Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 214 (2008) 309–329
www.elsevier.com/locate/entcs

General Refinement, Part Two: Flexible Refinement

Department of Computer Science
University of Waikato
Hamilton, New Zealand

Abstract

In the previous, companion, paper [13] to this paper we introduced our general model of refinement, discussed ideas around determinism and interfaces that the general definition raised, and gave several examples showing how the general definition could be specialised to the sorts of refinement we see in the literature.
In this paper we continue the story and we define vertical refinement on our general model. Vertical refinement can be seen as a generalisation of what, in the literature, has been called action refinement or non-atomic refinement. Alternatively, by viewing a special model (from the previous paper) as a logical theory, vertical refinement can be seen as a theory morphism, formalised as a Galois connection.
We give an example of the utility of this definition by constructing a vertical refinement between broadcast processes and interactive branching programs, and we see how interactive branching programs can be implemented on a platform which provides broadcast communication.
We also show how developments that fall outside the usual, special theories of refinement can be brought into the refinement world by giving examples of development which were thought not to be possible using refinement.
Throughout, the central, simple idea of refinement as a development process that moves from abstract to concrete while preserving certain valuable guarantees will guide us.

Keywords: general refinement, Galois connection, vertical refinement

1 Introduction

In this paper (in Section 3) we introduce a definition of refinement that, amongst other things, permits the refinement of the interpretation of the oper-

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

doi:10.1016/j.entcs.2008.06.014

ational semantics of the entities under consideration. Thus the interpretations that are usually fixed can now be changed during the development of a system by formal refinement steps. Using a series of examples in Section 4.1, Section 5.1 and Section 5.2 we show our definitions to formalise some quite natural development steps.

Our general theory, in the companion paper [13], centres around a parametrised definition of refinement, which was obtained by reflecting on several particular sorts of refinement, and also on a what seems to be a "natural" notion of refinement, and then abstracting. Various special theories come about by fixing the set of contexts and observations considered. Notable examples of special models we deal with are: abstract data types (ADT); handshake processes such as in Communicating Sequential Processes (CSP, [7]) or the Calculus for Communicating Systems (CCS, [9]), or broadcast processes such as in the Calculus of Broadcasting Systems (CBS, [10]); and individual operations.

In our general model we take as primitive the following three components: one, a set of **entities**, the specifications and implementations we wish to develop by refinement; two, a set of **contexts**, the environment with which the entities interact; and three, a **user** who uses an entity in a given context.

Concrete examples include:

(i) an entity as a motor, a context as the car in which the motor runs and the user as the driver of the car

(ii) an entity as an abstract data type, a context as the program using the abstract data type and the user a person (or other program) calling the context program.

(iii) an entity as a method, a context as the object containing the method and the user a program using the object.

In Section 3 we introduce the second part of our general theory *vertical refinement* between different special models. Viewing each such model as a *layer*, the lower, more detailed, layer can be seen as an implementation of the higher, more abstract layer.

As a concrete example of this we implement the IBP layer in the broadcast layer in Section 5.2. 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.

In Section 5.3 we show the usefulness of our general approach by giving a formal development of a simple, and very natural, system that combines both handshake and broadcast events.

Extension refinement [3] and behavioural sub-typing [5] have been defined so that they make visible (i.e. reverse hiding and restriction of various kinds) in the concrete entity events not visible in the abstract entity. It now turns out that this sort of refinement can be formalised as another special case of theory morphisms.

Finally, we show an example, that can now be treated as refinement, which was originally given as an example of what refinement can *not* do.

To summarise: existing refinements as given in the literature are, in our terms, special theories (of refinement) which come about by specialising (instantiating the parameters to) our general theory of refinement.

Each special theory can be viewed as a layer in a hierarchy of theories, each connected by vertical refinement, formalised as theory morphisms or Galois connections, which are properly seen as refinements since they preserve certain crucial guarantees.

Throughout this work it is the preservation of guarantees as we make development steps that justifies viewing these steps as refinements.

2 Developing the General Model of Refinement—adding layers

We now view the special models (and indeed any other specialisation of the general model) as a *layer* in the larger scheme of things.

A layer is formalised by a set of entities and a refinement relation. It is important to recall that : one, the entities in a layer can be ADTs, processes of various kinds and even individual operations; and two, different refinement relations can give different meanings to the same operational semantics.

Definition 2.1 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.

By considering only layers where the refinement relation is defined as in the companion paper [13], i.e. $\sqsubseteq_\mathsf{L} \triangleq \sqsubseteq_{\Xi_\mathsf{L}, O_\mathsf{L}}$, our layers can equally well be defined by the triple $(\mathsf{E}_\mathsf{L}, \Xi_\mathsf{L}, O_\mathsf{L})$.

For example, 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: one, 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 two, lift placing in a context from LTS to entities, i.e. $\mathsf{A}_\mathsf{L} =_\mathsf{L} \mathsf{B}_\mathsf{L} \Rightarrow \forall\, x \in \Xi_\mathsf{L}.[\mathsf{A}_\mathsf{L}]_x =_\mathsf{L} [\mathsf{B}_\mathsf{L}]_x$, as is the case for all the models we consider.

We make our general refinement more flexible in Section 3 by giving a general definition of *vertical refinement* between an abstract and a concrete

layer. Our definition of vertical refinement can be seen as a generalisation of non-atomic refinement [4] or action refinement [14,6] when we consider the LTS used to represent entities. But, when we consider predicates used to define the $\Xi \times \mathbb{O}$ relations then vertical refinement is a theory morphism similar to those used in UTP [8, Chapter 4] but based on different theories.

Our definition is based on two semantic mappings: $[\![\_]\!]_v$, that defines how to interpret the high-level abstract entities as low-level concrete entities; and vA, 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.

Our definition of vertical refinement is very much more powerful than the, as we shall call them, *horizontal* refinements within a layer and, as usual, with power comes responsibility. The implementer of a specification may be thought of as free to refine, horizontally, the specification in any way they choose. However, vertical refinement is powerful enough to formalise "design decisions" that are the responsibility of the customer (writer of the specification).

Fig. 1 shows how our "single layer" general theory generalises further once layers (and vertical refinement) are considered. This diagram is meant to give an idea of the generalisation we are about to make: the diagram is meant to be helpfully *suggestive*, not definitional, and its various components will be defined shortly. The unbroken line shows the steps of a refinement: one step in the top layer, one step of vertical refinement between layers, and one step in the bottom layer.

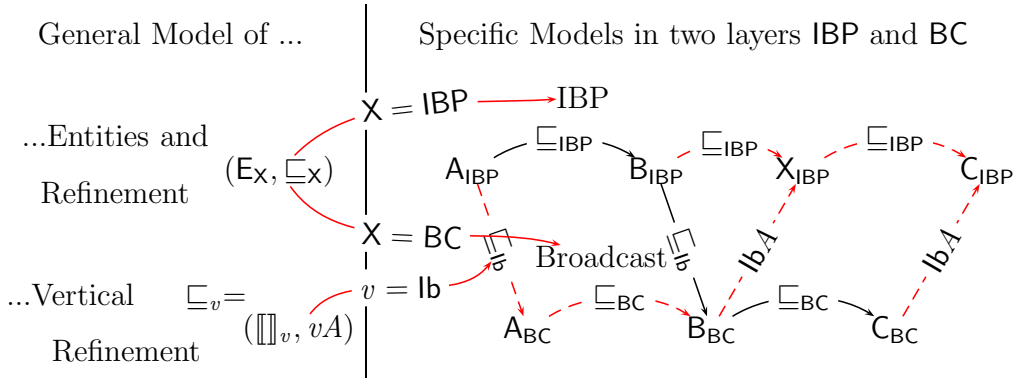

Fig. 1. Big picture

2.1 Motivating examples of Refinement

Before we give our abstract definition of vertical refinement we will discuss some special instances from both event-based and state-based theories. These examples are of interest in their own right but are used here to show how a single abstract definition of refinement can be of interest in quite distinct special models.

Our first example is taken from the state-based literature [1] and will be discussed in more detail in Section 4.2. Programmers frequently consider data structures (lists, trees, sets etc.) without fixing the maximum size they can grow to and restricting themselves to considering just the correct behaviour of the system. Nonetheless, at some point in the design, maybe near the end of the process, the maximum size must be fixed (even if only by accepting some system defaults), and error handling has to be tackled. Traditionally the fixing of the size would not be considered a formal refinement but some other informal design step. What we do here is relax the formal definition of refinement so that this step can be viewed as a formal refinement, with a guaranteed relation between the abstract specification and the implementation. The same goes for error handling.

Our second example, an event-based example, will be discussed in more detail in Section 5. But by way of introduction, note that an early phase in constructing an event-based formal model of any system is that of deciding what constitutes an event. This requires both the set of events, called the alphabet, to be fixed, along with their type of interaction. How the events interact can be modelled by defining the entities E_L and their contexts Ξ_L and the observations we can make O_L to define refinement $\sqsubseteq_{\Xi_\mathsf{L},O_\mathsf{L}}$ of the entities. Thus at this early step in the development a layer has been fixed and an entity chosen to represent the specification. This specification is then developed using the defined refinement.

Within a layer all entities are built from a common alphabet or set of events. These events are atomic viewed from within the layer, i.e. they have no internal structure. But the vertical refinement may give the high-level events internal structure by relating them to entities on the low-level layer. Such refinements have been extensively studied under the names *non-atomic refinement* [4] or *action refinement* [14,6].

There are two well-known issues that are immediately apparent. Firstly, the interleaving assumption must be avoided, and secondly: "The kind of steps one would like to make in top-down design do not always correspond completely to the kind of constructions allowed by action refinement."[6, section 7].

It is well-known [6] that interleaving can be avoided. Here we will side-step

the problem by restricting our attention to vertical refinements that relate one sequential entity to another sequential entity. We will focus our attention on the second issue, that of defining vertical refinement so that it is more relaxed than action refinement and reflects some steps that might appear in top-down design.

So, the system may have some features modelled by high-level events in alphabet Act_{H} and others modelled by low-level events in alphabet Act_{L}. We first model the features needing the high-level events, then we vertically refine this to an entity using only low-level events. This step preserves the meaning of the specification while embedding it in a more detailed low-level layer.

Recall our three basic types of interaction: one, method calling of programs; two, handshake synchronisation of process algebra; and three, broadcast communication. We use vertical refinement to embed one layer in another, thus defining how we might "implement" one style of interaction in another.

In Section 5.2 we construct a special(ised) vertical refinement from a specific abstract layer to a specific concrete layer. The events in the concrete layer are broadcast events and the events in the abstract layer are handshake events. But with the semantic mappings we have designed we have a Galois connection only when the handshake events are limited to the IBP entities. We have found no way build a Galois connection between handshake processes and broadcast processes. Indeed, we conjecture that this cannot be done.

3 General vertical refinement

We use a semantic mapping $[\![\_]\!]_v$ to embed, or *interpret*, high-level $\mathsf{E_H}$ entities as low-level entities $\mathsf{E_L}$. To allow for the possibility that not all the low-level entities and contexts are in the range of $[\![\_]\!]_v$ we use a separate semantic mapping vA to embed, or *interpret*, low-level entities as high-level entities.

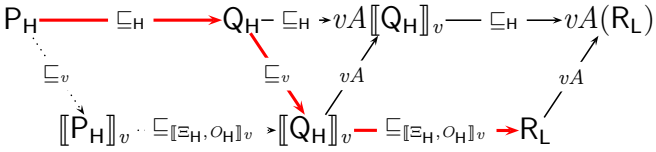

Fig. 2. Refinement → within and between layers

In top-down development a vertical refinement $\sqsubseteq_v = ([\![\_]\!]_v, vA)$ may be preceded by some high-level refinement steps and may itself precede low-level refinement steps (see Fig. 2, and here, to make the point we abuse notation and use $\sqsubseteq_v$ to emphasise the use of the refinement between layers when what actually does the mapping is $[\![\_]\!]_v$). The vertical refinement replaces a high-level entity by a low-level entity. But this new low-level entity cannot interact

with the old high-level contexts so the contexts must also be vertically refined. (It will turn out in all our examples here that the observation function always just gives us complete traces, though the range of that function may change—as it does with the operation refinement example in the companion paper [13]—as we do vertical refinements.) As we will see it is the application of the refinement mappings to the contexts that allows the contexts to be sufficiently different on each layer so that the refinement on each layer assigns a different interpretation of interaction in the different layers.

The "mixing interaction type" situations we have described use low-level entities, contexts and observations in the range of $[\![\_]\!]_v$ only. Hence $\sqsubseteq_{[\![\Xi_{\mathsf{H}}, O_{\mathsf{H}}]\!]_v}$ is an appropriate refinement.

For this example, if we have semantic mappings $[\![\_]\!]_v$ and vA so that:

(i) low-level refinement can be interpreted as high-level refinement
$$[\![\mathsf{P_H}]\!]_v \sqsubseteq_{[\![\Xi_{\mathsf{H}}, O_{\mathsf{H}}]\!]_v} \mathsf{R_L} \Rightarrow \mathsf{P_H} \sqsubseteq_{\mathsf{H}} vA(\mathsf{R_L})$$

(ii) high-level refinement can be interpreted as low-level refinement
$$[\![\mathsf{P_H}]\!]_v \sqsubseteq_{[\![\Xi_{\mathsf{H}}, O_{\mathsf{H}}]\!]_v} \mathsf{R_L} \Leftarrow \mathsf{P_H} \sqsubseteq_{\mathsf{H}} vA(\mathsf{R_L}).$$

then we have what we call a vertical refinement.

This is a special case of the following:

Definition 3.1 Semantic mappings $[\![\_]\!]_v^{\mathsf{HL}}$ and vA^{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:

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

We drop the superscripts where possible, using the context to determine H and L.

The entities in any layer are represented by equivalence sets of operational semantics (e.g. sets of LTS) not just a single operational semantics. When the semantic mappings define a vertical refinement then the equalities $=_{\mathsf{H}}$ and $=_{\mathsf{L}}$ are *congruent* w.r.t. the relevant semantic mappings $[\![\_]\!]_v$ and vA [3]. Consequently we are free to define the semantic mappings as mappings between individual operational semantics and then lift them to mapping between entities (equivalence classes of operational semantics e.g. LTS).

So, a *vertical refinement* $\sqsubseteq_v = ([\![\_]\!]_v, vA)$ defines a guaranteed relation between the more abstract high-level entities and the more concrete low-level entities.

[3] Monotonicity with respect to the preorders defining an adjunction is a well-known property of adjunctions [15, p151].

Thinking of layers as theories means the two functions $[\![\_]\!]_v^{\mathsf{HL}}$ and vA^{HL} define how to *interpret* one theory in the other, so we have a theory morphism, and consequently:

$\mathsf{H} \sqsubseteq_v^{\mathsf{HL}} \mathsf{L}$ **guarantees** that the high-level vA-interpretation of entity L behaves like (can be observed to have a subset of the observations of) entity H (e.g. $\mathsf{P_H}$ in Fig. 2) whenever it is placed in any high-level context Ξ_{H} and only the high-level observations $\mathbb{O}_{\mathsf{H}}$ are made.

In Section 5.2, we define a vertical refinement from an IBP layer to a broadcast layer. But, we have been unable to extend this vertical refinement to a high-level layer of handshake processes, see Section 5.3, as we can "implement" only the deterministic IBPs.

3.1 Subset morphisms

In this section we are interested in the special case of theories A and C where $\Xi_{\mathsf{A}} \subseteq \Xi_{\mathsf{C}}$ and $\mathbb{O}_{\mathsf{A}} \subseteq \mathbb{O}_{\mathsf{C}}$.

It is well-known ([15, p155] [8, 4.1]) that subset relations like $\Xi_{\mathsf{A}} \times \mathbb{O}_{\mathsf{A}} \subseteq \Xi_{\mathsf{C}} \times \mathbb{O}_{\mathsf{C}}$ form a simple theory morphism which we denote by $\sqsubseteq_{sub}^{\mathsf{AC}}$, where the interpretation mappings are:

embedding of the abstract in the more complex concrete, where for any $\mathsf{P_A} \in \mathsf{E_A}$ (using the definitions $\Xi_{\mathsf{C}\backslash\mathsf{A}} \triangleq \Xi_{\mathsf{C}} \backslash \Xi_{\mathsf{A}}$ and $\mathbb{O}_{\mathsf{C}\backslash\mathsf{A}} \triangleq \mathbb{O}_{\mathsf{C}} \backslash \mathbb{O}_{\mathsf{A}}$) :

$$[\![\mathsf{P_A}]\!]_{sub}^{\mathsf{AC}} \triangleq [\![\mathsf{P_A}]\!]_{\Xi_{\mathsf{A}}, O_{\mathsf{A}}} \cup \{(x, o) \mid x \in \Xi_{\mathsf{C}\backslash\mathsf{A}} \vee o \in \mathbb{O}_{\mathsf{C}\backslash\mathsf{A}}\};$$

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

$$subA^{\mathsf{AC}}(\mathsf{P_C}) \triangleq [\![\mathsf{P_C}]\!]_{\Xi_{\mathsf{A}}, O_{\mathsf{A}}}.$$

We can establish that $\sqsubseteq_{sub}^{\mathsf{AC}}$ is a theory morphism, i.e. that:

$$\forall \mathsf{X_A} \in \mathsf{E_A}, \mathsf{Y_C} \in \mathsf{E_C}. [\![\mathsf{X_A}]\!]_{sub}^{\mathsf{AC}} \sqsubseteq_{\mathsf{C}} \mathsf{Y_C} \Leftrightarrow \mathsf{X_A} \sqsubseteq_{\mathsf{A}} subA^{\mathsf{AC}}(\mathsf{Y_C})$$

by checking that:

$$\forall \mathsf{X_A} \in \mathsf{E_A}, \mathsf{Y_C} \in \mathsf{E_C}. [\![\mathsf{X_A}]\!]_{\Xi_{\mathsf{A}}, O_{\mathsf{A}}} \cup \{(x, o) \mid x \in \Xi_{\mathsf{C}\backslash\mathsf{A}} \vee o \in \mathbb{O}_{\mathsf{C}\backslash\mathsf{A}}\} \supseteq [\![\mathsf{Y_C}]\!]_{\Xi_{\mathsf{C}}, O_{\mathsf{C}}}$$

$$\Leftrightarrow [\![\mathsf{X_A}]\!]_{\Xi_{\mathsf{A}}, O_{\mathsf{A}}} \supseteq [\![\mathsf{Y_C}]\!]_{\Xi_{\mathsf{A}}, O_{\mathsf{A}}}$$

Intuitively we can think of $\Xi_{\mathsf{A}} \times \mathbb{O}_{\mathsf{A}}$ as defining a *frame* and subset morphisms as formalising **undefined outside frame**. That is to say the abstract A is silent outside of its frame.

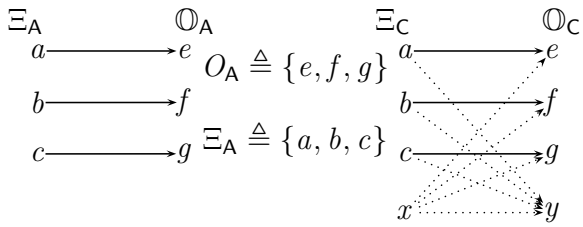

Fig. 3. $\mathsf{A} \sqsubseteq^{\mathsf{AC}}_{sub} \mathsf{C}$

Our subset theory morphism formalises the addition of new observations, the y in Fig. 3. The intuitive justification for adding $\{(a, y), (b, y), (c, y), (x, y)\}$ is that in the abstract specification $\Xi_A \times \mathbb{O}_A$ the y observation had not been considered (recorded). Although this definition of vertical refinement may seem unusual when considering the entity to be a single operation, it is no more than an application of Galois connections. It is the adding of the new observations that makes our formal model (of both single operations and machines) so flexible. In addition it is the preservation of the guarantee that allows us to view theory morphisms as refinements.

Recall that our theory can be applied both to single operations and to processes or machines. As not all context, observation relations represent valid processes we have the problem that the embedding $[\![\_]\!]^{\mathsf{AC}}_{sub}$ may return relations that are not the semantics of any valid processes. We could, like UTP, define healthiness condition on the relational semantics so that all processes have healthy relational semantics and all healthy relations are the semantics of some processes. But as we are interested primarily in refinement and the guaranteed relation between entities and what they are refined into, we choose to avoid any discussion about healthiness conditions.

The strict embedding projection morphisms satisfy a strict guarantee

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

But, this is not always very useful in practice as $[\![\mathsf{P_A}]\!]^{\mathsf{AC}}_{sub}$ may be unhealthy. So we take a more pragmatic view and consider $\mathsf{P_A} \sqsubseteq^{\mathsf{AC}}_{sub} [\![\mathsf{P_A}]\!]^{\mathsf{AC}}_{sub} \sqsubseteq_\mathsf{C} \mathsf{P_C}$. Hence we can choose some actual process $\mathsf{P_C}$ whose relational semantics is a subset of the potentially unhealthy $[\![\mathsf{P_A}]\!]^{\mathsf{AC}}_{sub}$ and we still have a useful refinement guarantee.

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

> $\mathsf{P_A} \sqsubseteq^{\mathsf{AC}}_{sub} [\![\mathsf{P_A}]\!]^{\mathsf{AC}}_{sub} \sqsubseteq_\mathsf{C} \mathsf{P_C}$ **guarantees** the high-level vA-interpretation of any entity $\mathsf{P_C}$ behaves like (can be observed to have a subset of the observations of) entity $\mathsf{P_A}$ whenever it is placed in any abstract context Ξ_A and only the

abstract observations $\mathbb{O}_{\mathsf{A}}$ are made.

Concrete examples of this appear in Section 5.1.

Thinking of Fig. 3 as representing a single operation then we observe that this is not how refinement is normally defined in the literature. In particular note the increase in non-determinism. Similarly if we consider x and y to be $\perp$, Fig. 3 is a definition of lifting and totalising the operation but not one that appears in the literature.

4 Vertical refinement between state-based systems

4.1 Operations

The ISO Z semantics is silent about termination and has been formalised in the companion paper [13] by $[\![\mathsf{E}]\!]_{(\Xi_{\mathsf{isoz}}, O_{\mathsf{isoz}})}$. Other interpretations, such as partially correct or undefined outside of precondition, can easily be formalised by defining appropriate observation functions. For these interpretations the frame (of reference) will be a superset of the ISO Z frame $\Xi_{\mathsf{isoz}} \times \mathbb{O}_{\mathsf{isoz}}$.

By the application of the obvious subset morphism the ISO Z semantics can be formally refined into other interpretations of behaviour. Thus the choice of how to interpret the ISO Z semantics need not be made as the first step in the development process but can be postponed until the choice, if ever, is needed. Importantly when the design decision is made it can be applied without leaving our formalism and a guarantee can be given as to the relation between the initial ISO Z specification and its new interpretation.

4.2 Abstract data types

We take this example, formalised in Z, from [1], but wish to stress our paper is not about Z. Also, the work on approximate refinement [2] provides another means to deal with these sorts of examples.

All the reader needs to know about Z is that state spaces and operations over them are defined by schemas: named boxes with declarations above the dividing line and predicates giving properties below the line. Operations are then to be understood as relations between "before and after" states, or pre- and post-states, using the useful convention that pre-state observation names are unprimed, e.g. s, and post-state observation names are primed, e.g. s'. This priming convention is also applied to state schemas, so $State'_{\mathsf{A}}$ has an observation named s'. [4]

[4] Seasoned Z readers will note conventions that we might have followed to make our Z more standard—we have omitted these since, as we said, this paper is not about Z, does not depend on it and no knowledge of Z is needed to read it.

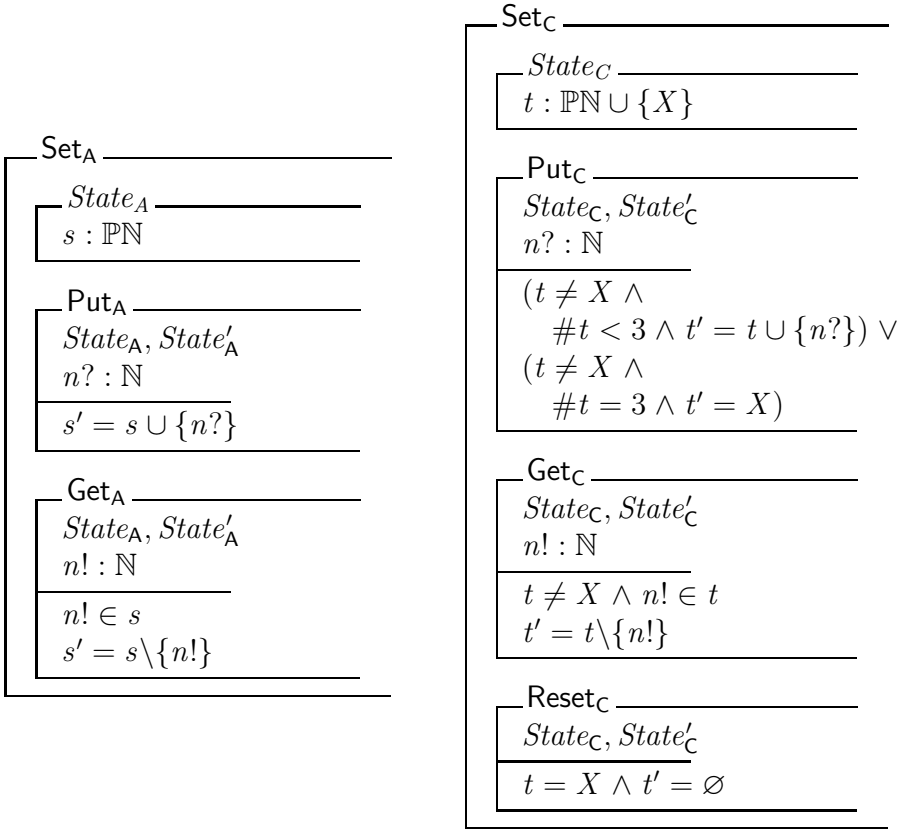

Fig. 4. Infinite $\mathsf{Set_A}$ and bounded $\mathsf{Set_C}$

The abstract definition $\mathsf{Set_A}$ of a (data structure for a) set containing natural numbers with two operations $\mathsf{Put_A}$, to add numbers to the set, and $\mathsf{Get_A}$, to remove them, can be found in Fig. 4. [5]

The black-box view of the abstract data type $\mathsf{Set_A}$ is that the state is private or unobservable. Only the public *method calls* are observable and consequently $\mathsf{Set_A}$ is interpreted as a specification guaranteeing that an infinite number of $\mathsf{Put_A}$ operations, each with distinct inputs, can be successfully called. This is plainly not possible to implement. Computers have a finite amount of storage and hence a program that is repeatedly executing $\mathsf{Put_A}$ will, at some point, simply run out of space. A more concrete, and now implementable, definition, $\mathsf{Set_C}$, with the size of the set bounded by three,

[5] We note that, in fact, since Z is strongly typed and so $\cup$ must join two things of the same type, the type $\mathbb{P}\mathbb{N} \cup \{X\}$ in this example as it stands cannot properly be said to be Z. However, with more work, we could make this proper Z, but it would complicate, somewhat, what is written for the type concerned and would therefore distract us from the point of this paper. Finally, we have used exactly the example from [1].

can be found in Fig. 4 too (for the moment ignore mention of the set $\{X\}$). If refinement is meant to capture this black-box guarantee then $\mathsf{Set_C}$ is not a refinement of $\mathsf{Set_A}$.

Clearly the example is very small and we could easily throw away $\mathsf{Set_A}$ and use the concrete specification $\mathsf{Set_C}$ in the first place. The point we make, though, is that if one were given a *large* complex specification on which a lot of time has been spent, then one would be reluctant to throw away all this effort and start again.

Nonetheless, if we accept that in some "practical situations" a reasonable person might wish to view $\mathsf{Set_C}$ as a refinement of $\mathsf{Set_A}$ then such a person cannot be giving $\mathsf{Set_A}$ this black-box interpretation.

Here we replace the black-box interpretation with a clear-box interpretation by regarding the state of $\mathsf{Set_A}$ to also be part of the specification. Thus, via the undefined outside of frame interpretation that subset morphisms allow, we can modify both the state of $\mathsf{Set_A}$ and the set of operations in $\mathsf{Set_A}$.

Given that we would like $\mathsf{Set_C}$ to be a refinement of $\mathsf{Set_A}$ we can, informally speaking, ask for a guarantee that $\mathsf{Set_C}$ behaves just like $\mathsf{Set_A}$ in contexts satisfying the following assumptions:

(i) the set is not used to store more than three different numbers; and

(ii) only the Put and Get operations are called.

This clear-box guarantee is certainly weaker than the (unreasonable because unimplementable) black-box guarantee we started with, but it seems to be the strongest guarantee we can expect, and, crucially, it is useful and, probably, all we were expecting all along (being reasonable people).

We will show how to formally model the development of $\mathsf{Set_A}$ into $\mathsf{Set_C}$ via three refinement steps and show that its formal guarantee corresponds to the above informal guarantee.

First we will perform the same subset morphism on both operations in $\mathsf{Set_A}$ and thus introduce some non-determinism. Secondly we will perform a normal horizontal refinement to remove the non-determinism we just introduced and finally we will perform a subset morphism on the whole data type to introduce a new operation.

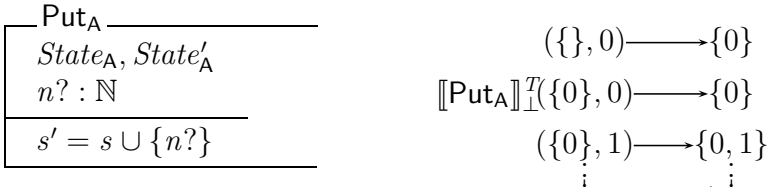

Fig. 5. $\mathsf{Put_A}$ and $[\![\mathsf{Put_A}]\!]^{T}_{\perp}$

The only difference between the relational semantics in Fig. 5 and the semantics that appeared in Section 3.1 is that the pre-state of Put is a pair, the first element being *State* and the second being the input value n

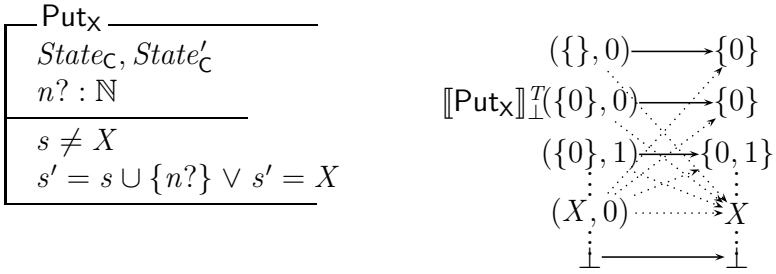

Fig. 6. $\mathsf{Put_A} \sqsubseteq^{\{X\}}_{sub} \mathsf{Put_X}$

That the operation $\mathsf{Put_A}$ can be refined via a subset morphism into $\mathsf{Put_X}$ using the results and definitions of Section 3.1 is clear and we leave the details to the interested reader.

We define a state-based refinement of an ADT to be the state-based refinement of all its operations and hence for example Fig. 7:

$$\mathsf{Set_A} \sqsubseteq^{\{X\}}_{sub} \mathsf{Set_X} \triangleq \mathsf{Put_A} \sqsubseteq^{\{X\}}_{sub} \mathsf{Put_X} \wedge \mathsf{Get_A} \sqsubseteq^{\{X\}}_{sub} \mathsf{Get_X}$$

Having introduced nondeterminism into the specification we are now free to remove it via horizontal refinement $Set_{\mathsf{X}} \sqsubseteq Set_{\mathsf{B}}$. The final refinement is applied to the whole of $\mathsf{Set_B}$ to introduce the new operation Reset. The contexts Ξ_{B} are any trace of the operations in $\mathsf{Set_B}$ so clearly any trace that includes Reset is not in this set of contexts and hence the behaviour of $\mathsf{Set_B}$ in this context is not defined. Hence it is easy to establish that there exists a subset morphism and refinement $\mathsf{Set_B} \sqsubseteq^{\{\mathsf{Reset}\}}_{sub} \mathsf{Set_C}$.

We adopt the clear-box interpretation of the specification $\mathsf{Set_A}$ hence it defines only the behaviour of its operations $\mathsf{Put_A}$ and $\mathsf{Get_A}$ and it only defines their behaviour when they remain within $State_{\mathsf{A}}$.

The first refinement $\mathsf{Set_A} \sqsubseteq^{\{X\}}_{sub} \mathsf{Set_X}$ guarantees that $\mathsf{Set_X}$ behaves like $\mathsf{Set_A}$ when operations $\mathsf{Put_X}$ and $\mathsf{Get_X}$ keep out of the error state X.

The second refinement step is a simple reduction of nondeterminism. It is in this step that the developer decides that the set is to have no more than three elements. Hence the guarantee is unchanged when sets never have more than three elements but any operation that attempts to increase the size to greater than three is free to return the new state X.

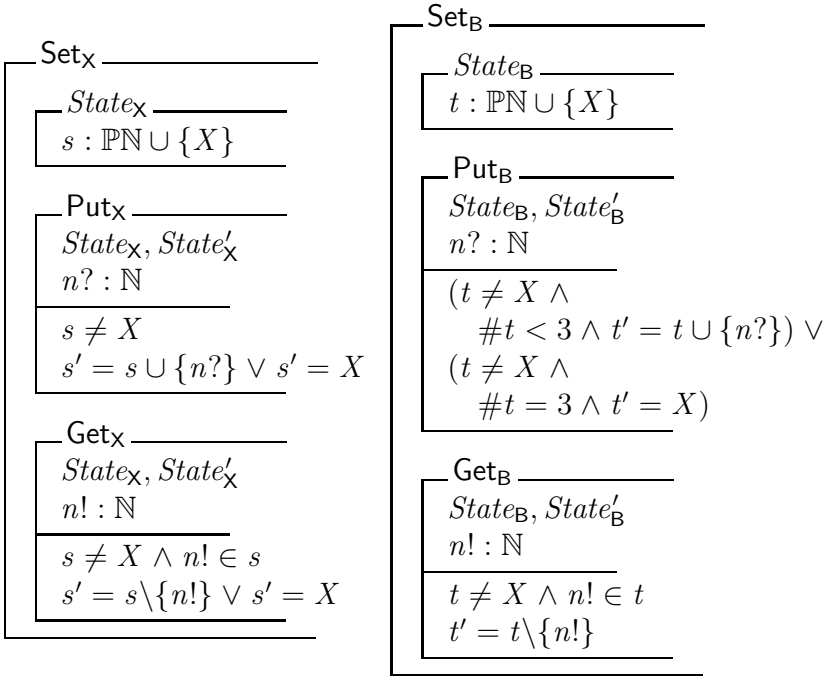

Fig. 7. $\mathsf{Set_A} \sqsubseteq_{sub}^{\{X\}} \mathsf{Set_X}$ and $\mathsf{Set_X} \sqsubseteq \mathsf{Set_B}$

The third and final refinement $\mathsf{Set_B} \sqsubseteq_{sub}^{\{\mathsf{Reset}\}} \mathsf{Set_C}$ guarantees that any behaviour of $\mathsf{Set_C}$, when used by programs that call only the "Put" and "Get" operations could also be a behaviour of $\mathsf{Set_B}$.

Together these guarantees form exactly the guarantee we wanted, as given in Section 4.2, and they have been captured formally via flexible refinement.

5 Vertical refinement between event-based LTS

By applying vertical refinement to processes with LTS operational semantics we are able to refine processes based on actions with one style of interaction (e.g.handshake) on a layer of process with another style of interaction (e.g.broadcast). Before considering vertical refinement in general we consider the much simpler subset morphisms and show how to use them to model restriction and hiding as found in the process literature CSP/CCS/ACP.

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. Having

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

The abstraction functions, Restriction and Hiding, can simply be applied to a process to remove events from a concrete process when ever the developer chooses. But here, as in [3,5], we are interested in reversing this process and introducing these events, new to the abstract process, and thus creating the concrete process. Further we are interested in viewing the introduction of these new events as a formal refinement step.

Let the alphabet of an entity A, written $\alpha(\mathsf{A})$, be the set of events that it can engage in. Further let the alphabet of the set of contexts Ξ_{A} be the union of the alphabet of the individual contexts $\alpha(\Xi_{\mathsf{A}}) \triangleq \{\alpha(\mathsf{x}) \mid \mathsf{x} \in \Xi_{\mathsf{A}}\}$.

We reverse the τ-abstraction and δ-abstraction of Definition 6.1 by extending refinement to introduce events in two quite separate ways [11,12].

Firstly if δ–refinement holds, $\mathsf{A} \sqsubseteq_{\Xi\delta Del} \mathsf{C}$ (as we apply the abstraction functions both to entities and contexts $\alpha(\Xi_{\mathsf{A}}) \cap Del = \varnothing$), then it introduces events 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, $\mathsf{A} \sqsubseteq_{\Xi\tau Hid} \mathsf{C}$ (where $\alpha(\mathsf{A}) \cap Hid = \varnothing$), then it introduces events that were previously not observable and never blocked in the more abstract view.

Definition 5.1 δ–refinement and τ–refinement. For LTS A and C :

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

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

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

5.2 $(\mathsf{T}_{\mathsf{IBP}}, \sqsubseteq_{\mathsf{IBP}}) \sqsubseteq_v (\mathsf{T}_{\mathsf{BC}}, \sqsubseteq_{\mathsf{BC}})$

In this section we will define a particular vertical refinement between high-level IBP entities and low-level broadcast processes, both from the companion paper [13]. We will then show that we have been unable to extend the high-level to all handshake processes. The reason appears to be related to the way handshake processes have abstracted away the *cause* and *response* nature of event synchronisation.

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.

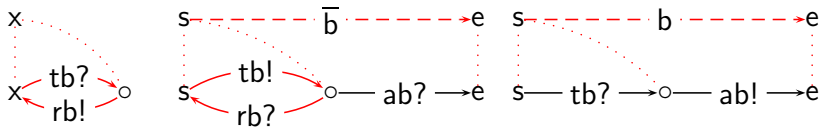

Fig. 8. Vertical refinement $[\![\_]\!]_B$

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 also add try-reject loops 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 [14].

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 tb!,rb! and ab! are not.

Definition 5.2 Let A be an LTS $(N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$.

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

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

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

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 abstraction vA_B. It should be noted that tx? events are replaced by two τ events, one each way.

Definition 5.3 Let A be an LTS $(N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$.

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

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

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

5.3 Vertical refinement failure—and success

We take the special vertical refinement above, defining how to refine IBP into broadcast processes, as being almost inevitably correct. 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 our companion paper ([13], Section 5.4) and reproduced here in Fig. 9. Recall that we described Rob as nondeterministic and here our "implementation" on a broadcast layer, as we show, also requires Rob to be nondeterministic.

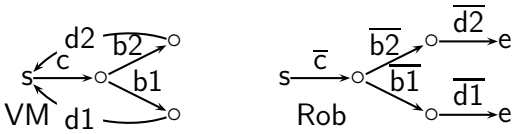

Fig. 9. Are VM and Rob deterministic?

The implementation on a broadcast layer is illustrated in Fig. 10[6].

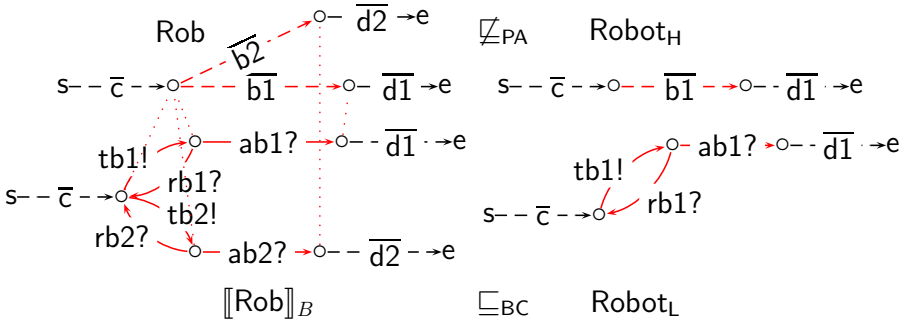

Fig. 10. $[\![\mathsf{Rob}]\!]_B \sqsubseteq_{\mathsf{BC}} \mathsf{Robot_L}$ but $\mathsf{Rob} \not\sqsubseteq_{\mathsf{PA}} \mathsf{Robot_H}$ and $M_{IBP}(\mathsf{Rob}) \sqsubseteq_{\mathsf{IBP}} \mathsf{Robot_H}$

$[\![\mathsf{Rob}]\!]_B$ in Fig. 10 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 $\mathsf{Robot_L}$ is a refinement of $[\![\mathsf{Rob}]\!]_B$ that will try button b1 only.

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

[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.

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.

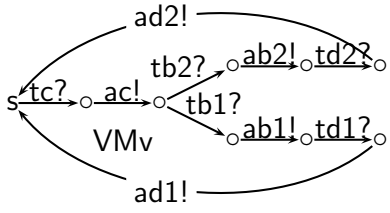

Fig. 11. (Fig. 9) VM $\sqsubseteq_v$ VMv

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. 11.

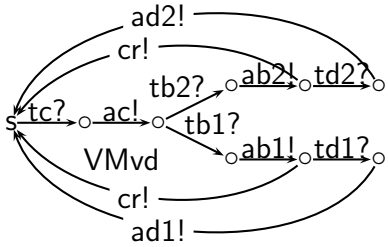

Fig. 12. (Fig. 11)VMv $\sqsubseteq_{BC\delta\{\mathsf{cr}\}}$ VMvd

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 here we simply view VMb as "sugar" for VMvd in Fig. 12 and leave the reader to expand the dashed lines in Fig. 8.

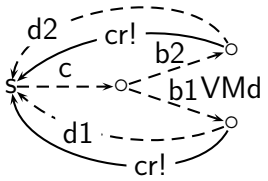

Fig. 13. VM $\sqsubseteq_{BC\delta\{\mathsf{cr}\}}$ VMb

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.

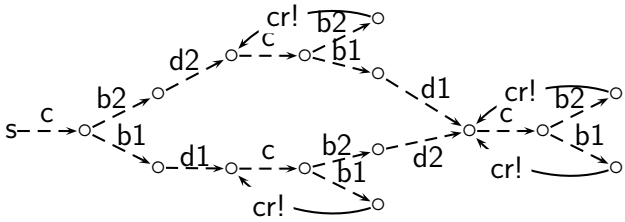

Fig. 14. VMvd $\sqsubseteq_{BC}$ VM2

6 Conclusion

By making use of an explicit representation of the contexts in which entities are placed we have been able to construct a flexible definition of refinement.

What can be observed is also a parameter in our flexible refinement and has been used in the following ways:

(i) Refinement steps that expand the set of considered observations, for historical reasons called a subset morphism, have been used for both state- and event-based systems—adding error states and new operations in Section 4.2 and adding new events in Section 5.1;

(ii) Using a function to relate entities with completely distinct sets of events in Section 5.3.

We define a layer as consisting of a set of entities and a refinement relation based on the style of interaction between entities in the layer. Using this we define vertical refinement (Section 3) between different layers where each layer may contain different styles of event-interaction. As an example we define vertical refinement from a "handshake layer" to a "broadcast layer".

Both for state-based models Section 4.2 and event-based models Section 5.3 we have shown examples of our very "relaxed" definition of refinement allowing the formal introduction of error handling. Both examples adopt a very similar methodology. They each come with a formally guaranteed relation between the specification and implementation (more concrete specification). For neither example have we found alternative formal solutions in the literature.

Acknowledgement

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] Banach, R. and J. Derrick, *Filtering retrenchments into refinements*, in: *Proceedings of SEFM'06* (2006), pp. 60–69.

[2] Boiten, E. and J. Derrick, *Formal program development with approximations*, in: H. Treharne, S. King, M. Henson and S. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B*, Lecture Notes in Computer Science **3455** (2005), pp. 374–392.

[3] Brinksma, E. 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).

[4] Derrick, J. and E. Boiten, *Non-atomic refinement in Z*, in: J. Wing, J. Woodcock and J. Davies, editors, *FM'99 World Congress on Formal Methods in the Development of Computing Systems*, Lecture Notes in Computer Science **1708** (1999), pp. 1477–1496.

[5] Fischer, C. and H. Wehrheim, *Behavioural subtyping relations for object-oriented formalisms*, Lecture Notes in Computer Science **1816** (2000), pp. 469–483.
URL citeseer.nj.nec.com/fischer00behavioural.html

[6] Gorrieri, R. and A. Rensink, *Action refinement*, in: J. A. Bergstra, A. Ponse and S. A. Smolka, editors, *Handbook of Process Algebra*, Elsevier, 2001 pp. 1047–1147.
URL citeseer.nj.nec.com/gorrieri00action.html

[7] Hoare, C., "Communicating Sequential Processes," Prentice Hall International Series in Computer Science, 1985.

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

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

[10] Prasad, K. V. S., *A calculus of broadcasting systems*, Science of Computer Programing **25** (1995), pp. 285–327.

[11] Reeves, S. and D. Streader, *Comparison of data and process refinement*, in: J. Woodcock and J. Dong, editors, *Proceedings of ICFEM 2003*, number 2885 in Lecture Notes in Computer Science (2003), pp. 266–285.

[12] Reeves, S. 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.
URL http://researchcommons.waikato.ac.nz/cms_papers/14/

[13] Reeves, S. and D. Streader, *General refinement, part one: interfaces, determinism and special refinement*, Proceedings of Refine 2008, Electronic Notes in Theoretical Computer Science (2008).

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

[15] Taylor, P., "Practical Foundations of Mathematics," Cambridge University Press, 1999, cambridge studies in advanced mathematics 59.

Appendix

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 6.1 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:

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