Accepted Manuscript

Contexts, refinement and determinism

Steve Reeves, David Streader

PII: S0167-6423(10)00210-8
DOI: 10.1016/j.scico.2010.11.011
Reference: SCICO 1270

To appear in: *Science of Computer Programming*

Received date: 5 May 2010
Revised date: 15 November 2010
Accepted date: 19 November 2010

Please cite this article as: S. Reeves, D. Streader, Contexts, refinement and determinism, *Science of Computer Programming* (2010), doi:10.1016/j.scico.2010.11.011

Contexts, Refinement and Determinism

Steve Reeves[a,*], David Streader[a,*]

[a]*Department of Computer Science, University of Waikato, Hamilton, New Zealand*

Abstract

In this paper we have been influenced by those who take an "engineering view" of the problem of designing systems, i.e. a view that is motivated by what someone designing a real system will be concerned with, and what questions will arise as they work on their design. Specifically, we have borrowed from the testing work of Hennessy, De Nicola and van Glabbeek, e.g. [13, 5, 21, 40, 39].

Here we concentrate on one fundamental part of the engineering view and where consideration of it leads. The aspects we are concerned with are computational entities in contexts, observed by users. This leads to formalising design steps that are often left informal, and that in turn gives insights into non-determinism and ultimately leads to being able to use refinement in situations where existing techniques fail.

1. Introduction

1.1. Initial comments

Refinement is the stepwise process of developing a specification towards, or perhaps into, a satisfactory implementation. Each refinement step formalises a design decision and transforms a more abstract specification into a more concrete one. One question that arises in practical situations is: "what sort of refinement should I be using on this problem?".

One way to answer this problem is to borrow the ideas behind testing and to take an "engineering view" of the problem of designing systems, i.e. a view that is motivated by what someone designing a real system will be

*Corresponding authors
Email addresses: stever@cs.waikato.ac.nz (Steve Reeves), dstr@cs.waikato.ac.nz (David Streader)

concerned with, and what questions will arise as they work on their design. While we do not address this engineering view directly in this paper, we are largely motivated by it and borrow from the related testing work of Hennessy, De Nicola and van Glabbeek, e.g. [13, 5, 21, 40, 39].

Using these ideas we introduced and formalised a general model of refinement in [27, 28] and used it to show how error states might be introduced during design, and how new operations, or features, can be added too. This general model is a generalisation of event-based models with handshake events, event-based models with broadcast events and state-based models of abstract data types (ADTs) and programs. We formalise our general model by largely avoiding syntax basing it on the operational semantics. Both event-based models use labelled transition systems (LTSs) to define their operational semantics and the state-based model uses sets of named partial relations (NPRs) to define the operational semantics of ADTs. To avoid alienating either the event-based community or the state-based community we postpone fixing which of the two equivalent operational semantics we use until we talk about specific instances of our general model.

We formally define refinement that preserves certain guarantees and have explored this theme of preservation of guarantees as a defining principle behind the concept of refinement in [30].

Following the engineering view, we take as primitive the following three components: a set of **entities**, the specifications and implementations we wish to develop by refinement; a set of **contexts**, the environment with which the entities interact; and a **user**, formalised by defining the set of observations that can be made when an entity is executed in a given context.

We frequently use the notion of a context X waiting to "accept" an entity because contexts can be pictured as, and defined by, terms with "holes" in. Informally speaking the context of an entity is no more than a definition of how the surrounding world interacts with the entity.

We are also explicit about what can be observed when an entity is placed in some context, so given a set of contexts Ξ we also have an observation function O. With non-deterministic entities O returns a set of observations taken from $\mathbb{O}$ and with deterministic entities the set that O returns is a singleton set. A function that returns a set is, of course, equivalent to a relation, hence the semantics of an entity E, written $[\![\mathsf{E}]\!]$, is a subset of $\Xi \times \mathbb{O}$.

A set of entities, their contexts and an observation function define a *layer* (rather like a layer in a protocol stack) at a certain stage of abstraction in a design. Fixing both Ξ and O also fixes $\Xi \times \mathbb{O}$ which we call the *frame* of

behaviours that we are interested in. So, layers and frames are closely linked ideas.

We will sometimes refer to refinement within a layer (pushing the layering analogy further) as *horizontal refinement*, which can also be viewed as the reduction of non-determinism within the frame until the behaviour is deterministic. Then, *vertical refinement* between layers, or frames, is what allows for the expansion of the frame, i.e. the introduction of new contexts and observations, and optionally renaming of the existing contexts and observations that go to make it up.

We still need to consider what exactly we mean by non-determinism and determinism. Milner, when talking about processes, makes a very interesting [20, p232] comment about determinism:

> "Whatever its precise definition, it certainly must have a lot to do with predictability; if we perform the same experiment twice on a determinate system – starting each time in its initial state – then we expect to get the same result, or behaviour, each time."

Even though our notion of entity is more general than the processes Milner was considering, the general idea that he is concerned with in the quote above is something we will use in this paper, so we formalise this comment as directly as we can. Thus "if we perform the same experiment twice on a determinate system – starting each time in its initial state –" becomes, for us, *if we place the entity in the same context and run the test twice* and " we expect to get the same result, or behaviour, each time" becomes *we observe the same behaviour*. Consequently,

> an entity (which will be situated in a certain frame, say $\Xi \times \mathbb{O}$) is deterministic if its behaviour within the frame is given by a function $\Xi \rightarrow \mathbb{O}$.

We call this characterisation *Gen-Det*. We use *Gen-Det* later in the paper when we consider how to refine processes which, conventionally, are viewed as deterministic and hence not refinable, which gives further motivation to the general view we are concerned with in this paper.

On finite automata the definition of deterministic is defined directly on the structure of the automata as:

> no state has more than one transition, or event, with a given label leaving it.

This definition of deterministic finite automata (which we call *DFA*) *corresponds* to, i.e. picks out exactly the same automata as, *Gen-Det* when the finite automata are placed only in contexts that are programs, i.e. linear sequences of events (method calls).

However, both processes with handshake actions, e.g. CSP and CCS, and processes with broadcast actions, e.g. IOA and CBS, place processes in more contexts than just programs.

The literature on broadcast processes frequently provides a new definition of determinism defined on the structure of the automata and it is quite easy to see that this definition corresponds to our *Gen-Det* when the finite automata are placed only in contexts that are valid broadcast processes.

But, for handshake processes like in CSP and CCS the *DFA* definition of deterministic is inherited, even though these event-based models can be placed in a much wider set of contexts than the *DFA* definition was designed for, and the *DFA* definition of deterministic is not the same predicate as *Gen-Det* within this wider set of contexts.

To help us understand why this definition *Gen-Det* of determinism is different from the others, we construct, at an entirely intuitive level, an interpretation or explanation of the difference in Section 3. It is this difference and its explanation that we find more insightful than the similarity between determinism in the other formal models.

1.2. Outline of paper

We show, in Section 2, how a consideration of the interfaces between the entities, contexts and their users is useful in providing clarity, and show how several familiar systems from the literature can be characterised as special cases of this general model. We consider the case where our entities are processes in Section 3, where we also delve further into the consequences of *Gen-Det* and *DFA*.

The view of these instances of the general model as layers, within which the notion of refinement as reduction of non-determinism exists, and the notion of refinement between layers, or *vertical refinement*, is considered in detail in Section 4.

In Section 5 we give an example of the utility of vertical refinement by constructing a refinement between broadcast processes and interactive branching programs, and we see how interactive branching programs can be implemented on a platform which provides (just) broadcast communication.

We also show how the design step of adding error events is expressible as vertical refinement.

An appendix covers some elementary, common definitions and ideas, for completeness.

2. Interfaces

Interfaces can be classified in various ways. In this section we will classify them into two types according to when interaction occurs. Later we will need to classify them according to the type of the interaction.

2.1. Interface types

We will refer to an interface as *transactional* if interaction (which we formalise as observation) occurs at no more than two distinct points: on initialisation, when the interaction starts, and finalisation, when and if the interaction ends. If termination is successful then there are distinct observations that could be made at finalisation, but if termination is unsuccessful then all that can be "observed" is that the entity fails to terminate.

An example of an entity with transactional interaction is a program that accepts a parameter when called and returns a value when it terminates. Clearly if the program fails to terminate no value can be returned.

In contrast we refer to an interface as *interactive* when interaction can occur and be observed at many points throughout the execution. Hence with interactive interfaces observations can be made prior to termination and even prior to non-termination.

Because we are interested in formalising, as directly as possible, the observation of interactive entities we break the observation of an entity into a sequence of more primitive observations. The observation of transactional entities can also be formalised as a sequence of observations, even though this is not normally how state-based approaches describe the observations they make.

An example of an interactive entity is a coffee machine. To obtain two cups of coffee the user first inserts a coin, then pushes the appropriate button and takes the first cup of coffee. But if, after inserting a second coin, the vending machine now fails to terminate, the previously successful interactions mean that what has been observed cannot be represented by noting non-termination alone. (We still have our first cup of coffee!)

So, we have two interfaces, of yet to be determined type. Clearly with two interfaces, each of which could be one of the two types transactional or interactive, we have four cases to consider.

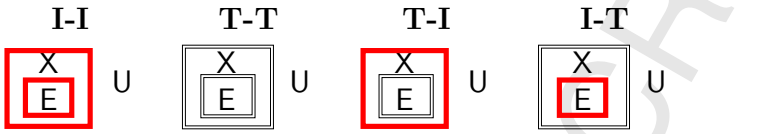

Figure 1: Interactive interfaces ▬ and transactional interfaces ═══

We are concerned with how user U observes entity E, even though the observation has to be made indirectly through context X. Clearly X acts as an intermediary in this communication. The most that U can usefully observe is all that occurs at the E–X interface hence, if we can find an X that acts as a perfect communication buffer between the two interfaces, it is safe to view the situation as having one interface, that between E and X, so in effect U=X.

By assuming that the set of contexts is sufficiently large we are able to find a context X that acts as a perfect communication buffer from the E–X interface to the X–U interface in the first three cases. In **T-T** and **T-I** we can build an X that passes any initialisation information from U to E and if E terminates then passes its response out to U.

Now consider the **I-I** case. We assume the existence of actions (we give them hats in the sequel) that our context X may perform that do not synchronise with any action of the entity E. Using these actions we can easily construct a context $\widehat{\mathsf{X}}$ that after synchronising with E performs a distinct special observable action $\widehat{\mathsf{a}}$ that announces to U the fact that the a action has been performed. So we have (considering the entities as given by LTS for the moment) that:[1]

$$\text{if } n \xrightarrow{\mathsf{a}}_{\mathsf{X}} m \text{ then } n \xrightarrow{\mathsf{a}}_{\widehat{\mathsf{X}}} z \xrightarrow{\widehat{\mathsf{a}}}_{\widehat{\mathsf{X}}} m \text{ where } z \text{ is not a node in X}$$

Such contexts are a perfect communication buffer as they have the effect of making visible, to the user U, any action in the E–X interface.

[1] In the relational semantics of [7, 8] they need to model the refusal of a set of operations as observable to give liveness semantics for processes. It should be noted that we do not need to do this because the domain of our relation is different.

In the **I-T** case X cannot be a perfect communication buffer. The problem lies in the fact that the interactive interface E–X is able to pass information from E to X even if E subsequently fails to terminate. But because the interface X–U is transactional it is unable to pass this information to U. Hence no matter how large the set of contexts there can be no perfect communication buffer for the **I-T** case.

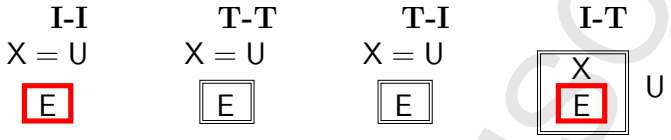

Figure 2: With a sufficiently large set of contexts Ξ

Later we will give more concrete examples of all four cases in Fig. 2, and in the **I-I**, **T-T** and **T-I** cases (left-hand three cases of Fig. 2) we will be able to define contexts that behave as perfect communication buffers and hence these cases can be modelled by considering only one interface.

2.2. Testing and divergence

We show the usefulness of **I-T** cases by using them to amend Hennessy and de Nicola's [5, 13] classic testing semantics. Their semantics models a divergent process as having chaotic behaviour and this is known to have the disadvantage that "divergence typically masks much [observed] behaviour we should really want to see" [32, p297]. The advantage of our amendment is that divergent processes mask less behaviour.

Hennessy and de Nicola make explicit use of both contexts and user (experimenter) in their testing semantics of processes, but only success $\top$ and failure $\bot$ are recorded. In our terminology this testing semantics is an **I-T** case. The obvious questions that arise from our description are: would the **I-I** style testing semantics give different and interesting results; and if so, which best describes how the processes we are interested in are observed?

It is easy to see that for potentially non-terminating processes the **I-I** style of tests do indeed make a difference. Let us consider A and B in Fig. 3. In both these processes there exists a t loop originating from state n so when run (i.e. when composed in parallel with—defined in the Appendix) in contexts containing t the system has a τ loop. With the interpretation that τ loops can run forever (as made by CSP) we must conclude that after executing an a

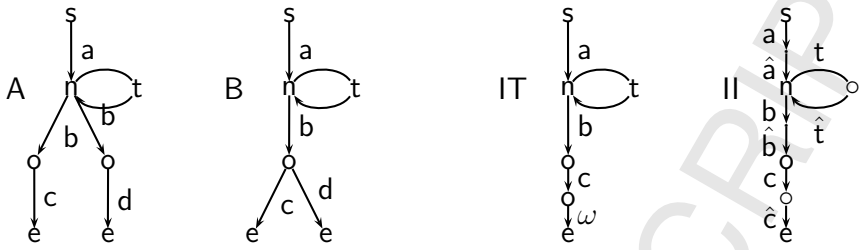

Figure 3: **I-I** tests give a less masking semantics of divergence than **I-T** tests

action both processes may never terminate. Consider a context IT that tries to execute a, b, c and ω, a special event to mark success $\top$, and assume that after a IT can perform a t loop. With this test, with either A or B, ω can be observed or nothing is observed. So, with **I-T** tests the two processes A and B cannot be distinguished as both success $\top$ and failure $\bot$ can be observed. We leave the interested reader to convince themselves that this is the case no matter what context is chosen.

But, with **I-I** tests they can be distinguished and we only need consider executions with II as context :

- the execution with B fails (i.e. $\hat{c}$ is not observed) only when B moves from state s to state n due to a and then forever executes a t action;
- the execution with A can fail (so $\hat{c}$ is not observed) when A, like B, moves from state s to state n due to a and then forever executes a t action; but in addition it might move from state s to state n due to a and then move from n to o-right using b.

Therefore A and B are different, as with test II only process A can have the trace of length two $\hat{a}$, $\hat{b}$ observed; for process B, if an observation contains $\hat{a}$, $\hat{b}$ then it must contain $\hat{c}$ too and be of length three. Thus by a slight change in Hennessy's testing semantics we have a semantics of non-termination that masks less of the behaviour of processes.

Having considered the two interfaces (entity/context and context/user) thrown-up by the testing-inspired entity-context-user model in this section, we move on to concentrating on the entity-context interface in the next, essentially by treating the user as an unvarying presence that can be "factored out".

3. Considering processes

In [28] we worked in the state-based world and we viewed an abstract data type (ADT) as being a set of named partial relations (NPR) and explored our general notion of refinement in that setting, with contexts being programs. A program is a linear, unbranching sequence of operations. Each ADT-program interaction is a call to (i.e. a use of) one of the ADT operations (i.e. one of the named partial relations).

In this section we turn to the event-based world and consider the case where entities are processes. While programs can be modelled by an unbranching sequence of calls to an ADT and programs are the only valid contexts for an ADT, processes can be placed in branching contexts. Thus processes will have a distinct semantics to ADTs because the contexts in which they can be placed are different to those for programs [29].

We will classify processes, as appearing in the literature, into two types. The *handshake* processes of CSP, CCS and ACP treat all events in the same way, i.e. give all events the same kind of semantics. The *broadcast* processes have two types of events, the active *output* events that cause the passive *input* events. The broadcast output event differs from all other observable events that we model in that it is under *local control*, i.e. it cannot be placed in a context that blocks its execution.

The handshake processes of CSP, CCS and ACP abstract away the cause and response nature of event synchronisation. By modelling both a vending machine and robot with the same type of actions the distinction between cause and response is lost. The point here is that, for example, the buttons on a vending machine are *passive* and a robot can actively *cause* the vending machine to *respond* by pushing a button on the vending machine, so something has been lost in the abstraction of events in this way.

As determinism can be thought of as *having a unique response to any action* we should not be surprised that losing the distinction between cause and response might affect how well determinism can be defined. We spend much of the section considering this.

In Section 3.1 we review broadcast processes, and consider what broadcast processes are deterministic on Section 3.2, then in Section 3.3 and Section 3.4 we do the same for handshake processes.

Our *interactive branching programs* of Section 3.5 are classified as processes (regardless of their name!) because they and their contexts can both branch. They can be viewed as a restricted class of handshake processes for

which the cause and response nature of event synchronisation has not been abstracted away and which have active and passive events. We consider determinism for these process in Section 3.6.

Since we are no longer dealing with transactional interfaces, we need make no distinction between context and user in what follows. Further, contexts are going to be special processes that are composed in parallel with the entity processes. Because we wish to define determinism by a unique observation in any deterministic context then we must have that if the user *can* observe something of a process in some state then it *must* observe it. If this was not the case then our act of observation would introduce non-determinism.

This we can easily do by requiring that any action from some state of a context that the user can observe must be the only action enabled from that state.

Placing entity E in a context X will be written $[\mathsf{E}]_{\mathsf{X}}$. We can break this down into parts: E is the entity; the context consists of X and a means of composing it with E. Let us rewrite $[\mathsf{E}]_{\mathsf{X}}$ using a binary operation Op, to give $\mathsf{E}\ Op\ \mathsf{X}$. It is easy to see that if Op is an associative operation and Ξ is closed under Op then refinement is a precongruence, i.e. a congruence based on the refinement pre-order, with respect to Op.

For processes, E will be an LTS , Op will be parallel composition and X will be an LTS too, and in X:

> if $n \xrightarrow{\mathsf{u}} m$ is part of X and action u is not synchronised with E actions but is observed by the user then $n \xrightarrow{\mathsf{v}} p$ implies $\mathsf{v} = \mathsf{u}$ and $m = p$.

We call this *property 1*. We restrict ourselves to contexts satisfying this property to ensure that non-deterministic observers are not allowed.

3.1. Broadcast processes

There has long been interest in the relation between handshake- and broadcast-style communication, but there are many variations of both styles to be considered when trying to elucidate the relationship. A comparison of the point-to-point handshake communication of CCS with the multi-way broadcast of CBS can be found in [9]. But handshake need not be point-to-point, and both CSP and ACP allow multi-way handshake synchronisation. Handshake and broadcast styles also differ in that broadcast has *local control of output*, i.e. a listener cannot block a multi-way radio message from being

broadcast nor can a receiver block a point-to-point email message from being broadcast, whereas with handshake-style communication any events can be blocked. The only difference between our handshake and broadcast models will be that broadcasts cannot be blocked by any context and both will model point-to-point communication (hence synchronisation will be between pairs of events).

Even restricting communication to point-to-point there is a variety of different ways to formalise broadcast communication. Some models of broadcast systems [35, 22, 16, 10] define parallel composition in such a way that output events cannot be blocked. The alternative approach, found in [36, 23, 33, 19, 18] and used here, is to keep parallel composition the same as defined for handshake operations and consider only entities, and thus contexts, that have input actions always enabled so that outputs cannot be blocked.

In what follows we will often decorate events with question marks and exclamation marks just to emphasise that the events concerned are most helpfully to be thought of as events considered as inputs ("listening") or events considered as outputs ("broadcast") , respectively. When composed synchronisation still happens as usual between events with the same (base) name, e.g. $\mathsf{a?}$ and $\mathsf{a!}$ will synchronise since they have the same base name a.

In terms of sets of contexts and sets of entities we can characterise broadcast process thus:

$$\Xi_{\mathsf{BC}} \triangleq \{[\_]_x \mid x \in \mathsf{T}_{\mathsf{BC}}\}$$

and

$$\mathsf{T}_{\mathsf{BC}} \triangleq \{\mathsf{A}\ an\ LTS \mid \forall\, n \in N_{\mathsf{A}}, \mathsf{a?} \in Act.\, \exists\, m \in N_{\mathsf{A}}.\ n \xrightarrow{\mathsf{a?}} m\}$$

where

$$Act = \{\mathsf{a!} \mid \mathsf{a} \in Names\} \cup \{\mathsf{a?} \mid \mathsf{a} \in Names\} \cup \{\tau\}$$

3.2. Determinism and broadcasting

Here we turn to our theme of seeing how determinism looks in the context of our various sorts of process.

We define a function M_{BC} that turns a LTS into a broadcast process by simply adding *listening loops* $n \xrightarrow{\mathsf{a?}} n$ to any n for which $\mathsf{a?}$ is not enabled:

$$M_{BC}(\mathsf{A}) \triangleq (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}} \cup \{n \xrightarrow{\mathsf{a?}} n \mid \neg\ n \xrightarrow{\mathsf{a?}}\})$$

Having introduced them, it is frequently clearer to not explicitly show listening loops (see Fig. 4). Such LTSs can be interpreted as broadcast processes by leaving listening loops implicit.

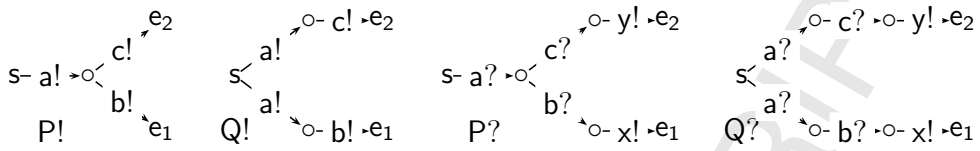

Figure 4: $M_{BC}(\mathsf{P!}) =_{BC} M_{BC}(\mathsf{Q!})$ and $M_{BC}(\mathsf{P?}) \neq_{BC} M_{BC}(\mathsf{Q?})$

The effect of M_{BC} can be best understood by considering some examples. Consider Fig. 4. Processes $M_{BC}(\mathsf{P?})$ and $M_{BC}(\mathsf{Q?})$ are not trace equivalent, e.g. $\mathsf{a?b?c?y!} \notin M_{BC}(\mathsf{P?})$ because if $M_{BC}(\mathsf{P?})$ hears a $\mathsf{b?}$ event after the initial $\mathsf{a?}$ event then it must output $\mathsf{x!}$ not $\mathsf{y!}$ but $\mathsf{a?b?c?y!} \in M_{BC}(\mathsf{Q?})$ as the process, on hearing $\mathsf{a?}$, can make one of two moves, one of which will lead to output $\mathsf{y!}$ being made. This is not the result that might be expected from the handshake perspective where trace semantics are unable to distinguish $\mathsf{P?}$ and $\mathsf{Q?}$. (Remember that we assume listening loops on all nodes.)

$\mathsf{P!}$ can broadcast either $\mathsf{b!}$ or $\mathsf{c!}$. As broadcast output is under local control no other process can block either of these events. Hence it seems unavoidable that we consider $\mathsf{P!}$ to be non-deterministic. Yet clearly $\mathsf{P!}$ and $M_{BC}(\mathsf{P!})$ are deterministic transition systems according to the usual criterion *DFA* (there are no nodes with two (or more) transitions with the same name leaving them).

Clearly there is a mismatch between our intuitions on the one hand and the usual *DFA* characterisation on the other hand. Because of this mismatch we turn to another characterisation, *Gen-Det* from Section 1.1.

We define the set of deterministic broadcast processes, as in [36, 23], as processes, ignoring listening loops (prior to applying M_{BC}), that branch on only input events with different names (and where $Act^?$ is $\{\mathsf{a}^? \mid \mathsf{a} \in Names\}$):

Definition 1. *The set of deterministic broadcast processes, D_{BC}:*

$$
\begin{aligned}
D_{BC} \triangleq \{\mathsf{B}\ \textit{an LTS} \mid\ & (n \xrightarrow{\mathsf{x!}}_{\mathsf{B}} m \wedge\ n \xrightarrow{\mathsf{y}}_{\mathsf{B}} k) \Rightarrow \\
& (\mathsf{y} = \mathsf{x!} \wedge m = k \\
& \vee\ \mathsf{y} \in Act^? \wedge k = n) \\
\vee\ & (n \xrightarrow{\mathsf{x?}}_{\mathsf{B}} m \wedge\ n \xrightarrow{\mathsf{y}}_{\mathsf{B}} k \wedge n \neq m) \Rightarrow \\
& \mathsf{y} \neq \mathsf{x?} \wedge \mathsf{y} \in Act^?\}
\end{aligned}
$$

Lemma 1. *If* D *is a deterministic broadcast process from* D_{BC}*, then Gen_Det holds, thus for all contexts* X *we have that* $O([\mathsf{D}]_{\mathsf{X}})$ *is a singleton set.*

Proof. (By contradiction.) Assume $\mathsf{D} \in D_{BC}$ and $O([\mathsf{D}]_{\mathsf{X}})$ is not a singleton set. As $O([\mathsf{D}]_{\mathsf{X}})$ is not a singleton set $[\mathsf{D}]_{\mathsf{X}}$ must have at least two execution traces that are observed to be distinct. As only X actions can be observed and as property 1 (given just before section 3.1) tells us only one action is enabled from any state when an observable action is enabled, we know that these two execution traces must lead to distinct X states.

By induction on the length of the execution traces we show this cannot be the case.

Base case: length equals one. The only way X can lead to two states after one action is if $s_N \xrightarrow{\mathsf{x?}} p$ and $s_N \xrightarrow{\mathsf{y?}} q$ are in X but for both of these listening events to be executable D would have to be prepared to output two distinct actions. In order for this to happen D would already have to be in two distinct states, whereas it must be in state s_N.

Induction hypothesis: no branching has occurred for the first $n-1$ actions. The only way X can lead to two states after n actions is if $s_{n-1} \xrightarrow{\mathsf{x?}} p \wedge s_{n-1} \xrightarrow{\mathsf{y?}} q$ but for both of these listening events to be executable D would have to be prepared to output two distinct actions. In order for this to happen D would already have to be in two distinct states. For this to be the case X would have had to been prepared to execute two output actions and this could only be the case X was in two distinct states, and by the induction hypothesis this cannot be true. •

D_{BC} accords with Milner's comment (from Section 1.1) and we draw the reader's attention to the fact that the definition of determinism in [36, 23] is consistent with our characterisation *Gen-Det*. In CBS all sequential processes are deterministic: "Speakers in parallel are the only source of non-determinism in CBS" [23]. An informal justification for this limitation is that branching outputs of a sequential process could not be implemented on a deterministic machine.

3.3. Handshaking processes

Any LTS can be used as the operational semantics for a handshake process and such processes can be placed in a context consisting of any LTS. Hence for handshake processes the characterisation is:

$$\Xi_{\mathsf{PA}} \triangleq \{[\_]_x \mid x \ \in \mathsf{T}_{\mathsf{PA}}\}$$

and

$$\mathsf{T}_{\mathsf{PA}} \triangleq \{\mathsf{A}\ an\ LTS \mid \alpha(\mathsf{A}) \subseteq Names\}^2$$

3.4. Determinism and handshaking

The characterisation of deterministic processes we have chosen, e.g. for broadcast processes in Section 3.2, is very different to the characterisation (basically that a process is deterministic if no node has two transitions with the same label leaving it) that is found in the process-algebraic literature. We consider two simple examples of processes to investigate this.

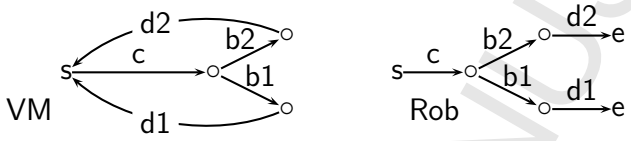

Figure 5: Are VM and Rob deterministic?

The vending machine VM in Fig. 5 starts by waiting for a coin to be inserted (c) and then for one of two buttons to be pushed (b1 or b2) after which a drink (d1 or d2) is dispensed and the vending machine returns to the start state. We will show that the interpretation of the robotic user Rob in Fig. 5 requires some thought.

Non-determinism can arise naturally with concurrent processes, for example running processes R1 $\triangleq$ c;b1 and R2 $\triangleq$ b2 in parallel with VM. After R1 inputs a coin R1 and R2 *race* to push different buttons and which button is pushed is not determined. We accept Hoare's view [15, p81] that: "*There is nothing mysterious about this kind of non-determinism: it arises from a deliberate decision to ignore the factors which influence the selection*". By restricting ourselves to untimed models of processes we view this non-determinism as arising from a deliberate decision to ignore time. Alternatively, non-determinism can be viewed as partial specification to be resolved by refinement prior to implementation.

In CSP, CCS and ACP Rob is deterministic but exhibits non-deterministic behaviour when interacting with VM, that is, when Rob and VM are run in

[2] α gives the set of names used by an LTS, i.e. its alphabet.

parallel the drink that Rob ends up with is not determined. It is not clear from the literature whether the non-determinism of $[\mathsf{Rob}]_{\mathsf{VM}}$ is a natural consequence of implementable processes or is due to partial specification and is unavoidable because the model has abstracted away the cause; or should we expect to resolve it by further refinement? Unfortunately, however, both Rob and VM are viewed as deterministic in CSP, CCS and ACP and therefore neither can be refined.

This leads us to the obvious question: what factor is ignored in the Rob and VM example that causes this non-determinism to arise? It is our view that the robot, not the vending machine, has to select what button to push and consequently it must be the *robot's choice* that has been ignored.

Note that an important point, which emerges on comparing the two examples here, is that the non-determinism comes from *different* factors being ignored. As we said, time is ignored in the first example involving R1 and R2, giving rise to their racing. In the second example we have ignored cause-and-effect, and this has led to the non-determinism there. Thus, since the reasons for the non-determinism are different, it would be entirely reasonable if the "solutions" in each case might be different too. Put another way, since we can differentiate between two different sorts of non-determinism (by reason of the different factors ignored) then it would not be surprising if we dealt with them in different ways too. In one case, the race case, we might accept it and in the other, the cause-and-effect case, we might not and seek to remove it.

Process algebras have abstracted away the *cause* and *response* nature of event synchronisation, e.g. the robot's "button pushing" events cause the vending machine's "button pushed" event to occur. This makes it hard for process algebra to require that the robot, and not the vending machine, must make a choice as to what button to push.

Cause and response are modelled in broadcast operations in Section 3.1 by requiring pairs of events that synchronise to consist of one passive event and one active event, the latter causing the former to occur. We apply this approach to handshake processes in the next section.

This is the only model in which our characterisation of determinism differs from that found in the literature. This can be used to argue that there is a weakness in our general model. But an alternative view is that because these process models have chosen to abstract away the *cause* and *response* nature of event synchronisation they are forced to accept that determinism is hard to define: recall Milner's comment [20, p232] about determinism from

Section 1.1.

We must take care not to be misled here: processes such as Rob and VM can be coded in the Occam programming language and their concurrent execution can be performed on a single transputer, so we might think that there is no non-determinism here in our example as transputers, like other digital computers, are finite-state deterministic machines and so they cannot exhibit non-deterministic[3] behaviour, and so our processes running together cannot, therefore, be non-deterministic, so there is no problem after all; it might be thought that we are mistakenly worrying about this.

But, the Occam compiler in fact decides which button is pushed. This could be described as the Occam compiler refining $[\mathsf{Rob}]_{\mathsf{VM}}$ by removing non-determinism and then implementing the deterministic process produced by the refinement. So, Occam does not actually implement the non-deterministic process (obviously; how could it?) we are concerned with, but it refines this process (it might, for example, have a strategy within the compiler which says: "when there is a choice take the first alternative") and implements *that*. So, there is non-determinism here, and we view the source of it as being the interpretation of Rob given by the process algebras, and Rob in our opinion is therefore *not implementable* as it stands. Occam shows one way to change Rob so as to render it implementable, and there may be other ways.

3.5. Interactive branching programs, IBP

Interactive programs are different from the processes of CSP/CCS. Processes are prepared to perform an operation from a whole set of operations, whereas programs are only prepared to perform one specific operation. For example, a program can perform some sequences of push and pop operations on a stack that offers both these operations. But a process, not a program, can offer the stack the ability to perform either push or pop and allow the stack to select which.

We have seen different styles of event interactions for both processes and programs and now we introduce another style of interaction, interactive branching programs (IBP) from [25], that combines process and program ideas.

It is common in the literature on handshake events ([15, 32, 20, 2]) to treat events that synchronise in exactly the same way, and not differentiate

[3]They can exhibit complex behaviour that approximates non-deterministic behaviour but they are inherently deterministic.

between, for example, the events of Rob and the events of VM. It is our intuition that the events of a vending machine VM are *passive* and the events of a robot Rob are *active* and cause the passive events of VM to occur, just as a program causes a method of an ADT to be executed. For IBP we view the active events as causing the performance of the passive events, but unlike broadcast events, and like programs and ADT, we do not have local control of the active events. Thus we allow the active events to be blocked by a context. The active events are written with the name over-lined (e.g. $\overline{\mathsf{a}}$) and the passive events with no over-line (e.g. a).

As the active events of IBP are the calling of a method (or the causing of a passive event) we model it as *committing*, i.e. once started the caller cannot back off but is blocked if the passive event cannot be executed.

In order to formalise this we restrict the LTS that can be used to represent IBP. These LTS require that active events must be preceded by a unique τ event (see Fig. 6 for an example of how this looks) and after this τ event only the single active event can be executed. We therefore characterise IBP as follows:

$$\Xi_{\mathsf{IBP}} \triangleq \{[\_]_x \mid x \in \mathsf{T}_{\mathsf{IBP}}\}$$

where

$$\mathsf{T}_{\mathsf{IBP}} \triangleq \{\mathsf{A}\ an\ LTS \mid n \xrightarrow{\overline{\mathsf{a}}}_{\mathsf{A}} r \wedge n \xrightarrow{\mathsf{x}}_{\mathsf{A}} t \Rightarrow (\overline{\mathsf{a}} = \mathsf{x} \wedge r = t) \wedge$$

$$q \xrightarrow{\mathsf{y}}_{\mathsf{A}} n \xrightarrow{\overline{\mathsf{a}}}_{\mathsf{A}} \wedge\ p \xrightarrow{\mathsf{z}}_{\mathsf{A}} n \Rightarrow (\mathsf{y} = \mathsf{z} = \tau \wedge p = q)\}$$

3.6. Determinism and IBP

We define $M_{IBP}(\mathsf{A})$ which changes an LTS's operational semantics to be IBP processes. The only change it makes is to active events.

Definition 2. *For* A *an LTS* $(N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$*:*

$$M_{IBP}(\mathsf{A}) \triangleq (N_{M_{IBP}(\mathsf{A})}, s_{\mathsf{A}}, T_{M_{IBP}(\mathsf{A})})$$

where

$$N_{M_{IBP}(\mathsf{A})} \triangleq N_{\mathsf{A}} \cup \{z_{(n,\mathsf{a},m)} \mid n \xrightarrow{\overline{\mathsf{a}}}_{\mathsf{A}} m\}$$

and

$$T_{M_{IBP}(\mathsf{A})} \triangleq \{n \xrightarrow{\mathsf{a}} m \mid n \xrightarrow{\mathsf{a}}_{\mathsf{A}} m\} \cup \{n \xrightarrow{\tau} z_{(n,\mathsf{a},m)} \xrightarrow{\overline{\mathsf{a}}} m \mid n \xrightarrow{\overline{\mathsf{a}}}_{\mathsf{A}} m\}$$

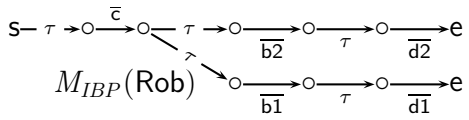

Figure 6:

The IBP process $M_{IBP}(\mathsf{Rob})$ (Fig. 6) is a non-deterministic specification of the behaviour of Rob in Fig. 5 where the non-determinism arises from not specifying which button the robot will push.

We define the set of deterministic IBP in the same way as the deterministic broadcast processes in Section 3.1. The deterministic IBP are the processes, prior to applying M_{IBP}, that branch on only passive events with different names.

Definition 3. *The set of deterministic IBP, D_{IBP}:*

$$D_{IBP} \triangleq \{\mathsf{P} \textit{ an IBP} \mid q \xrightarrow{\tau}_{\mathsf{P}} n \xrightarrow{\overline{\mathsf{y}}}_{\mathsf{P}} r \wedge q \xrightarrow{\mathsf{z}}_{\mathsf{P}} m \Rightarrow \tau = \mathsf{z} \wedge n = m \wedge n \xrightarrow{\mathsf{x}}_{\mathsf{P}} m \wedge n \xrightarrow{\mathsf{x}}_{\mathsf{P}} k \Rightarrow m = k\}$$

$[\mathsf{Rob}]_{\mathsf{VM}}$ (taking Rob and VM from Fig. 5) and both $[M_{IBP}(\mathsf{Rob})]_{M_{IBP}(\mathsf{VM})}$ and $M_{IBP}(\mathsf{Rob})$ (see Fig. 6) are non-deterministic. This is not because distinct sequential processes are racing to perform active events but because the robot fails to choose what active event it will perform. What is more $M_{IBP}(\mathsf{Rob})$ can be refined into a deterministic IBP, whereas no refinement of the robot was possible using the process semantics of Fig. 5.

There are two ways to relate IBP and process algebra. Either we say that IBP is a subset of process algebras, $\mathsf{T}_{\mathsf{IBP}} \subset \mathsf{T}_{\mathsf{PA}}$, or IBP can be mapped onto process algebras by removing the τ events. With this second relation, we will see, IBP refinement extends process algebra refinement, $\sqsubseteq_{\mathsf{PA}} \subset \sqsubseteq_{\mathsf{IBP}}$.

We leave it for the interested reader to check that D_{IBP} satisfies *Gen-Det* (in Section 1.1) but draw the reader's attention to the fact that this definition of determinism for IBPs is consistent with our abstract characterisation. Thus IBP is a subset of handshaking-style processes in Section 3.3 for which the *cause* and *response* nature of event synchronisation has not been abstracted away and for which determinism is consistent with our abstract definition.

4. General Refinement

The general notion of refinement as used in [28] is given by the following definition.

Definition 4. *General refinement. Let* Ξ *be a set of contexts each of which the entities* A *and* C *can communicate privately with, and let* O *be a function which returns a set of observations, each observation being what a user observes of an execution. Then:*

$$\mathsf{A} \sqsubseteq_{\Xi,O} \mathsf{C} \triangleq \forall\, x \in \Xi . O([\mathsf{C}]_x) \subseteq O([\mathsf{A}]_x)$$

Note that for event-based interaction the observations will be sequences of more primitive observations, i.e. *traces*. For abstract data types, i.e. state-based systems, the observations will be of the traditional "started" and "finished" sort (which we can also read as traces with at most two primitive observations in them).

We view the various systems given in the previous sections as *special models* since, using the various characterisations of contexts and entities given, and using them in Definition 4, gives us various specialisations of the general notion of refinement. We now, further, regard these special theories as *layers* in the larger scheme of things (as introduced in [28], where we also introduces the notions of frames and vertical refinement).

4.1. Refinement and layers

By the characterisation given above, a layer is formalised by a set of entities and a set of contexts, and hence by a refinement relation (once we agree to use the same method of observation for all models, as we do here). It is important to recall that the entities in a layer can be ADTs, processes of various kinds and even individual operations.

Definition 5. *A layer* L *is* $(\mathsf{E}_\mathsf{L}, \sqsubseteq_\mathsf{L})$ *where* E_L *is a set of entities and* $\sqsubseteq_\mathsf{L} \subseteq \mathsf{E}_\mathsf{L} \times \mathsf{E}_\mathsf{L}$ *is a refinement relation*

General refinement gives us a way of linking layers and contexts and observation functions. A triple consisting of a set of LTSs representing entities, a set of LTSs representing contexts and an observation function on LTSs, also defines a layer if we can lift the observation function from LTSs to entities, i.e. if $\mathsf{A}_\mathsf{L} =_\mathsf{L} \mathsf{B}_\mathsf{L} \Rightarrow O(\mathsf{A}_\mathsf{L}) = O(\mathsf{B}_\mathsf{L})$, and lift placement in a context from LTS

to entities, i.e. $\mathsf{A_L} =_\mathsf{L} \mathsf{B_L} \Rightarrow \forall\, x \in \Xi_\mathsf{L}.[\mathsf{A_L}]_x =_\mathsf{L} [\mathsf{B_L}]_x$. This is the case for all the models we consider.

We can make the idea of refinement more flexible now that we have layers, by giving a general definition of *vertical refinement* between an abstract and a concrete layer. Since layers themselves have their own refinement relations, this gives us the flexibility to move between theories which have different definitions of refinement while still keeping the essence of what makes refinement a valuable concept—the relation that exists between entities and the guarantees about their behaviours relative to one another.

Definition 6. *Semantic mappings* $[\![\_]\!]_v^{\mathsf{HL}}$ *and* $\mathbb{A}_v^{\mathsf{HL}}$ *define a vertical refinement* $\sqsubseteq_v^{\mathsf{HL}}$ *between high-level layer* $(\mathsf{E_H}, \sqsubseteq_\mathsf{H})$ *and low-level layer* $(\mathsf{E_L}, \sqsubseteq_\mathsf{L})$ *if they are adjoint:*

$$\forall\, \mathsf{X_H} \in \mathsf{E_H}, \mathsf{Y_L} \in \mathsf{E_L}.[\![\mathsf{X_H}]\!]_v^{\mathsf{HL}} \sqsubseteq_\mathsf{L} \mathsf{Y_L} \Leftrightarrow \mathsf{X_H} \sqsubseteq_\mathsf{H} \mathbb{A}_v^{\mathsf{HL}}(\mathsf{Y_L})$$

The definition is based on two semantic mappings: $[\![\_]\!]_v$, that defines how to interpret the high-level abstract entities as low-level concrete entities; and $\mathbb{A}_v$, that defines how to interpret the low-level concrete entities as high-level abstract entities. The semantic mappings are vertical refinements if and only if any low-level refinement is interpreted as a high-level refinement and any high-level refinement is interpreted as a low-level refinement. Mathematically our vertical refinement is a Galois connection (or an adjunction) between the layers.

This definition of vertical refinement can be seen as a generalisation of non-atomic refinement [6] or action refinement [31, 12] when we consider the LTS used to represent entities.

It also turns out to be useful to consider the relational semantics of an entity. This is defined by

Definition 7. *Let* L *be a layer. Let* Ξ_L *be a set of contexts each of which the entity* $\mathsf{A_L}$ *can communicate privately with, and* O_L *be a function which returns a subset of the set* $\mathbb{O}_\mathsf{L}$ *all of observations, each observation being what a user observes of an execution.The relational semantics of an entity* $\mathsf{A_L}$ *is a subset of* $\Xi_\mathsf{L} \times \mathbb{O}_\mathsf{L}$*:*

$$[\![\mathsf{A_L}]\!]_{\Xi_\mathsf{L}, O_\mathsf{L}} \triangleq \{(x, o) \mid x \in \Xi_\mathsf{L}, o \in O_\mathsf{L}([\mathsf{A_L}]_x)\}$$

The whole of the product that goes to provide a universe for the relational model is what we call a *frame*:

$$Frame_{\mathsf{L}} \triangleq \Xi_{\mathsf{L}} \times \mathbb{O}_{\mathsf{L}}$$

and we describe the relation as being a *relation over the frame*. When the frame is the same for all relations under consideration it is frequently left implicit but here we are interested in changing the frame and thus must be explicit. In particular, we are interested here in theories (or layers) H and L where $Frame_{\mathsf{H}} \subseteq Frame_{\mathsf{L}}$.

It is well-known ([34, p155] [14, 4.1]) that subset relations like $Frame_{\mathsf{H}} \subseteq Frame_{\mathsf{L}}$ form a simple theory morphism, and hence a vertical refinement, which we denote by $\sqsubseteq^{\mathsf{HL}}_{sub}$, where the interpretation mappings are:

embedding of the abstract in the more complex concrete, where for any $\mathsf{P_H} \in \mathsf{E_H}$ (using the definitions $\Xi_{\mathsf{L}\setminus\mathsf{H}} \triangleq \Xi_{\mathsf{L}} \setminus \Xi_{\mathsf{H}}$ and $\mathbb{O}_{\mathsf{L}\setminus\mathsf{H}} \triangleq \mathbb{O}_{\mathsf{L}} \setminus \mathbb{O}_{\mathsf{H}}$) :

$$[\![\mathsf{P_H}]\!]^{\mathsf{HL}}_{sub} \triangleq [\![\mathsf{P_H}]\!]_{\Xi_{\mathsf{H}},O_{\mathsf{H}}} \cup \{(x,o) \mid x \in \Xi_{\mathsf{L}\setminus\mathsf{H}} \vee o \in \mathbb{O}_{\mathsf{L}\setminus\mathsf{H}}\};$$

projection of the concrete back into the abstract, where for any $\mathsf{P_L} \in \mathsf{E_L}$:

$$\mathbb{A}^{\mathsf{HL}}_{sub}(\mathsf{P_L}) \triangleq [\![\mathsf{P_L}]\!]_{\Xi_{\mathsf{H}},O_{\mathsf{H}}}.$$

We can establish that $\sqsubseteq^{\mathsf{HL}}_{sub}$ is a theory morphism or vertical refinement, i.e. that:

$$\forall\, \mathsf{X_H} \in \mathsf{E_H}, \mathsf{Y_L} \in \mathsf{E_L}.[\![\mathsf{X_H}]\!]^{\mathsf{HL}}_{sub} \sqsubseteq_{\mathsf{L}} \mathsf{Y_L} \Leftrightarrow \mathsf{X_H} \sqsubseteq_{\mathsf{H}} \mathbb{A}^{\mathsf{HL}}_{sub}(\mathsf{Y_L})$$

by checking that:

$$\forall\, \mathsf{X_H} \in \mathsf{E_H}, \mathsf{Y_L} \in \mathsf{E_L}.[\![\mathsf{X_H}]\!]_{\Xi_{\mathsf{H}},O_{\mathsf{H}}} \cup \{(x,o) \mid x \in \Xi_{\mathsf{L}\setminus\mathsf{H}} \vee o \in \mathbb{O}_{\mathsf{L}\setminus\mathsf{H}}\} \supseteq [\![\mathsf{Y_L}]\!]_{\Xi_{\mathsf{L}},O_{\mathsf{L}}}$$

$$\Leftrightarrow [\![\mathsf{X_H}]\!]_{\Xi_{\mathsf{H}},O_{\mathsf{H}}} \supseteq [\![\mathsf{Y_L}]\!]_{\Xi_{\mathsf{H}},O_{\mathsf{H}}}$$

This is true by basic set theory.

Intuitively we can think of subset morphisms as formalising **being silent outside of frame**.

Our subset theory morphism formalises the addition of new observations, like the y in Fig. 7. The justification for adding $\{(a,y),(b,y),(c,y),(x,y)\}$ is that in the abstract specification, i.e. the relation over $Frame_H$, the y

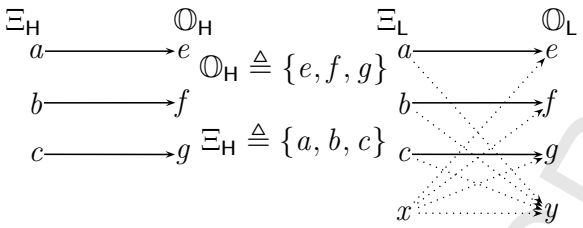

Figure 7: $\mathsf{H} \sqsubseteq^{\mathsf{HL}}_{sub} \mathsf{L}$

observation had not been considered or recorded, so we have no information (yet) about it, which is to say the abstract H is silent outside of $Frame_{\mathsf{H}}$.

It is the adding of the new observations that makes this an example of the flexibility of our formal model of both single operations and machines, and hence an example of the flexibility of vertical refinement. More informally, it is the preservation of the guarantee that allows us to view such theory morphisms *as refinements*, which we now discuss.

The strict embedding projection morphisms satisfy a strict guarantee:

> $\mathsf{P_H} \sqsubseteq^{\mathsf{HL}}_{sub} [\![\mathsf{P_H}]\!]^{\mathsf{HL}}_{sub}$ guarantees the high-level $\mathbb{A}_{sub}$-interpretation of $[\![\mathsf{P_H}]\!]^{\mathsf{HL}}_{sub}$ behaves exactly like entity $\mathsf{P_H}$ whenever it is placed in any abstract context Ξ_{H} and only the abstract observations $\mathbb{O}_{\mathsf{H}}$ are made.

But, this is not always very useful in practice as $[\![\mathsf{P_H}]\!]^{\mathsf{HL}}_{sub}$ may be unhealthy (by which we mean that it is not in fact an entity in L). So we take a more pragmatic view and consider $\mathsf{P_H} \sqsubseteq^{\mathsf{HL}}_{sub} [\![\mathsf{P_H}]\!]^{\mathsf{HL}}_{sub} \sqsubseteq_{\mathsf{L}} \mathsf{P_L}$. Hence we can choose some actual entity $\mathsf{P_L}$ whose relational semantics is a subset of the potentially unhealthy $[\![\mathsf{P_H}]\!]^{\mathsf{HL}}_{sub}$ and we still have a useful refinement guarantee.

Restricting the guarantee for vertical refinement to this special case we get:

> $\mathsf{P_H} \sqsubseteq^{\mathsf{HL}}_{sub} [\![\mathsf{P_H}]\!]^{\mathsf{HL}}_{sub} \sqsubseteq_{\mathsf{L}} \mathsf{P_L}$ **guarantees** the high-level $\mathbb{A}_{sub}$-interpretation of any entity $\mathsf{P_L}$ behaves like (can be observed to have a subset of the observations of) entity $\mathsf{P_H}$ whenever it is placed in any abstract context Ξ_{H} and only the abstract observations $\mathbb{O}_{\mathsf{H}}$ are made.

4.2. Common frame examples for programs and processes

4.2.1. Programs where an abstract data type was expected

Given that non-termination is usually regarded as a bad thing from a transactional view, some people restrict the set of programs considered as valid contexts to those that, when using the abstract data type A, always succeed [29] (i.e. once started, giving the observation *start*, always stop, giving the observation *stop*), where:

$$\Xi_{\mathsf{ass}} \triangleq \{\mathsf{P} \mid O([\mathsf{A}]_{\mathsf{P}}) = \{(start, stop)\}\} \text{ and } \mathbb{O}_{\mathsf{ass}} \triangleq \{(start, stop), (start)\}$$

Note a subtle but vital point here: (*start*) is in the frame as a possible observation, even though it will not be observed for the given set of contexts (by construction). The point is that if (*start*) were not mentioned as a possible (though actually non-occurring) observation then we are being *silent* about this observation, which would leave open the possibility it might subsequently be introduced as the system is further developed. The fact that we have said it is a possibility but then not used it fixes the fact that it will never occur, as required.

4.2.2. Processes with compatible alphabets

Lotos's extension refinement [4], which allows refinement by adding new operations[4], can be formalised using a semantics where the contexts are restricted to be processes X that attempt to communicate with a process A only via its alphabet $\alpha(\mathsf{A})$ [29] (i.e. X never uses an operation outside of A's alphabet) and the observation function is unchanged:

$$\Xi_{\alpha(\mathsf{A})} \triangleq \{\mathsf{X} \mid \alpha(\mathsf{X}) = \alpha(\mathsf{A})\}$$

As the frame is defined in terms of the abstract process and all processes on one layer have the same frame, the concrete process resulting from extension refinement will be on a different layer to the abstract process because its frame is different (because it is extended). But for extension refinement we have $\mathit{Frame}_{\mathsf{A}} \subseteq \mathit{Frame}_{\mathsf{C}}$ and consequently we always have the necessary subset morphism $\sqsubseteq^{\mathsf{AC}}_{sub}$, so:

$$\mathsf{A} \sqsubseteq^{\mathsf{AC}}_{sub} \mathsf{C}$$

[4]Lotos allows 'standard', i.e. failures, refinement within a layer.

4.2.3. Component states with compatible alphabets; deterministic components

The next two examples are more recent and are an attempt to "provide a more expressive way of describing component interfaces" [17]. Neither provide a testing semantics and both have some explicit control over the contexts in which they can be placed.

The usual CSP-style interpretation of an action a not being enabled at some state n (i.e. $n \overset{\mathsf{a}}{\not\longrightarrow}$) is a **guarantee** that any process trying to execute an action is blocked. An alternative interpretation is adopted in Interface Automata [1], where a not enabled action defines an **assumption** that no context tries to execute the action while the process is in the state from which the action is not enabled. We can capture this interpretation by defining $Frame_{IA}$ where the contexts satisfy the assumption

$$\Xi_{IA} \triangleq \{\mathsf{X} \mid \forall\, nm\rho.\mathsf{X} \xrightarrow{\rho} n \wedge \mathsf{A} \xrightarrow{\rho} m \Rightarrow \pi(n) \subseteq \pi(m)\}^{5}$$

The definition of refinement of Interface Automata [1] is called *alternating simulation* and is not based on a testing semantics and is different in its detail from our testing-based semantics.

Interface Input Output Automata [17] use two automata to define the interface: one to define the environment *Env* (context) and one to define the system *Sys* (entity). They also base their definition of refinement and implementation on the subset of prefixed closed traces relation and thus limit themselves to safety-only properties. To capture this we define the observation function to return the set of prefixed closed traces (not just the complete traces).

5. Vertical refinement between event-based LTS

As a concrete example of vertical refinement we implement the IBP layer in the broadcast layer. What is particularly interesting about this is that we can find no way to extend this to be able to implement handshake on broadcast. The problem appears when considering the same processes that cause problems with the definition of determinism.

5.1. From Restriction and Hiding to subset morphisms

Restriction and hiding in the process literature refer to functions that remove events from a process and can be viewed as "abstraction" functions.

[5] The function π maps a node to the set of nodes next reachable from it.

Having defined the observational semantics of processes ACP models Restriction as a function renaming events to certain "δ events", here called δ-abstraction, and Hiding as a function renaming events to τ events, here called τ-abstraction.

The abstraction functions, Restriction and Hiding, can simply be applied to a process to remove events from a concrete process whenever the developer chooses. But here, as in [4, 11], we are interested in reversing this process and introducing these events to the abstract process, and thus creating the concrete process. Further, and crucially for our example, we are interested in viewing the introduction of these new events as a formal refinement step rather than an informal design decision.

We reverse the τ-abstraction and δ-abstraction (Definition 16 in the Appendix) by extending refinement to introduce events in two quite separate ways [29, 26].

Definition 8. *δ–refinement and τ–refinement. For LTS* A *and* C *:*

$$\mathsf{A} \sqsubseteq_{\Xi\delta Del} \mathsf{C} \triangleq \mathsf{A} \sqsubseteq_{\Xi} \mathsf{C}\delta_{Del}$$
$$\mathsf{A} \sqsubseteq_{\Xi\tau Hid} \mathsf{C} \triangleq \mathsf{A} \sqsubseteq_{\Xi} \mathsf{C}\tau_{Hid}$$

Firstly if δ–refinement holds, i.e. $\mathsf{A} \sqsubseteq_{\Xi\delta Del} \mathsf{C}$ and $\alpha(\Xi_{\mathsf{A}}) \cap Del = \varnothing$, then events are introduced that were previously not observable and always blocked. This would be used, for example, to refine a specification that defined successful behaviour and assumed error events, in set Del, never occurred.

Secondly if τ–refinement holds, i.e. $\mathsf{A} \sqsubseteq_{\Xi\tau Hid} \mathsf{C}$ and $\alpha(\mathsf{A}) \cap Hid = \varnothing$, then events are introduced that were previously not observable and never blocked in the more abstract view.

Clearly the guarantee from the subset refinement applies in both these cases.

5.2. Refining the $(\mathsf{T}_{\mathsf{IBP}}, \sqsubseteq_{\mathsf{IBP}})$ *layer into the* $(\mathsf{T}_{\mathsf{BC}}, \sqsubseteq_{\mathsf{BC}})$ *layer*

In this section we will define a particular vertical refinement between high-level IBP entities and low-level broadcast processes. We will then show that we have been unable to extend the high-level layer to all handshake processes. An explanation can be found by considering the way handshake processes have abstracted away the *cause* and *response* nature of event synchronisation.

Definition 9. *Let* A *be an LTS* $(N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$.

$$[\![\mathsf{A}]\!]_B \triangleq M_{BC}(N_{[\![\mathsf{A}]\!]_B}, s_{\mathsf{A}}, T_{[\![\mathsf{A}]\!]_B})$$

$$N_{[\![\mathsf{A}]\!]_B} \triangleq N_{\mathsf{A}} \cup \{n_t \mid t \in T_{\mathsf{A}}\} \cup \{n_{(m,\mathsf{a})} \mid m \in N_{\mathsf{A}} \wedge m \overset{\mathsf{a}}{\not\longrightarrow}\}$$

$$\begin{aligned} T_{[\![\mathsf{A}]\!]_B} \triangleq \quad & \{s \xrightarrow{\mathsf{tx!}} z, z \xrightarrow{\mathsf{rx?}} s, z \xrightarrow{\mathsf{ax?}} t \mid s \xrightarrow{\overline{\mathsf{x}}} t \wedge z = n_{s \xrightarrow{\overline{\mathsf{x}}} t}\} \cup \\ & \{s \xrightarrow{\mathsf{tx?}} z, z \xrightarrow{\mathsf{ax!}} t \mid s \xrightarrow{\mathsf{x}} t \wedge z = n_{s \xrightarrow{\mathsf{x}} t}\} \cup \\ & \{s \xrightarrow{\mathsf{tx?}} z, z \xrightarrow{\mathsf{rx!}} s \mid s \overset{\mathsf{x}}{\not\longrightarrow} \wedge z = n_{(s,\mathsf{x})}\} \end{aligned}$$

We map an active high-level event such as $\overline{\mathsf{b}}$ (see Fig. 8) into three parts. The try event $\mathsf{tb!}$ is performed, subsequently either aborting ($\mathsf{rb?}$) if the context cannot interact on b, or succeeding ($\mathsf{ab?}$) if the context can interact on b. The mapping for the passive event b can be seen in right-hand side of Fig. 8.

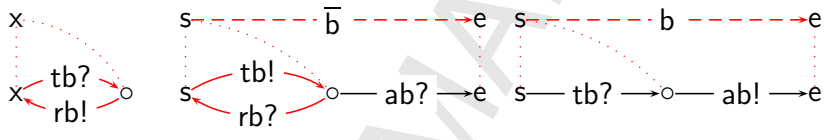

Figure 8: Mapping from high to low using $[\![\_]\!]_B$

Our semantic mapping $[\![\_]\!]_B$ from a high-level layer to a low-level layer will not only map events $\overline{\mathsf{b}}$ and b to different processes but will also add try-reject loops $\mathsf{tb?rb!}$ wherever a passive event b cannot be performed, i.e. when $\mathsf{b} \notin \pi(\mathsf{x})$ (see left-hand side of Fig. 8).

Although we see this as the natural solution, because of the addition of the try-reject loops it is neither an action refinement nor indeed an instance of vertical implementation [31].

We need some care in interpreting the events of Fig. 8. In particular both handshake events $\overline{\mathsf{b}}$ and b are able to be blocked but the broadcast events $\mathsf{tb!}$,$\mathsf{rb!}$ and $\mathsf{ab!}$ are not.

Not all the processes $(N_{[\![\mathsf{A}]\!]_B}, s_{\mathsf{A}}, T_{[\![\mathsf{A}]\!]_B})$ are valid broadcast processes, i.e. they are not all in T_{BC}. For this reason we have applied M_{BC}. For ease of understanding we have not shown the events added by M_{BC} in Fig. 8.

Next we define the abstraction $\mathbb{A}_B$. It should be noted that $\mathsf{tx?}$ events are replaced by two τ events, one each way.

Definition 10. *Let* A *be an LTS* $(N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$.

$$\mathbb{A}_B(\mathsf{A}) \triangleq (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathbb{A}_B(\mathsf{A})})$$

$$\begin{aligned} T_{\mathbb{A}_B(\mathsf{A})} \triangleq \quad & \{s \xrightarrow{\overline{\mathsf{x}}} t \mid s \xrightarrow{\mathsf{ax?}} t\} \cup \{s \xrightarrow{\mathsf{x}} t \mid s \xrightarrow{\mathsf{ax!}} t\} \cup \\ & \{s \xrightarrow{\tau} t \mid s \xrightarrow{\mathsf{tx!}} t \vee s \xrightarrow{\mathsf{rx!}} t \vee s \xrightarrow{\mathsf{rx?}} t \vee s \xrightarrow{\tau} t \vee s \xrightarrow{\mathsf{tx?}} t \vee t \xrightarrow{\mathsf{tx?}} s\} \end{aligned}$$

We will need some lemmas in what follows:

Lemma 2. *For any vertical refinement* $\sqsubseteq_v^{\mathsf{HL}}$, $[\![\_]\!]_v$ *and* $\mathbb{A}_v$ *are adjoint if and only if* $[\![\_]\!]_v$ *and* $\mathbb{A}_v$ *are monotone and* $[\![\mathbb{A}_v(\_)]\!]_v \sqsubseteq_{\mathsf{L}} id_{\mathsf{L}}$ *and* $id_{\mathsf{H}} \sqsubseteq_{\mathsf{H}} \mathbb{A}_v([\![\_]\!]_v)$

Proof [34, p151]. •

Lemma 3. *For any entities* A *and* C *in* T_{BC}*, if* $Tr^c(\mathsf{C}) \subseteq Tr^c(\mathsf{A})$ *then* $Tr^c(\mathbb{A}_B(\mathsf{C})) \subseteq Tr^c(\mathbb{A}_B(\mathsf{A}))$.

Similarly, For any entities A *and* C *in* $\mathsf{T}_{\mathsf{IBP}}$*, if* $Tr^c(\mathsf{C}) \subseteq Tr^c(\mathsf{A})$ *then* $Tr^c([\![\mathsf{C}]\!]_B) \subseteq Tr^c([\![\mathsf{A}]\!]_B)$*. (See the appendix for a definition of* Tr^c*.)*

•

Lemma 4. *Let* N *be a set of low-level* BC *events and let* $\mathbb{A}_B(N)$ *be the set of observable high-level events built from them by applying* $\mathbb{A}_B$ *to each event in turn. Then, for any* $\mathsf{X} \in \mathsf{T}_{\mathsf{BC}}$*:*

$$Tr^c(\mathbb{A}_B(\mathsf{C} \|_N \mathsf{X})) = Tr^c(\mathbb{A}_B(\mathsf{C}) \|_{\mathbb{A}_B(N)} \mathbb{A}_B(\mathsf{X})).$$

Similarly, let N *be a set of high-level* IBP *events and let* $[\![N]\!]_B$ *be the set of observable low-level events built from them by applying* $[\![\_]\!]_B$ *to each event in turn. Then, for any* $\mathsf{X} \in \mathsf{T}_{\mathsf{IBP}}$*:*

$$Tr^c([\![\mathsf{C} \|_N \mathsf{X}]\!]_B) = Tr^c([\![\mathsf{C}]\!]_B \|_{[\![N]\!]_B} [\![\mathsf{X}]\!]_B).$$

•

Theorem 1. *Semantic mappings* $\mathbb{A}_B$ *and* $[\![\_]\!]_B$ *define a vertical refinement* $\sqsubseteq_B^{\mathsf{IBP\ BC}}$ *from the handshake layer (with its refinement* $\sqsubseteq_{\Xi_{\mathsf{IBP}}, Tr^c}$ *within the layer) to the broadcast layer (with its refinement* $\sqsubseteq_{\Xi_{\mathsf{BC}}, Tr^c}$ *within the layer).*

Proof. We will prove our result using Lemma 2. To do this we must prove that $\mathbb{A}_B$ and $[\![\_]\!]_B$ are monotone and $[\![\mathbb{A}_B(\_)]\!]_B \sqsubseteq_{\mathsf{BC}} id_{\mathsf{BC}}$ and $id_{\mathsf{IBP}} \sqsubseteq_{\mathsf{IBP}} \mathbb{A}_B([\![\_]\!]_B)$.

First we prove that $\mathbb{A}_B$ and $[\![\_]\!]_B$ are monotone.

Monotonicity: $\forall\, \mathsf{P_{BC}}, \mathsf{Q_{BC}} \in \mathsf{T_{BC}}.\ \mathsf{P_{BC}} \sqsubseteq_{\mathsf{BC}} \mathsf{Q_{BC}} \Rightarrow \mathbb{A}_B(\mathsf{P_{BC}}) \sqsubseteq_{\mathsf{IBP}} \mathbb{A}_B(\mathsf{Q_{BC}})$

Consider arbitrary $\mathsf{P_{BC}}, \mathsf{Q_{BC}} \in \mathsf{T_{BC}}$ and assume $\mathsf{P_{BC}} \sqsubseteq_{\mathsf{BC}} \mathsf{Q_{BC}}$

$\forall\, x \in \Xi_{\mathsf{BC}}.\ Tr^c([\mathsf{Q_{BC}}]_x) \subseteq Tr^c([\mathsf{P_{BC}}]_x)$ Defn. of general refinement

$\forall\, x \in \Xi_{\mathsf{BC}}\ Tr^c(\mathbb{A}_B([\mathsf{Q_{BC}}]_x)) \subseteq Tr^c(\mathbb{A}_B([\mathsf{P_{BC}}]_x))$ From Lemma 3

$\forall\, x \in \Xi_{\mathsf{BC}}\ Tr^c([\mathbb{A}_B(\mathsf{Q_{BC}})]_{\mathbb{A}_B(x)}) \subseteq Tr^c([\mathbb{A}_B(\mathsf{P_{BC}})]_{\mathbb{A}_B(x)})$ From Lemma 4

$\forall\, y \in \Xi_{\mathsf{IBP}}\ Tr^c([\mathbb{A}_B(\mathsf{Q_{BC}})]_y) \subseteq Tr^c([\mathbb{A}_B(\mathsf{P_{BC}})]_y)$ $\mathbb{A}_B$ is surjective

$\therefore\ \mathbb{A}_B(\mathsf{P_{BC}}) \sqsubseteq_{\mathsf{IBP}} \mathbb{A}_B(\mathsf{Q_{BC}})$.

$\mathsf{P_{IBP}} \sqsubseteq_{\mathsf{IBP}} \mathsf{Q_{IBP}} \Rightarrow [\![\mathsf{P_{IBP}}]\!] \sqsubseteq_{\mathsf{BC}} [\![\mathsf{Q_{IBP}}]\!]$ is similar.

Consider arbitrary $\mathsf{P_{IBP}}, \mathsf{Q_{IBP}} \in \mathsf{T_{IBP}}$ and assume $\mathsf{P_{IBP}} \sqsubseteq_{\mathsf{IBP}} \mathsf{Q_{IBP}}$

$\forall\, x \in \Xi_{\mathsf{IBP}}\ Tr^c([\mathsf{Q_{IBP}}]_x) \subseteq Tr^c([\mathsf{P_{IBP}}]_x)$ Defn. of general refinement

$\forall\, x \in \Xi_{\mathsf{IBP}}\ Tr^c([[\mathsf{Q_{IBP}}]_x]\!]_B) \subseteq Tr^c([[\mathsf{P_{IBP}}]_x]\!]_B)$ From Lemma 3

$\forall\, x \in \Xi_{\mathsf{IBP}}\ Tr^c([[\mathsf{Q_{IBP}}]\!]_B]_{[x]\!]_B}) \subseteq Tr^c([[\mathsf{P_{IBP}}]\!]_B]_{[x]\!]_B})$ From Lemma 4

$\forall\, y \in \Xi_{\mathsf{BC}}\ Tr^c([[\mathsf{Q_{IBP}}]\!]_B]_y)) \subseteq Tr^c([[\mathsf{P_{IBP}}]\!]_B]_y))$ $[\![\_]\!]_B$ is surjective

We must prove $[\![\mathbb{A}_B(\_)]\!]_B \sqsubseteq_{\mathsf{BC}} id_{\mathsf{BC}}$ and $id_{\mathsf{IBP}} \sqsubseteq_{\mathsf{IBP}} \mathbb{A}_B([\![\_]\!]_B)$ but actually we prove the stronger results $[\![\mathbb{A}_B(\_)]\!]_B = id_{\mathsf{BC}}$ and $id_{\mathsf{IBP}} = \mathbb{A}_B([\![\_]\!]_B)$.

To do this we prove $\mathsf{P_{IBP}} = \mathbb{A}_B([\![\mathsf{P_{IBP}}]\!]_B)$, that is to say these LTSs are isomorphic. There is clearly a bijective mapping from nodes to nodes as $[\![\_]\!]_B$ first adds some new nodes and $\mathbb{A}_B$ removes them. This bijection between the nodes maps start node to start node.

That there is a bijection between the transitions can be seen by case analysis:

Case 1. the $\mathsf{tx?rx!}$ loop is first added by $[\![\_]\!]_B$ and removed by $\mathbb{A}_B$.

Case 2. the $\mathsf{tx?rx?}$ loop and $\mathsf{ax?}$ replace an $\overline{\mathsf{b}}$ action by application of $[\![\_]\!]_B$ and this is then reversed by $\mathbb{A}_B$.

Case 3. the $\mathsf{tx!ax!}$ transitions replace an b action by application of $[\![\_]\!]_B$ and this is then reversed by $\mathbb{A}_B$.

From these three cases we have $id_{\mathsf{IBP}} = \mathbb{A}_B([\![\_]\!]_B)$.

Applying $[\![\_]\!]_B$ to above gives $[\![\_]\!]_B = [\![\mathbb{A}_B([\![\_]\!]_B)]\!]_B$. Then because we restrict the low-level to be in the range of $[\![\_]\!]_B$ we have $[\![\mathbb{A}_B(\_)]\!]_B = id_{\mathsf{BC}}$. •

5.3. Vertical refinement failure—and success

In this section we look at two examples using the two vertical refinements introduced in the previous sections.

5.3.1. Using $\sqsubseteq_B$

We take the vertical refinement $\sqsubseteq_B$, defining how to refine IBPs into broadcast processes, as being correct by construction. But, we find that we cannot expand IBP to all processes as defined by CSP/CCS etc. as is illustrated by returning to the example from a previous section (Fig. 5) and reproduced, as IBP processes, here in Fig. 9. Recall that we described Rob as non-deterministic (even though according to the usual process algebra definitions it is not recognised as such) and here our "implementation" on a broadcast layer will also have its version $[\![\mathsf{Rob}]\!]_B$ of Rob as non-deterministic, as we can see in Fig. 10,[6] where $[\![\mathsf{Rob}]\!]_B$ is a non-deterministic 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 non-deterministic.

Process $\mathsf{Robot}_\mathsf{L}$ in the BC layer is a refinement of $[\![\mathsf{Rob}]\!]_B$ that will try button b1 only, and this refinement is possible because in broadcast communication this move is a removal of non-determinism. Note, then, that we have a vertical refinement between Rob in IBP and $\mathsf{Robot}_\mathsf{L}$ in BC. No such refinement relation exists between Rob in PA and $\mathsf{Robot}_\mathsf{L}$ in BC because although $\mathsf{Robot}_\mathsf{H} \sqsubseteq_B \mathsf{Robot}_\mathsf{L}$ we know that $\mathsf{Rob} \not\sqsubseteq_{\mathsf{PA}} \mathsf{Robot}_\mathsf{H}$. This lack happens whenever the notion of determinism in a layer differs from our notion *Gen-Det*, based on Milner's understanding of determinism.

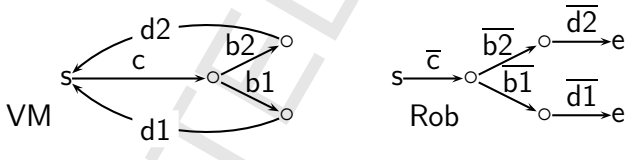

Figure 9: Are VM and Rob deterministic?

5.3.2. Using $\sqsubseteq_v$

Let us assume we wish to stepwise refine our model to formalise the design decision that the vending machine only has two cups and that when out of

[6]So as to keep the lower level diagrams small we have expanded only the high-level events b1! and b2!. The expansion of the other events is obvious from Fig. 8.

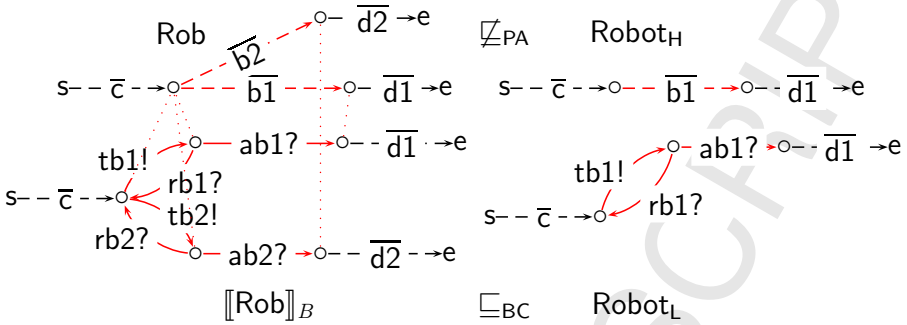

Figure 10: $[\![\mathsf{Rob}]\!]_B \sqsubseteq_{\mathsf{BC}} \mathsf{Robot_L}$ but $\mathsf{Rob} \not\sqsubseteq_{\mathsf{PA}} \mathsf{Robot_H}$ and $M_{IBP}(\mathsf{Rob}) \sqsubseteq_{\mathsf{IBP}} \mathsf{Robot_H}$

cups it responds to further requests with error events that are broadcast not handshake events.

First the vending machine VM in Fig. 9 is defined with handshake interactions. This can be vertically refined into an entity with broadcast interactions, VMv in Fig. 11.

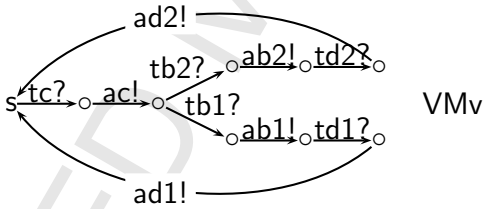

Figure 11: (Fig. 9) $\mathsf{VM} \sqsubseteq_v \mathsf{VMv}$

Secondly we add an error event, the "return of the coin". This event is to occur if a button is pushed but the vending machine has none of the required drink left. But since we do not wish this error event to be blocked by a user (robot), it must be under *local control*. Thus the return of the coin event is a broadcast event cr!.

This step is formalised by a δ-refinement, as discussed in Section 5.1, to give VMvd in Fig. 12.

A more compact way to view this process is VMb in Fig. 13 where the original handshake events are shown with the newly visible broadcast event cr!. We could formalise this by defining LTS with four types of event but

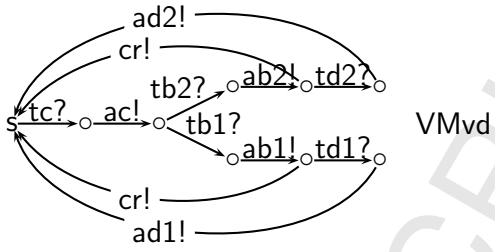

Figure 12: (Fig. 11) VMv $\sqsubseteq_{BC\delta\{\mathsf{cr}\}}$ VMvd

here we simply view VMb as "sugar" for VMvd in Fig. 12 and leave the reader to expand the dashed lines in Fig. 13.

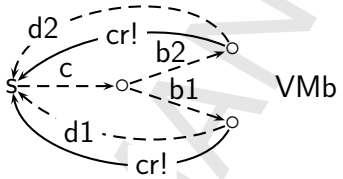

Figure 13: (Fig. 9) VM $\sqsubseteq_{BC\delta\{\mathsf{cr}\}}$ VMb

Having made visible the return of coin event we now have an entity that is non-deterministic, as you can never tell if the result of pushing a button will be to dispense a drink or return the coin. More technically, the events cr! and td2? both leave the same node.

We can easily refine this specification to model a vending machine which can vend a total of two drinks only, i.e. d1 and then d2 or d2 and then d1, thus giving Fig. 14.

6. Conclusions

We have seen how considering a system as made up from contexts for entities and observations of the behaviour of those entities in those contexts can lead to insights, as does consideration of the interfaces between those three elements of a system.

Also, refinement as presented in this paper is parameterised on the set of contexts an entity can be placed in and on the observations that can be

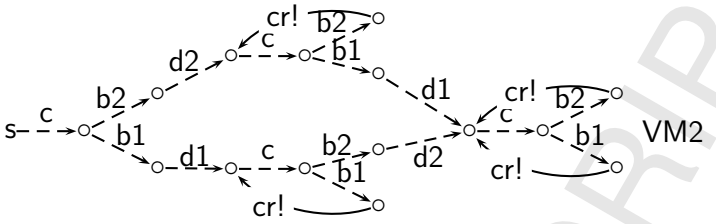

Figure 14: (Fig. 12) VMvd $\sqsubseteq_{BC}$ VM2

made by a user of the system thus formed. Since refinement turns out to have sets of contexts Ξ as a parameter, by changing Ξ we were able to model different types of interaction [29].

We continued the story with a generalisation, which we call *vertical refinement*, of what, in the literature, has been called action refinement or non-atomic refinement. The traditional view of refinement is that it all happens in one *layer* or logical theory. Vertical refinement allows movement (i.e. design steps) between different layers, where each layer may contain different styles of event-interaction, or may introduce new events or states. As an example we defined vertical refinement from a "handshake layer" to a "broadcast layer", and elsewhere (e.g. [27, 28]) we have shown how the design step of adding error handling or new events can be described formally. This formalisation between handshake and broadcast layers brings out different assumptions made about determinism and this issue has had to be addressed too.

Acknowledgements

We would like to thank the many people who have discussed the ideas presented in this paper over many years—you know who you are! In particular, though, we give thanks to Lindsay Groves, John Derrick, Jim Woodcock, Jim Davies, Eerke Boiten and Mark Utting.

References

[1] Luca Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.

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

[3] Christie Bolton and Jim Davies. A singleton failures semantics for Communicating Sequential Processes. Research Report PRG-RR-01-11, Oxford University Computing Laboratory, 2001.

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

[5] R. de Nicola and M Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34, 84.

[6] John Derrick and Eerke Boiten. Non-atomic refinement in Z. In J.M. Wing, J.C.P. Woodcock, and J. Davies, editors, *FM'99 World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes in Computer Science*, pages 1477–1496, Berlin, September 1999. Springer.

[7] John Derrick and Eerke Boiten. Relational concurrent refinement. *Formal Aspects of Computing*, 15(2 - 3):182–214, 2003.

[8] Steve Dunne and Stacey Conroy. Process refinement in B. In Helen Treharne, Steve King, Martin C. Henson, and Steve Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users*, volume 3455 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2005.

[9] C. Ene and T. Muntean. Expressiveness of Point-to-Point versus Broadcast Communications. In *Fundamentals of Computation Theory, 12th International Symposium FCT'99*, volume 1684 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

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

[11] Clemens Fischer and Heike Wehrheim. Behavioural subtyping relations for object-oriented formalisms. *Lecture Notes in Computer Science*, 1816:469–483, 2000.

[12] R. Gorrieri and A. Rensink. Action refinement. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 1047–1147. Elsevier, 2001.

[13] M Hennessy. *Algebraic Theory of Processes*. The MIT Press, 1988.

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

[15] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[16] R. Kumar and M. Heymann. Masked prioritized synchronization for interaction and control of discrete event systems. *IEEE Transactions on Automatic Control*, 45(11):1970–1982, Nov 2000.

[17] Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. Interface input/output automata. In *International Symposium on Formal Methods*, volume 4085 of *LNCS*, 2006.

[18] N. Lynch and R. Segala. A Comparison of Simulation Techniques and Algebraic Techniques for Verifying Concurrent Systems. *Formal Aspects of Computing Journal*, 7(3):231–265, 1995.

[19] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, pages 2(3):219–246, 1989.

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

[21] R. De Nicola and M Hennessy. CCS without τs. *LNCS 250*, pages 138–151, 92.

[22] K. V. S. Prasad. A calculus of value broadcasts. In *Parallel Architectures and Languages Europe*, pages 391–402, 1993.

[23] K. V. S. Prasad. A calculus of broadcasting systems. *Science of Computer Programing*, 25((2-3)):285–327, 1995.

[24] S. Reeves and D. Streader. Atomic Components. Technical report, University of Waikato, http://hdl.handle.net/10289/1002, 2004. Computer Science Technical Report 01/2004, http://www.cs.waikato.ac.nz/~dstr.

[25] S. Reeves and D. Streader. Constructing programs or processes. *Journal of Universal Computer Science*, 11(12):2034–2045, December 2005.

[26] S. Reeves and D. Streader. Liberalising Event B without changing it. Technical report, University of Waikato, 2006. Computer Science Working Paper Series 07/2006, ISSN 1170-487X.

[27] S. Reeves and D. Streader. Feature refinement. *Fifth IEEE International Conference on Software Engineering and Formal Methods SEFM*, pages 371–380, 2007.

[28] S. Reeves and D. Streader. Generics tools via general refinement. In *Proceedings of TTSS, Electronic Notes in Theoretical Computer Science*, 2007.

[29] Steve Reeves and David Streader. Comparison of data and process refinement. In J.C.P. Woodcock and J.S. Dong, editors, *Proceedings of ICFEM 2003*, number 2885 in Lecture Notes in Computer Science, pages 266–285. Springer-Verlag, 2003.

[30] Steve Reeves and David Streader. A Robust Semantics Hides Fewer Errors. In Ana Cavalcanti and Dennis Dams, editors, *FM2009: Formal Methods*, volume 5850 of *LNCS*, pages 499–515. Springer-Verlag, 2009.

[31] Arend Rensink and Roberto Gorrieri. Vertical implementation. *Information and Computation*, 170:95–133, 2001. Extended version of "Vertical Bisimulation" (TAPSOFT '97). Full report version: Hildesheimer Informatik-Bericht 9/98, University of Hildesheim.

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

[33] R. Segala. A Process Algebraic View of I/O Automata. Technical Report MIT/LCS/TR-557, Massachusetts Institute of Technology, 1992.

[34] P. Taylor. *Practical Foundations of Mathematics*. Cambridge University Press, 1999. Cambridge studies in advanced mathematics 59.

[35] D. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, Faculteit der Informatica, 1992.

[36] Frits W. Vaandrager. On the relationship between process algebra and input/output automata. In *Logic in Computer Science*, pages 387–398, 1991.

[37] A. Valmari and M. Tienari. An improved failure equivalence for finite-state systems with a reduction algorithm. In *Protocol Specification, Testing and Verification*, IFIP XI. North-Holland, 1991.

[38] Antti Valmari and Martti Tienari. Compositional Failure-based Semantics Models for Basic LOTOS. *Formal Aspects of Computing*, 7(4):440–468, 1995.

[39] R. J. van Glabbeek. Linear Time-Branching Time Spectrum I. In *CONCUR '90 Theories of Concurrency: Unification and Extension*, LNCS 458, pages 278–297. Springer-Verlag, 1990.

[40] R. L. van Glabbeek. The linear time - branching time spectrum I. the semantics of concrete sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier Science, Amsterdam, The Netherlands, 2001.

Appendix

The basics of operational semantics

In this section we gather together some standard definitions.

We are interested in modelling entities that have been considered as either state-based or event-based. By defining mappings between the state-based operational semantics (relation-based) and the event-based operational semantics (labelled transition system-based) we are free to switch how we view our entities. This correspondence rests upon the usual and simple idea that transitions can be represented as relations (we often see this in finite-state automaton accounts, where the diagrams use transitions and the text uses transition relations).

We assume a universe containing a set of names *Names* that will be used to give names to operations in a state-based system and names to events in an event-based system. A special event τ is introduced that models an event that can neither be seen nor blocked.

First the state-based operational semantics, a relation-based semantics. Interacting entities can be given a state-based semantics by using named relations (which share state and relate the state before an operation takes place to the state after an operation takes place).

Definition 11. *Let Σ_{A} be a state space and $init_{\mathsf{A}}$ a start state. Named partial relational (NPR) semantics A is given by $\mathsf{A} \triangleq (\Sigma_{\mathsf{A}}, init_{\mathsf{A}}, Npr_{\mathsf{A}})$ where $init_A \in \Sigma_{\mathsf{A}}$ and we have a set of named partial relations*

$$Npr_{\mathsf{A}} \subseteq \{(\mathsf{o}, R) \mid \mathsf{o} \in Names \cup \tau \wedge R_{\mathsf{o}} \subseteq \Sigma_{\mathsf{A}} \times \Sigma_{\mathsf{A}}\}$$

Let $Op(\mathsf{A}) \triangleq \{\mathsf{o} \mid \exists\, R.(\mathsf{o}, R) \in N_{\mathsf{A}}\}$ be the set of operation names of NPR semantics A. •

Now we move to event-based operational semantics, a labelled transition system-based semantics. Interacting entities can given an event-based semantics (by labelling a state transition with an event) for process algebras CSP [15, 32], CCS [20], ACP [2], for broadcast systems IOA [19], CBS [23], for abstract data types [3] and for objects [7].

The observable event a can be performed only when the process is executed in a context that includes a parallel process that is also ready to execute a. It is only events that interact at an interface; there is no shared state. But, LTS are open to a variety of interpretations. To define the semantics of entities unambiguously they can be represented by equivalence classes of LTS, where the choice of equivalence relation more accurately captures what the LTS should be interpreted as, e.g. as a handshake entity or as an ADT and so on. In handshake processes the execution of events is not under local (their own) control and they are blocked from execution whenever the context they are in is not ready to execute them.

Definition 12. *Let N_{A} be a finite set of nodes and s_{A} the start node. Labelled transition system (LTS) A is given by $\mathsf{A} \triangleq (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$ where $s_{\mathsf{A}} \in N_{\mathsf{A}}$ and we have a set of transitions*

$$T_{\mathsf{A}} \subseteq \{(n, \mathsf{a}, m) \mid n, m \in N_{\mathsf{A}} \wedge \mathsf{a} \in Names \cup \tau\}$$

Let $\alpha(\mathsf{A}) \triangleq \{\mathsf{a} \mid \exists\, x, y.(x, \mathsf{a}, y) \in T_{\mathsf{A}}\}$ be the alphabet of the LTS A. We write $x \xrightarrow{\mathsf{a}} y$ for $(x, \mathsf{a}, y) \in T_{\mathsf{A}}$ where A is obvious from context and refer to event a as being enabled *in state x. We write $n \xrightarrow{\mathsf{a}}$ for $\exists\, m.(n, \mathsf{a}, m) \in T_{\mathsf{A}}$.* •

To take account of τ actions being unobservable we define the observational semantics $\Longrightarrow$. One small point to notice is that when we consider event-based models, where we are interested in a testing semantics where entities and components are composed by parallel composition, we need semantics no finer than failures semantics. Consequently our observational semantics is free to remove all non-looping τ actions (see [24] for details). The removal of τ loops is treated in various different ways in the literature. CSP [32] takes the view that they must be removed and hence models them as having divergent behaviour. CCS [20] views choice as fair and hence loops can at most cause a delay and as CCS is untimed they can safely be removed. Finally NDFD and CFFD [37, 38] treat choice as unfair but, unlike CSP, do not model such loops as having divergent behaviour (see [24] for more details). Here our observational semantics is based on the CCS model as this is the most appropriate for broadcast operations. But Section 2.2 is implicitly based upon the unfair interpretation of divergence, so we have followed the view of NDFD and the operational semantics from [24].

Definition 13. $\Longrightarrow_o$ *is a predicate where:*

$$s_1 \stackrel{\tau}{\Longrightarrow} s_n \stackrel{\text{def}}{=} s_0 \stackrel{\tau}{\longrightarrow} s_1, s_1 \stackrel{\tau}{\longrightarrow} s_2, \ldots s_{n-1} \stackrel{\tau}{\longrightarrow} s_n \vee s_1 = s_n$$

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

Traces: $Tr(\mathsf{A}) \stackrel{\text{def}}{=} \{\rho \mid \exists\, m. s_{\mathsf{A}} \stackrel{\rho}{\Longrightarrow}_o m\}$

Complete traces: $Tr^c(\mathsf{A}) \stackrel{\text{def}}{=} \{\rho \mid s_{\mathsf{A}} \stackrel{\rho}{\Longrightarrow}_o n \wedge (\pi(n) = \varnothing \vee \mid \rho \mid = \infty)\}$[7]

To make the point that LTS and NPR are essentially the same, we will define a translation *lts* from relation-based semantics to LTS and its inverse *npr*. As we previously stated both operational semantics are open to many different interpretations so we view them as giving just part of the semantic story (completed by giving contexts and observations). By defining the translation between state-based systems and event-based systems on the operational semantics we have not restricted ourselves to a particular interpretation of the operational semantics.

Definition 14.

$$lts((\Sigma_{\mathsf{A}}, init_{\mathsf{A}}, Npr_{\mathsf{A}})) \triangleq (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$$

[7] The function π maps a node to the set of nodes next reachable from it.

where $N_{\mathsf{A}} \triangleq \Sigma_{\mathsf{A}}$, $s_{\mathsf{A}} \triangleq init_{\mathsf{A}}$ *and*

$$T_{\mathsf{A}} \triangleq \{(x, n, y) \mid (n, R) \in Npr_{\mathsf{A}} \wedge (x, y) \in R\}$$

Also:

$$npr((N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})) \triangleq (\Sigma_{\mathsf{A}}, init_{\mathsf{A}}, Npr_{\mathsf{A}})$$

where $\Sigma_{\mathsf{A}} \triangleq N_{\mathsf{A}}$, $init_{\mathsf{A}} \triangleq s_{\mathsf{A}}$ *and*

$$Npr_{\mathsf{A}} \triangleq \{(n, R) \mid x \xrightarrow{n} y \in T_{\mathsf{A}} \Leftrightarrow (x, y) \in R\}$$

•

Parallel composition is defined to represent the point-to-point private communication between concurrent entities.

Definition 15. *Parallel composition of* $\mathsf{A} = (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$ *and* $\mathsf{B} = (N_{\mathsf{B}}, s_{\mathsf{B}}, T_{\mathsf{B}})$*: for* $S \subseteq Names$, $N_{\mathsf{A}\|_S\mathsf{B}} \triangleq N_{\mathsf{A}} \times N_{\mathsf{B}}$, $s_{\mathsf{A}\|_S\mathsf{B}} = (s_{\mathsf{A}}, s_{\mathsf{B}})$ *and* $T_{\mathsf{A}\|_S\mathsf{B}}$ *is defined as follows.*

Let $\mathsf{x} \in Names \cup \tau$:

$$\frac{n \xrightarrow{\mathsf{x}}_{\mathsf{A}} l, \mathsf{x} \notin S}{(n, m) \xrightarrow{\mathsf{x}}_{\mathsf{A}\|_S\mathsf{B}} (l, m)} \qquad \frac{n \xrightarrow{\mathsf{x}}_{\mathsf{B}} l, \mathsf{x} \notin S}{(m, n) \xrightarrow{\mathsf{x}}_{\mathsf{A}\|_S\mathsf{B}} (m, l)}$$

$$\frac{n \xrightarrow{\mathsf{a}}_{\mathsf{A}} l, m \xrightarrow{\mathsf{a}}_{\mathsf{B}} k, \mathsf{a} \in S}{(n, m) \xrightarrow{\tau}_{\mathsf{A}\|_S\mathsf{B}} (l, k)} \qquad \mathsf{A} \parallel_S \mathsf{B} \triangleq (N_{\mathsf{A}\|_S\mathsf{B}}, s_{\mathsf{A}\|_S\mathsf{B}}, T_{\mathsf{A}\|_S\mathsf{B}}) \quad \bullet$$

Note that our definition of the entity/context interface requires synchronisation on *Act*, all possible events in the entity.

τ–Abstraction and δ–Abstraction

In process algebra, events can be abstracted from a process in two distinct ways. In CCS these ways are *restriction* and *hiding*. Here we will use the ACP special events δ and τ to define the two distinct ways δ–abstraction and τ–abstraction to abstract events.

Definition 16. *δ–abstraction and τ–abstraction. Given LTS* $\mathsf{A} = (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$ *and* $Del \subseteq Names \cup \overline{Names}$ *we have:*

$$\mathsf{A}\delta_{Del} \triangleq (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}\delta_{Del}})$$

where, for all $\mathsf{x} \in Names \cup \overline{Names}$, $T_{\mathsf{A}\delta_{Del}}$ *is defined by:*

$$\frac{n \xrightarrow{\mathsf{x}}_{\mathsf{A}} l, \mathsf{x} \notin Del}{n \xrightarrow{\mathsf{x}}_{\mathsf{A}\delta_{Del}} l}$$

Let $Hid \subseteq Names \cup \overline{Names}$ *and*

$$\mathsf{A}\tau_{Hid} \triangleq Abs(N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}\tau_{Hid}})$$

where for all $\mathsf{x} \in Names \cup \overline{Names}$, $T_{\mathsf{A}\tau_{Hid}}$ *is defined by:*

$$\frac{n \xrightarrow{\mathsf{x}}_{\mathsf{A}} l, \mathsf{x} \notin Hid}{n \xrightarrow{\mathsf{x}}_{\mathsf{A}\tau_{Hid}} l} \qquad \frac{n \xrightarrow{\mathsf{x}}_{\mathsf{A}} l, \mathsf{x} \in Hid}{n \xrightarrow{\tau}_{\mathsf{A}\tau_{Hid}} l}$$

Research Highlights:

- We show how considering interfaces of entities, contexts and users gives clarity
- We show how it relates to characterisations of systems via a notion of testing
- We give examples of five refinement relations as instances of a general model
- An example is refinement between broadcast and interactive branching programs
- Refinement moves from abstract to concrete, preserving certain valuable guarantees