink d2.

4 Individual layers

Before we can define a vertical refinement between a handshake and a broadcast layer we must define the individual layers using the definitions in Section 2.1 and Section 2.3. In this section we define our layers and to give some confidence that the resulting refinement preorders are reasonable we show that the handshake preorder is the fair failures of [8,19] and the broadcast preorder is very similar to the *quiescence* preorder of Segala [23].

4.1 Handshake layer $(\mathsf{T_{Hs}}, \Xi_{\mathsf{Hs}}, \sqsubseteq_{\mathsf{Hs}})$

The handshake layer allows any LTS to be a thing $\mathsf{T_{Hs}} \stackrel{\text{def}}{=} LTS$ and contexts to be $\Xi_{\mathsf{Hs}} \stackrel{\text{def}}{=} \{\_ \parallel x | x \in \mathsf{T_{Hs}}\}$. We define $\sqsubseteq_{\mathsf{Hs}} \stackrel{\text{def}}{=} \sqsubseteq_{(\Xi_{\mathsf{Hs}}, Tr^{cf})}$. For terminating processes $\sqsubseteq_{\mathsf{Hs}}$ is failure refinement (see [22] for details).

Because, in our definition of refinement, we allow fair traces to be observed the refinement of cyclic processes is not that of CSP, but is the same as the definition in [8,19] where further details of this refinement can be found.

Assuming fair tests and only the special action ω observable we have the *should* tests of [8] that characterise $\sqsubseteq_{\mathbf{should}}$ refinement.

Definition 7 P **should** $\mathsf{T} \stackrel{\text{def}}{=} \forall \rho \in Act^*.\mathsf{P} \parallel \mathsf{T} \xrightarrow{\rho} \mathsf{Q} \implies \exists \mu \in Act^*.\mathsf{Q} \xrightarrow{\mu\omega}$

$\mathsf{A} \sqsubseteq_{\mathbf{should}} \mathsf{C} \stackrel{\text{def}}{=} \forall \mathsf{T}\ \mathsf{A}$ **should** $\mathsf{T} \Rightarrow \mathsf{C}$ **should** T. *From [8]*

As we might expect (and as Lemma 1 shows), using fair tests and all actions observable still characterises the same preorder. We show that $\sqsubseteq_{(\Xi_{\mathsf{Hs}}, Tr^{cf})}$ refinement is the same preorder as $\sqsubseteq_{\mathbf{should}}$, the refinement of [8] and thus the pre-congruence results in [8] apply to our refinement.

Lemma 1 $\mathsf{A} \sqsubseteq_{(\Xi_{Hs}, Tr^{cf})} \mathsf{C} \iff \mathsf{A} \sqsubseteq_{\mathbf{should}} \mathsf{C}$

4.2 Broadcast layer $(\mathsf{T_{BC}}, \Xi_{\mathsf{BC}}, \sqsubseteq_{\mathsf{BC}})$

There has long been interest in the relation between handshake and broadcast style communication but there are many variations of both styles. A comparison of the "point to point" handshake communication of CCS with the multi-way broadcast of CBS can be found in [15]. But the handshake and broadcast styles also differ in that broadcast has **local control of output** whereas with handshake-style communication all actions can be blocked. The only difference between our handshake and broadcast models will be that broadcast cannot be blocked by any context.

Here we relate handshake and broadcast models that both use point to point communication. We believe we could have considered handshake and broadcast models that both use multi-way synchronisation (CSP, ACP and CBS) but here we choose to keep to the simpler point to point model.

We require broadcast processes to have all input actions enabled from all states, this being a common way to model local control of output [23,14].

Definition 8
$$\mathsf{T}_{\mathsf{BC}} \stackrel{\text{def}}{=} \{\mathsf{A} | \forall n \in N_{.\mathsf{A}} \forall \mathsf{a}? \in Act^{?} n \xrightarrow{\mathsf{a}?}\}$$
$$\Xi_{\mathsf{BC}} \stackrel{\text{def}}{=} \{\_ \parallel x | x \in \mathsf{T}_{\mathsf{BC}}\}.$$

The finite traces in our semantics $Tr^{?}$ are the usual quiescent traces [23], i.e. finite traces that stop in a state that can only listen, and the infinite traces in $Tr^{?}$ are fair.

$$Tr^{?}(\mathsf{A}) \stackrel{\text{def}}{=} \{\rho^{\alpha} | (s_{\mathsf{A}} \xrightarrow{\rho^{\alpha}} n \wedge \pi(n) \subseteq Act^{?}) \vee (s_{\mathsf{A}} \xrightarrow{\rho^{\alpha}} \wedge |\rho| = \infty)\} \cap Tr^{f}(\mathsf{A}).$$

These traces can be used as the set of observations in our definition of refinement (Definition 3) and directly as a denotational semantics.

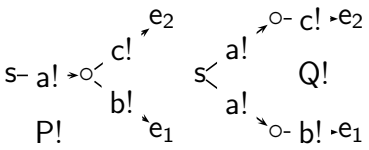
Definition 9 *Broadcast semantics:*
$$\mathsf{A} \sqsubseteq_{Tr^{?}} \mathsf{C} \stackrel{\text{def}}{=} Tr^{?}(\mathsf{C}) \subseteq Tr^{?}(\mathsf{A}) \quad \bullet$$

Semantics that, in the simple setting that we consider, are very similar to Definition 9 appear in [23,14]. But whereas Segala [23] gives meaning to τ actions in his definition of fair trace we use Abs so that the same definition can be applied to both handshake and broadcast processes. Consequently our semantics are slightly different to that in [23].

Lemma 2 $\mathsf{P} =_{Tr^{?}} \mathsf{Q}$ *implies* $\mathsf{P} =_{aTr^{?}} \mathsf{Q}$

It is frequently clearer to not show all the listening loops. Such LTS can be interpreted as broadcast processes by assuming listening loops to be implicit. Function M_{BC} turns a process into a broadcast process by simply adding $n \xrightarrow{\mathsf{a}?} n$ to any n for which $\mathsf{a}?$ is not enabled.

$$M_{BC}(\mathsf{A}) \stackrel{\text{def}}{=} (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}} \cup \{n \xrightarrow{\mathsf{a}?} n | \neg n \xrightarrow{\mathsf{a}?}\})$$



The effect of M_{BC} on the semantics can be best understood by considering some examples. Process $\mathsf{P}!$ (see right) would be deterministic in the handshake world, i.e. a context can "choose" if $\mathsf{P}!$ performs $\mathsf{b}!$ or $\mathsf{c}!$. But applying M_{BC} to any context in which $\mathsf{P}!$ can be placed prevents the contexts from blocking $\mathsf{b}!$ or $\mathsf{c}!$ thus making $\mathsf{P}!$ nondeterministic. Later we will find that the nondeterminism of the choice between output, as seen in $\mathsf{P}!$, is essential for our definition of refinement to satisfy the specification in Section 1.1.

Finally we establish that our general definition of refinement when applied to broadcast processes generates the same preorder as our denotational semantics.

Lemma 3 $\mathsf{A} \sqsubseteq_{(\Xi_{BC}, Tr^{cf})} \mathsf{C} \iff \mathsf{A} \sqsubseteq_{Tr^{?}} \mathsf{C}$

5 Building on a broadcast layer

Having defined our handshake and broadcast layers in Section 4 we can now define a vertical refinement between them.

We map an active high-level action such as b! (see Fig. 3) into three parts, first performing the try action try_b! and subsequently either aborting (rej_b?) if the context cannot synchronise on b or succeeding (acc_b?) if the context can synchronise on b. The mapping for the passive action b? can be seen in right-hand side of Fig. 3.

Our function $[\![\_]\!]$ from a high-level layer to a low-level layer will not only map a! and a? actions to different processes but also add try/reject loops wherever an action cannot be performed (see left-hand side of Fig. 3).

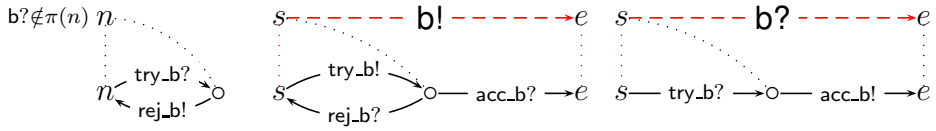


Fig. 3. Vertical refinement$[\![_]\!]$

Although we see this as the natural solution, because of the addition of the try_b?, rej_b! loops at all nodes n for which b? $\notin \pi(n)$, the refinement $[\![\_]\!]$ mapping is neither an action refinement [4] nor indeed an instance of Rensink and Gorrieri's Vertical Implementation [5].

We need some care in interpreting the actions of Fig. 3. In particular both handshake actions b! and b? are able to be blocked but the broadcast actions try_b!,rej_b! and acc_b! are not.

Definition 10 *Let* A $\stackrel{\text{def}}{=} (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$

$$ {}_H[\![\mathsf{A}]\!]_B \stackrel{\text{def}}{=} M_{BC}(N_{{}_H[\![\mathsf{A}]\!]_B}, s_{\mathsf{A}}, e_{\mathsf{A}}, T_{{}_H[\![\mathsf{A}]\!]_B}) $$

$$ N_{{}_H[\![\mathsf{A}]\!]_B} \stackrel{\text{def}}{=} N_{\mathsf{A}} \cup \{n_t | t \in T_{\mathsf{A}}\} \cup \{n_{(m,\mathsf{a?})} | m \in N_{\mathsf{A}} \wedge m \stackrel{\mathsf{a?}}{\not\longrightarrow}\} $$

$$ \begin{aligned} T_{{}_H[\![\mathsf{A}]\!]_B} \stackrel{\text{def}}{=} & \{s \xrightarrow{\mathsf{try\_}x!} z, z \xrightarrow{\mathsf{rej\_}x?} s, z \xrightarrow{\mathsf{acc\_}x?} t | s \xrightarrow{x!} t \wedge z = n_{s \xrightarrow{x!} t}\} \cup \\ & \{s \xrightarrow{\mathsf{try\_}x?} z, z \xrightarrow{\mathsf{acc\_}x!} t | s \xrightarrow{x?} t \wedge z = n_{s \xrightarrow{x?} t}\} \cup \\ & \{s \xrightarrow{\mathsf{try\_}x?} z, z \xrightarrow{\mathsf{rej\_}x!} s | s \stackrel{x?}{\not\longrightarrow} \wedge z = n_{(s,x?)}\} \end{aligned} $$ •

The processes $(N_{{}_H[\![\mathsf{A}]\!]_B}, s_{\mathsf{A}}, T_{{}_H[\![\mathsf{A}]\!]_B})$ are not all valid broadcast processes, i.e. $\not\subseteq \mathsf{T}_B$. For this reason we have applied M_{BC}. For ease of understanding we have not shown the actions added by M_{BC} in Fig. 3.

Next we define vertical abstraction ${}_B Abs_H$. It should be noted that each try_x? action is replaced by two τ actions, one each way.

Definition 11 *Let* A $\stackrel{\text{def}}{=} (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$

$$ (\mathsf{A})\,{}_B Abs_H \stackrel{\text{def}}{=} Abs(N_{\mathsf{A}}, s_{\mathsf{A}}, T_{(\mathsf{A})\,{}_B Abs_H}) $$

$$ \begin{aligned} T_{(\mathsf{A})\,{}_B Abs_H} \stackrel{\text{def}}{=} & \{s \xrightarrow{x!} t | s \xrightarrow{\mathsf{acc\_}x?} t\} \cup \{s \xrightarrow{x?} t | s \xrightarrow{\mathsf{acc\_}x!} t\} \cup \\ & \{s \xrightarrow{\tau} t | s \xrightarrow{\mathsf{try\_}x!} t \vee s \xrightarrow{\mathsf{rej\_}x!} t \vee s \xrightarrow{\mathsf{rej\_}x?} t \vee s \xrightarrow{\tau} t \vee s \xrightarrow{\mathsf{try\_}x?} t \vee t \xrightarrow{\mathsf{try\_}x?} s\} \end{aligned} $$ •

5.1 *Handshake on Broadcast*

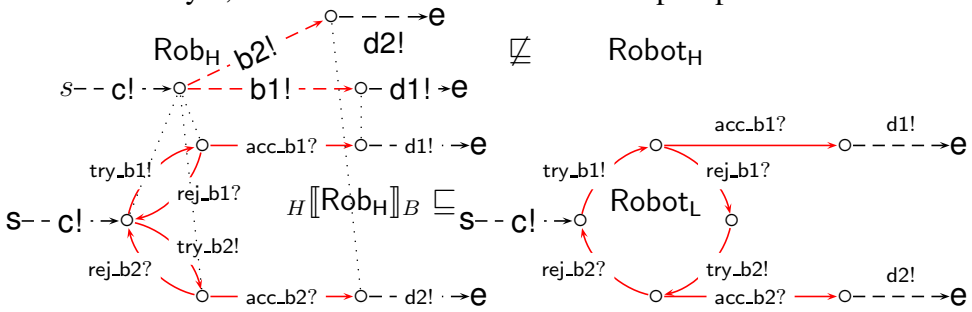
The refinement for our high-level handshake layer is failure subset and on the low-level broadcast layer it is trace subset.

Theorem 1 *Functions* ${}_B Abs_H$ *and* ${}_H[\![\_]\!]_B$ *define a vertical refinement from the handshake layer with* $\sqsubseteq_{(\Xi_{HS}, Tr^{cf})}$ *to the broadcast layer with* $\sqsubseteq_{(\Xi_{BC}, Tr^{cf})}$.

Theorem 1 shows that it is reasonable to model certain components of the broadcast layer as "atomic actions" on the handshake layer. The set of high-level contexts Ξ_H are mapped into ${}_H[\![\Xi_H]\!]_B$. As ${}_H[\![\Xi_H]\!]_B \subset \Xi_B$ the low-level refinement permits a greater set of contexts.

5.2 *Continuing the Robot example from Section 1.1*

Having shown in Theorem 1 that functions from Definition 10 and Definition 11 constitute a vertical refinement between a high-level handshake layer and a low-level broadcast layer, we now use it to refine our example specification.



So as to keep the lower level diagrams small we have only expanded the high-level actions b1! and b2!. The expansion of the other actions is obvious from Fig. 3.

Fig. 4. ${}_H[\![\mathsf{Rob_H}]\!]_B$ can be refined into $\mathsf{Robot_L}$ but there is no $\mathsf{Robot_H}$

${}_H[\![\mathsf{Rob_H}]\!]_B$ in Fig. 4 is a nondeterministic broadcast process. In particular which button, b1 or b2, it tries to push first is not determined. Hence when offered both buttons by VM its behaviour is nondeterministic. Process $\mathsf{Robot_L}$ is a refinement of ${}_H[\![\mathsf{Rob_H}]\!]_B$ that will always try button b1 before b2. Hence when offered both buttons b1 and b2 it will always push b1.

Broadcast processes can be viewed as handshake processes that happen to have input always enabled. But if we treat them as handshake processes we must apply handshake refinement. Viewed as a handshake process ${}_H[\![\mathsf{Rob_H}]\!]_B$ in Fig. 4 is deterministic and cannot be refined, not even into $\mathsf{Robot_L}$.

Starting with "the robot must obtain drink d1 from VM1" (see Fig. 1) we build $\mathsf{R1} \stackrel{\text{def}}{=} \mathsf{c!}, \mathsf{b1!}, \mathsf{d1!}$. In order to satisfy "the robot must obtain drink d2 from VM2" this high-level process is refined into Rob. Because we cannot satisfy "the robot must obtain drink d1 from VM" we apply the functions from Definition 10 and Definition 11 that perform a vertical refinement (Theorem 1), transforming the high-level processes into low-level processes. We then perform a low-level refinement that satisfies the third and final part of the specification seen in Fig. 4.

Although for finite high-level processes we have been using failure refinement it is not true that Rob is a failure refinement of R1. But this is not a problem as Rob is a refinement of R1 using LOTOS's extension [10], *conf* [11] and $\sqsubseteq_{F\delta}$ "weak sub-typing" of [16]. As it is well known that failure refinement implies all of these refinements, see [22] for details, then vertical refinement must preserve these refinements also.

Our refinement steps were $\mathsf{R1} \sqsubseteq_{conf} \mathsf{Rob} \sqsubseteq_V {}_H[\![\mathsf{Rob_H}]\!]_B \sqsubseteq_{(\Xi_{BC}, Tr^{cf})} \mathsf{Robot_L}$. Because $\sqsubseteq_V$ is a vertical refinement we know that $\mathsf{Rob} \sqsubseteq_{(\Xi_{Hs}, Tr^{cf})} {}_B Abs_H(\mathsf{Robot_L})$ and as for finite processes $\sqsubseteq_{(\Xi_{Hs}, Tr^{cf})}$ is $\sqsubseteq_F$ and as it is well known that $\sqsubseteq_F \Rightarrow \sqsubseteq_{conf}$ we have that $\mathsf{R1} \sqsubseteq_{conf} {}_B Abs_H(\mathsf{Robot_L})$ and that $\mathsf{Robot_L}$ satisfies all three of the specified properties.

6 Conclusion

In this work we have applied a general framework similar to that of [22] to fair nonterminating processes. We show that our definition of refinement for handshake communication is essentially the same as those in [8,19]. In addition our definition of refinement for broadcast communication is very similar to those in [23].

The two contributions of this work are:

(i) A definition of vertical refinement between processes with atomic actions that may be of a distinct kind, e.g. handshake or broadcast;

(ii) The stepwise refinement of some simple specifications that, to the best of our knowledge, were previously unsatisfiable in existing formal methods.

6.1 Comparison of vertical refinement with the literature

Our vertical refinement is clearly related to *non-atomic refinement* in Z and Object Z [13,12], *action refinement*, i.e. the replacing of a high-level action by a low-level process, and *vertical implementation* [4,5] . Non-atomic refinement in Z and Object Z is defined as a constrained form of action refinement, in particular high-level actions can only be replaced by a sequence of low-level actions. For an interesting survey of action refinement see [4].

It has been powerfully argued that action refinement is overly restrictive as a method of top-down design (see [4, Ch 7]). One solution to the restrictions of action refinement is vertical implementation [5,4] which like action refinement uses a function from actions to processes.

Our vertical refinement is based not on a function between high-level actions and low-level processes but on a relations between high-level processes and low-level processes. Further, vertical refinement allows the individual layers to have distinct refinement relations and this means that different layers can model different kinds of actions, e.g. handshake, broadcast or even abstract data types.

7 Appendix

Proofs for Section 4.1 Handshake Fair Failure

Lemma 4 $\mathsf{A} \sqsubseteq_{(\Xi_{Hs}, Tr^{cf})} \mathsf{C} \iff \mathsf{A} \sqsubseteq_{should} \mathsf{C}$

Proof We use an intermediate definition: P **fmust** $\mathsf{T} \stackrel{\text{def}}{=} Tr^{cf}([\mathsf{P}]_{\mathsf{T}}) \subseteq Tr^{cf}(\mathsf{T})$

$\mathsf{A} \sqsubseteq_{fmust} \mathsf{C} \stackrel{\text{def}}{=} \forall \mathsf{T}\ \mathsf{A}$ **fmust** $\mathsf{T} \Rightarrow \mathsf{C}$ **fmust** T

Step 1. $\mathsf{A} \sqsubseteq_{should} \mathsf{C} \iff \mathsf{A} \sqsubseteq_{fmust} \mathsf{C}$

$\mathsf{A} \sqsubseteq_{should} \mathsf{C} \stackrel{\text{def}}{=} \forall \mathsf{T}\ \mathsf{A}$ **should** $\mathsf{T} \Rightarrow \mathsf{C}$ **should** T. But not all tests T are needed. Clearly from definition pruning actions after a ω will not affect the success or failure of test. Hence let $\omega \notin \alpha(\mathsf{T})$ and $\mathsf{A} \sqsubseteq_{should} \mathsf{C} \iff \forall \mathsf{T}\ \mathsf{A}$ **should** $\mathsf{T}; \omega \Rightarrow \mathsf{C}$ **should** $\mathsf{T}; \omega$.

Part 1. If P **should** $\mathsf{T}; \omega$ then $\forall \rho \in Act^*.[\mathsf{P}]_{\mathsf{T};\omega} \xrightarrow{\rho} \mathsf{Q} \Rightarrow \exists \mu \in Act^*.\mathsf{Q} \xrightarrow{\mu\omega}$. Let $\rho \in Tr^{cf}$ and $\mu = \epsilon$ gives: $\forall \rho \in Tr^{cf}([\mathsf{P}]_{\mathsf{T}}).[\mathsf{P}]_{\mathsf{T};\omega} \xrightarrow{\rho_0} \mathsf{Q}_1, \mathsf{Q}_1 \xrightarrow{\rho_1} \mathsf{Q}_2, \ldots, \mathsf{Q}_i \xrightarrow{\omega}$ from which we have P **fmust** T.

Part 2. If P **fairmust** T then $Tr^{cf}([\mathsf{P}]_{\mathsf{T}}) \subseteq Tr^{cf}(\mathsf{T})$ and
if $\rho \in prefix(Tr^{cf}(\mathsf{P} \parallel \mathsf{T}; \omega))$ then $\exists \mu \in Act^*.\rho\mu\omega \in Tr^{cf}(\mathsf{T}; \omega)$ and
hence P **should** $\mathsf{T}; \omega$.

From Part 1 and Part 2 Step 1 must follow.

Step 2 $\mathsf{A} \sqsubseteq_{fmust} \mathsf{C} \iff \mathsf{A} \sqsubseteq_{(\Xi_{Hs}, Tr^{cf})} \mathsf{C}$

$\mathsf{A} \sqsubseteq_{fmust} \mathsf{C}$ defined as $\forall \mathsf{T}.Tr^{cf}([\mathsf{A}]_{\mathsf{T}}) \subseteq Tr^{cf}(\mathsf{T}) \Rightarrow Tr^{cf}([\mathsf{C}]_{\mathsf{T}}) \subseteq Tr^{cf}(\mathsf{T})$
$\mathsf{A} \sqsubseteq_{(\Xi_{Hs}, Tr^{cf})} \mathsf{C}$ defined as $\forall \mathsf{T}.Tr^{cf}([\mathsf{C}]_{\mathsf{T}}) \subseteq Tr^{cf}([\mathsf{A}]_{\mathsf{T}})$

Hence clearly $\mathsf{A} \sqsubseteq_{(\Xi_{Hs}, Tr^{cf})} \mathsf{C} \Rightarrow \mathsf{A} \sqsubseteq_{fmust} \mathsf{C}$ 1

Assume $\mathsf{A} \sqsubseteq_{fmust} \mathsf{C}$
As is usual in testing semantics construct T_ρ such that $Tr^{cf}(\mathsf{T}_\rho) = Act^* - \{\rho\}$
$Tr^{cf}([\mathsf{A}]_{\mathsf{T}_\rho}) \subseteq Tr^{cf}(\mathsf{T}_\rho) \Rightarrow Tr^{cf}([\mathsf{C}]_{\mathsf{T}_\rho}) \subseteq Tr^{cf}(\mathsf{T}_\rho)$
$\rho \notin Tr^{cf}([\mathsf{A}]_{\mathsf{T}_\rho}) \Rightarrow \rho \notin Tr^{cf}([\mathsf{C}]_{\mathsf{T}_\rho})$ and then $Tr^{cf}([\mathsf{C}]_{\mathsf{T}_\rho}) \subseteq Tr^{cf}([\mathsf{A}]_{\mathsf{T}_\rho})$
then $\mathsf{A} \sqsubseteq_{(\Xi_{Hs}, Tr^{cf})} \mathsf{C}$ and from assumption:
$\mathsf{A} \sqsubseteq_{(\Xi_{Hs}, Tr^{cf})} \mathsf{C} \Leftarrow \mathsf{A} \sqsubseteq_{fmust} \mathsf{C}$ 2

From 1 and 2 we prove Step 2. From Step 1 and Step 2 we prove our result. •

Proofs for Section 4.2 Broadcast semantics

Lemma 5 $\mathsf{P} =_{Tr^?} \mathsf{Q}$ *implies* $\mathsf{P} =_{oTr^?} \mathsf{Q}$

Proof Assume $\mathsf{P} =_{Tr^?} \mathsf{Q}$ and $\rho \in Tr^?(Abs_o(\mathsf{P}))$. From Abs there exists $\mu \in Tr(\mathsf{P})$ such that μ is an interleaving of ρ and a number of τ actions. Because ρ is fair either a finite number of τ actions are used in the interleaving or P has reached a state where all branches are τ loops. In either case μ is fair, $\mu \in Tr^?(\mathsf{P})$. From $\mathsf{P} =_{Tr^?} \mathsf{Q}$ we have $\mu \in Tr^?(\mathsf{Q})$ and by a similar argument we have $\rho \in Tr^?(Abs_o(\mathsf{Q}))$. Hence $Tr^?(Abs_o(\mathsf{P})) \subseteq Tr^?(Abs_o(\mathsf{Q}))$ and the equality hold by a symmetric argument. Finally from definition $\mathsf{P} =_{oTr^?} \mathsf{Q}$. •

Lemma 6 $\mathsf{A} \sqsubseteq_{(\Xi_{BC}, Tr^{cf})} \mathsf{C} \iff \mathsf{A} \sqsubseteq_{(\Xi_{BC}, Tr^?)} \mathsf{C} \iff \mathsf{A} \sqsubseteq_{Tr^?} \mathsf{C}$

Proof From Definition 3 the first $\Longleftrightarrow$ reduces to: $\forall [\_]_x \in \Xi_{BC}.Tr^{cf}([\mathsf{C}]_x) \subseteq Tr^{cf}([\mathsf{A}]_x) \Leftrightarrow Tr^{?}([\mathsf{C}]_x) \subseteq Tr^{?}([\mathsf{A}]_x)$. As Tr^{cf} can be constructed from $Tr^{?}$ by prefixing each trace by any sequence of inputs the result follows.

From Definition 9 $\mathsf{A} \sqsubseteq_{Tr^{?}} \mathsf{C} \stackrel{\text{def}}{=} Tr^{?}(\mathsf{C}) \subseteq Tr^{?}(\mathsf{A})$ and result follows from congruence w.r.t. $\|$. •

Proofs for Section 5.1: handshake on broadcast is a vertical refinement

Theorem 2 *Functions* ${}_{B}Abs_{H}$ *and* ${}_{H}[\![\_]\!]_{B}$ *define a vertical refinement from the high level* $(\mathsf{T_{Hs}}, \Xi_{Hs}, \sqsubseteq_{a(\Xi_{HS}, Tr^{cf})})$ *to the low level* $(\mathsf{T_{BC}}, \Xi_{BC}, \sqsubseteq_{a(\Xi_{BC}, Tr^{cf})})$.

Proof Monotonicity: $\mathsf{P_L} \sqsubseteq_{aL} \mathsf{Q_L} \Rightarrow vA(\mathsf{P_L}) \sqsubseteq_{aH} vA(\mathsf{Q_L})$

Assume $\mathsf{P_L} \sqsubseteq_{aL} \mathsf{Q_L}$

$\forall x \in \Xi_{\mathsf{L}} Tr^{cf}(Abs([\mathsf{Q_L}]_x)) \subseteq Tr^{cf}(Abs([\mathsf{P_L}]_x))$ *Definition 3*

$\forall x \in \Xi_{\mathsf{L}} Tr^{cf} \circ vA([\mathsf{Q_L}]_x) \subseteq Tr^{cf} \circ vA([\mathsf{P_L}]_x)$ From vA

$\forall x \in \Xi_{\mathsf{L}} Tr^{cf}(Abs([vA(\mathsf{Q_L})]_{vA(x)})) \subseteq Tr^{cf}(Abs([vA(\mathsf{P_L})]_{vA(x)}))$ vA is *distributive*

$\forall y \in \Xi_{\mathsf{H}} Tr^{cf}(Abs([vA(\mathsf{Q_L})]_y)) \subseteq Tr^{cf}(Abs([vA(\mathsf{P_L})]_y))$ vA is *surjective*

$vA(\mathsf{P_L}) \sqsubseteq_{aH} vA(\mathsf{Q_L})$.

From Fig. 2 we can see that we only need to prove $\mathsf{P_H} =_{\mathsf{aH}} vA([\![\mathsf{P_H}]\!])$. All try_x!, try_x?, rej_x? and rej_x! actions added by $[\![]\!]$ will be turned into τ loops by vA that can be collapsed into a single state by $=_{\mathsf{H}}$. This just leaves the renaming x! into acc_x? plus x? into acc_x! and back again. •

References

[1] M. Hennessy, Algebraic Theory of Processes, MIT Press, 1988.

[2] J. C. M. Baeten and W. P. Weijland, Process Algebra, Cambridge Tracts in Theoretical Computer Science 18, 1990.

[3] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice Hall International Series in Computer Science, 1997.

[4] R. Gorrieri and A. Rensink, Action Refinement, Handbook of Process Algebra, Elsevier, 2001, p. 1047-1147.

[5] A. Rensink and R. Gorrieri, Vertical Implementation, Information and Computation, 170-1, 95-133, 2001.

[6] A. Valmari and M. Tienari, Compositional Failure-based Semantics Models for Basic LOTOS, Formal Aspects of Computing, 7-4, 440-468,1995.

[7] C. Bolton and J. Davies, A Singleton Failures Semantics for Communicating Sequential Processes, Oxford University Computing Laboratory,PRG-RR-01-11,2001.

[8] E. Brinksma and A. Rensink and W. Vogler, Fair testing, LNCS 962,Springer-Verlag,313-327,1995.

[9] E. Brinksma and A. Rensink and W. Vogler, Applications of Fair testing, FORTE, IFIP Conference Proceedings,69, 1996.

[10] E. Brinksma and G. Scollo, Formal notions of implementation and conformance in LOTOS, INF-86-13, Twente University of Technology, Department of Informatics, Enschede, The Netherlands,1986.

[11] E. Brinksma and G. Scollo and C. Steenbergen, LOTOS specifications, their implementation and their tests., B. Sarikaya and G. V. Bochmann, Protocol Specification, Testing and Verification, VI, 349–360, 1986.

[12] J. Derrick and E. A. Boiten, Refinement in Z and Object-Z: Foundations and Advanced Applications, 2001, Formal Approaches to Computing and Information Technology.

[13] J. Derrick and E. A. Boiten, Non-atomic Refinement in Z, FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II, 1999, 1477–1496.

[14] C. Ene and T. Muntean, Testing Theories for Broadcasting Processes, Submitted for publication, http://www.esil.univ-mrs.fr/∼cene, 2004.

[15] C. Ene and T. Muntean, Expressiveness of Point-to-Point versus Broadcast Communications, In Fundamentals of Computation Theory,12th International Symposium FCT'99, FCT'99LNCS 1684 1999

[16] C. Fischer and H. Wehrheim, Behavioural Subtyping Relations for Object-Oriented Formalisms, LNCS,1816,469–483,2000.

[17] C. Hoare and H. Jifeng, Unifying Theories of Programming, Prentice Hall International Series in Computer Science, 1998 .

[18] R. Milner, Communication and Concurrency, Prentice-Hall International, 1989.

[19] V. Natarajan and R. Cleaveland, Zoltán Fülöp and Ferenc Gécseg, Divergence and Fair Testing, ICALP, Springer, LNCS, 944, 1995, 648-659.

[20] S. Reeves and D. Streader, Atomic Components, ICTAC 2004,LNCS 3407,128-139.

[21] S. Reeves and D. Streader, Constructing Programs or Processes, Computer Science Technical Report 09/2005.

[22] S. Reeves and D. Streader, Comparison of Data and Process Refinement, ICFEM 2003, LNCS 2885, J. S. Dong and J. C. P. Woodcock, 266-285.

[23] R. Segala, Quiescence, Fairness, Testing, and the Notion of Implementation (Extended Abstract), International Conference on Concurrency Theory, 324-338, 1993.

[24] W.J. Fokkink, J.F. Groote and M.A. Reniers, Process algebra needs proof methodology, The Concurrency Column (L. Aceto, editor), Bulletin of the EATCS,L. Aceto, editor, 2004, 82, 108-125.

[25] T. Bolognesi and E. Börger, Abstract State Processes, 218-228, LNCS, 2589, Abstract State Machines, Advances in Theory and Practice, 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, Proceedings, 2003.