Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 214 (2008) 277–307
www.elsevier.com/locate/entcs

General Refinement, Part One: Interfaces, Determinism and Special Refinement

Steve Reeves[1] and David Streader[2]

Department of Computer Science
University of Waikato
Hamilton, New Zealand

Abstract

We introduce a general model of refinement. This is defined in terms of what contexts an entity can appear in, and what observations can be made of it in those contexts.
We show explicitly how five refinement relations, taken from the refinement literature, are instances of our general model. Henceforth, since they are specialisations of a general model, we call these instances *special* models. We show these theories of refinement are special models simply by fixing the sets of contexts and observations involved in appropriate ways.

Keywords: refinement, operational semantics, state-based, event-based, determinism

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 entity into a more concrete entity.

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

doi:10.1016/j.entcs.2008.06.013

What, essentially, does this idea of satisfaction consist in? What is it that gets "passed on" or preserved down the chain of refinement steps that connects the original specification with its more concrete relatives? The answer: a guarantee that the concrete is as acceptable as the specification. Keeping our eye on this guarantee guides much of the development in this paper.

One of the central aims of formal methods is to define refinement so that an implementation built by formally verified refinement steps must satisfy the original specification. Once the entity is sufficiently detailed it can be implemented with no further design decisions.

For construction by formal refinement to be of wide use in practice there is an obvious need for refinement to be as flexible as possible. In order to express concepts most conveniently we give a characterisation of refinement at a very general level and then specialise this general theory to several (we give five examples) particular (special) theories, which are taken from, and are well-known in, the refinement literature.

Our general model is expressed in terms of an operational semantics.

In Section 2 (and in a companion paper [31]) we define our general model without fixing a specific operational semantics, thus allowing our general model to give rise to a wide variety of special models. A well-known specific operational semantics that our general method can incorporate is (state-based) relational semantics and another is (event-based) labelled transition semantics. Any one of these particular incorporations can be specialised further, so, as examples, CSP [18] interprets the events in its labelled transition system semantics as *handshake events*, whereas CBS [26] interprets the events in the same labelled transition system as *broadcast events*.

Our general model in Section 2 can be specialised to different particular special models and using this as a bridge we compare the definition of determinism found in different special models. We do this because the reduction of non-determinism underpins many definitions of refinement found in a variety of special models.

To our surprise we find the definition of determinism commonly used in the process algebra literature to be at odds with determinism as defined in other special models. In order to rectify this situation we return to the intuitions expressed by Milner within CCS and by formalising these intuitions we are able to define determinism in process algebra in such a way that it is no longer at odds with the definitions we have taken from other special models. Using our abstract definition of determinism we are able to construct a new model, interactive branching programs (IBP), that is an implementable subset of process algebra.

1.2 Further comments

Our general theory, in Section 2, centres around a parametrised definition of refinement, which was obtained by reflecting on several particular sorts of refinement and also on what seems to be a "natural" notion of refinement. Each of the particular models can in turn be seen as specialisations of the general theory, and so, doubtless, can many others. These 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, [18]) or the Calculus for Communicating Systems (CCS, [24]), or broadcast processes such as in the Calculus of Broadcasting Systems (CBS, [26]); and individual operations.

Our general model will 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** formalised by defining the set of observations that can be made when an entity is executed 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 object (abstract data type), a context as the program using the object 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.

Because of the very wide variety of examples we wish to consider our general theory must be quite abstract. In particular we do not wish to fix, at the general level, the operational semantics of its components. What we will concentrate on in the general model is the nature of the interfaces between the components and the rest of the system.

However, a reader familiar with one of the many event-based process formalisms is free to think about entities and contexts as labelled transition systems (LTS) and a reader familiar with one of the many state-based formalisms is free to think about entities as sets of named operations where each operation has a partial relational semantics (named partial relations, NPR).

It is important to recall that LTS have been used to model entities with different styles of interaction, e.g. abstract data types with singleton failure semantics [7], handshake processes with failure and trace semantics [18], broad-

cast semantics [26], etc. A similar situation exists in the state-based world where the operational semantics of objects and abstract data types are based on named partial relations, and these have also been given many different interpretations. For example, the operational semantics of an individual operation is frequently a partial relation. In Z as in [40,34] and B [2], in addition, operations are interpreted as undefined outside of precondition and totally correct, while in [7] they are interpreted as guarded outside of precondition and totally correct, but in [10, Chapter 1-7] they are interpreted as partially correct. ISO Z [1] uses partial relation semantics with no further interpretations.

What this shows us is that operational semantics are flexible because they are open to many different interpretations, and we view them as giving just *part* of the semantic story. As is common in both the state- and event-based worlds, and has been done in all the examples [7,18,26,40,34,2,10], different "meanings" have been given to the operational semantics, as above. In fact, all these different interpretations can be characterised by using different definitions of refinement to "complete" the semantics.

The main novelty in our general model is the explicit modelling of contexts. We frequently use the notation $[\_]_{\mathsf{X}}$ for a context (depending on some parameter X) because contexts can be pictured as, and defined by, terms with "holes" in, shown by $\_$. 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. Our general definition of refinement is thus parametrised by a set Ξ of possible contexts and a function O which determines what can be observed. This definition of refinement is, as close as we have been able to make it, a direct formalisation of an informal definition of refinement that appears widely in the literature, as we shall see. [3]

In the event-based world the number of definitions of refinement is huge, and frequently testing semantics are used to decide which refinement is appropriate in any particular situation (see [38,39] for surveys of testing semantics). The point here is that if a particular testing semantics closely formalises the interaction you are interested in then the refinement characterised by this testing semantics should be applied. Exactly the same can be said of our parametrised definition of refinement, i.e. select the set of contexts that an entity will be placed in and the sort of observations that can be made of it,

[3] The semantics (meaning) of an entity is given by an equivalence class over the operational semantics (e.g.an equivalence class of LTS or NPR). Where the equivalence relation ($=_{\Xi}$) is clear from the context we will refer to the operational semantics as defining the semantics of an entity.

specialise the general definition by instantiating the context parameter accordingly and use the resulting special refinement. We have shown in [27] that this approach gives results that would be expected from the literature.

Let us re-cap: once a decision has been made on the set of contexts that entities can be placed in and what can be observed of them, we can construct an appropriate, specialised, definition of refinement based on this decision. This definition of refinement is then used to complete the operational semantics which will model the entities.

A further benefit of having our general model is that by lifting or rephrasing definitions and methods from one special model into our general model we are often subsequently able to specialise these now generalised definitions and methods into other special models, so we get, so to speak, concept portability.

As refinement is used to complete the meaning of the operational semantics the question "Which refinement, the one in [16] or the one in [3], is correct?" is not sensible. What it is helpful to ask is "In what situations is it safe to use the refinement in [16] and in what situations is it safe to use the refinement in [3]?"

In addition to porting (transferring) ideas between special models, we are able to compare how the same intuitive idea is formalised in distinct special models. In particular we will compare how *non-determinism* is defined in different special models. As we will see this apparently simple idea has proven difficult to define to everyone's satisfaction.

1.3 Agenda

In Section 2 we introduce the first part of our general theory, a general definition of refinement that can be specialised to known definitions of refinement found in different parts of the refinement literature.

In Section 4 we look at ADTs and illustrate the usefulness of our general approach by showing, contrary to what appears in the literature, that there are two distinct notions of ADT refinement.

In Section 5 we consider first broadcast processes and then handshake processes (CSP/CCS). Because our definition of deterministic processes is at odds with a definition commonly used in the literature on handshake processes we must consider determinism in some detail.

The model of handshake processes has abstracted away the *cause* and *response* nature of event synchronisation. Consequently, in Section 5.5, we describe interactive branching programs (IBP), a model that combines aspects of handshake processes with the cause and response nature of event synchronisation found in programs. IBP can be thought of as a variation of handshake

processes for which all non-determinism can be removed from sequential entities by refinement. In this model our characterisation of determinism is, as we show, a formalisation of one of Milner's intuitions.

In Section 6 we investigate a special model which allows us to view entities as single operations.

All of these investigations and ideas are made possible because each of the special models is an instance of the single general model.

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.

In fact, each special theory can be viewed as a layer in a hierarchy of theories, each connected by further sorts of refinement—much more on this in the companion paper [31].

2 General model of refinement

In this section we give a general definition of a standard natural notion of refinement. We use three distinct systems: E, the entity being refined; X, the context which interacts privately with E; and U, a user that observes X.

All interaction occurs at the interface between two systems and an interaction will be modelled as an *action*. In the event-based models our actions are events but in state-based models actions can be either *being in a particular state* or *calling a method*.

Our user U takes on the role of a tester, so it passively observes any action in the interface between X and U.

Readers familiar with definitions of refinement may be wondering why we need both contexts and users seeing as the roles these two systems play is normally played by a single system. The reason we need both contexts and users is that in some situations two interfaces, each with distinct properties, are needed. Details will follow in Section 2.3.1.

We will use the following natural notion of refinement that appears in many places in the literature [6,10,40,12,11,2] and can be applied to operations, processes, machines etc.:

> The concrete entity C is a refinement of an abstract entity A when no user of A could observe if they were given C in place of A.

In order to formalise this notion we must decide what the user can observe, so we make some assumptions. In practice we are interested in reasoning about and refining small modules of a larger entity. Thus we model the entity (module) E as existing in some context X (rest of larger whole) interacting on

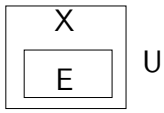

Fig. 1. Entity, conteXt and User and their interfaces

the set of actions *Act* where $Act \subseteq Names$ (where *Names* is a set consisting of all possible action names). All E's actions interact with X at the E-X interface (see Fig. 1). So, the actions in the set $Names \setminus Act$ are those which cannot appear in E and which, therefore, X and U communicate with, without interfering with communication between E and X. We model the observer as a *passive* user U that is a third entity that observes or interacts with X, but cannot block the X actions.

Note that the informal notion of refinement as presented earlier in this section talks about not only the entities involved in the refinement, but also the observations a user can make of them. Also, since the user, in order to make observations, must presumably use the entities they must have been placed in some contexts (e.g. programs which call the operations the entities provide). We should be careful when formalising refinement not to lose track of, or throw away, these contexts and observations.

We will give a formal general definition of refinement with explicit parameters representing both Ξ, the contexts in which A and C will be placed, and O, a function from entities to sets of traces $\wp(\mathbb{O})$ (e.g. of event names or states). Where each trace $tr \in \mathbb{O}$ is a potential observation. Our definition will have the following useful features:

One we can construct a guarantee that C *satisfies* A that is parameterised on both Ξ and O;

Two we can construct a simple logical theory, based on $\Xi \times \mathbb{O}$ relations, where $\mathbb{O}$ is the set of all traces. In this theory refinement is modelled by implication;

Three the well-known Galois connections can be used to define a new interpretation of entities, contexts and observations in terms of existing ones, consequently giving a new interpretation to both refinement and what refinement guarantees.

This general model can be made more concrete by instantiating its parameters Ξ and O to give what we call a *special theory*. It has been shown ([30]) that some of the classic theories of both abstract data types (ADT) and processes that appear in the literature are special theories of the general model given here.

Definition 2.1 General refinement. Let Ξ be a set of contexts each of which the entities A and C can communicate privately with, and O be a function which returns a set of traces, each trace being what a user observes of an execution. Then:

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

This general definition of refinement is one of the central parts of this paper and later it will be specialised (made more concrete) by:

one defining how we represent our entities;

two defining the sets of contexts Ξ; and

three defining the observation function O from entities to sets of traces.

We also define equality between representations:

Definition 2.2 Entity equality. Let Ξ be a set of contexts each of which the entities A and C can communicate privately with, and O be a function which returns a set of traces, each trace being what a user observes of an execution.

$$\mathsf{A} =_{\Xi,O} \mathsf{B} \triangleq \mathsf{A} \sqsubseteq_{\Xi,O} \mathsf{B} \wedge \mathsf{B} \sqsubseteq_{\Xi,O} \mathsf{A}$$

It should be remembered that this definition of refinement can, as we shall later see, be used to consider refinement for: an individual operation; a CSP/CCS-style process with "handshake" interaction; a CBS-style process with "broadcast" interaction; and even to ADTs with "method calling" interaction, all by selecting, for each case, a specific set of contexts and a particular observation function.

In the rest of this work we will usually consider only total correctness (live) semantics and hence O will need to return a complete trace. Thus, to reduce the notational clutter. we will frequently write $\sqsubseteq_\Xi$ in place of $\sqsubseteq_{\Xi,Tr^c}$ and $=_\Xi$ in place of $=_{\Xi,Tr^c}$.

Also, in all but the final section on operation refinement, in the rest of this paper we define "putting in a context" by:

Definition 2.3

$$[\_]_x \triangleq \_ \,||\, x$$

i.e. putting in a context means, in LTS terms, composing the entity in parallel with the context and allowing communication via any action.

2.1 Relational semantics

It is easy to see that we can give entities in our general model a relational semantics. We are not the first to use relations as a semantics for a diverse range

of models: indeed Hoare and He in their Unifying Theories of Programming (UTP, [19]) do just this.

Definition 2.4 Let Ξ be a set of contexts each of which the entity A can communicate privately with, and O be a function which returns a subset of the set $\mathbb{O}$ all of traces, each trace being what a user observes of an execution.The relational semantics of an entity A is a subset of $\Xi \times \mathbb{O}$:

$$[\![\mathsf{A}]\!]_{\Xi,O} \triangleq \{(x, o) \mid x \in \Xi, o \in O([\mathsf{A}]_x)\}$$

It should be noted that we use quite different relations to those in UTP, but like UTP we have refinement as subset of the relations or implication between the predicates that define them. Thus, like UTP, we can view each specialisation as defining a logical theory where refinement is implication.

For any A and C we have:

$$\mathsf{A} \sqsubseteq_{\Xi,O} \mathsf{C} \Leftrightarrow [\![\mathsf{C}]\!]_{\Xi,O} \subseteq [\![\mathsf{A}]\!]_{\Xi,O}$$

2.2 Determinism

Given that refinement is frequently characterised as the reduction of non-determinism we feel we need to define and then discuss (in Section 3) what it means to be deterministic in our general model.

Definition 2.5 An entity is deterministic if its relational semantics is a function.

$$Det_{\Xi,O}(\mathsf{A}) \triangleq (x, o) \in [\![\mathsf{A}]\!]_{\Xi,O} \wedge (x, p) \in [\![\mathsf{A}]\!]_{\Xi,O} \Rightarrow o = p$$

Although this definition is a perfectly valid definition it is of little use when contexts and entities are the same kind (e.g. processes) because we would need to restrict the contexts to being deterministic, but this is what the definition is supposed to be defining. So, when contexts and entities are quite distinct kinds of things then the above provides us with a definition of determinism that is parametrised on Ξ and O, but when contexts are built from deterministic entities then the above provides us with only a healthiness condition. Details will be discussed in Section 3 when specific examples are given.

2.3 Interfaces

In this section we will show why both contexts and users are needed to define refinement by demonstrating situations where two different types of interfaces are needed. one between entities and contexts but a second different type of interface between context and user.

2.3.1 Interface types

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.

We will refer to an interface as *transactional* if interaction (observation) occurs at no more than two distinct points: initialisation and finalisation of the entity. 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 at many points throughout the execution. Hence with interactive interfaces observations can be made prior to termination and even prior to non-termination.

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!)

From Fig. 1 we can see that 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.

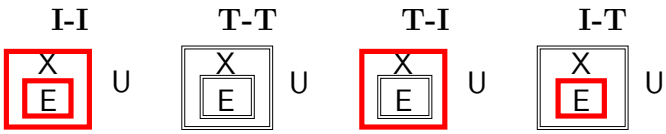

Fig. 2. Interactive interfaces and transactional interfaces .

Let us recall that we are defining how U observes E, even though the observation has to be made indirectly through X. In our definition of refinement we quantify over all X in some set of contexts Ξ. Clearly X acts as an intermediate 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 $X \in \Xi$ 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 that our contexts Ξ may perform that do not synchronise with any action of the entity E. Using these actions we can easily construct contexts $\widehat{\mathsf{X}}$ that after synchronising with E perform 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: [4]

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

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.

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.

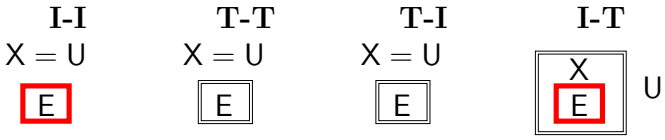

Fig. 3. With a sufficiently large set of contexts Ξ

Later we will give more concrete examples of all four cases in Fig. 3, and in the **I-I**, **T-T** and **T-I** cases (left-hand three cases of Fig. 3) 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.

It is only the **I-T** case (right hand case in Fig. 3) that requires both interfaces (and we deal with this case in Section 4.2). Elsewhere we will show that it is safe to consider the context as the user. But this does not mean that explicit contexts are not useful to formalise actual interfaces of all four types. As we will see later when we formalise interfaces with various types of interaction, we will do so by restricting what constitutes both a valid entity and a valid context.

[4] In the relational semantics of [12,13] 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.

2.4 Transition—moving from general to special theories

So far we have introduced and discussed our general theory of refinement together with a consideration of the sorts of interfaces that have to be dealt with in typical systems. In what follows, we first have a section (which can be skipped initially if desired) which takes a closer look at what is meant by determinism. This crops up in various places in the rest of the paper and needs careful treatment, hence its initial presentation in a section of its own.

In the three sections following the next section we look at several special theories, i.e. specialisations of the general theory, which should be familiar as particular systems which arise when developing software. The theme running through these sections is that operational semantics standing alone does not tell the whole semantic story. By considering what contexts we need to deal with and what observations can subsequently be made in those contexts we "complete" the meaning of the operational semantics and in particular fix the notion of refinement for it.

3 Determinism

This section looks in some detail at particular ways that determinism has been treated in various places. It will turn out that determinism has a crucial role when we look at refinement of processes in later sections. Indeed, these sections, having introduced refinement, mainly concentrate on determinism and its role. When we consider that making progress towards more and more deterministic processes is a large part of what refinement for processes means, this is not so surprising. Also, because we build a general model that permits us to compare determinism from different special points-of-view we can see determinism in a very wide perspective.

This section, then, is necessary if we wish to give a complete picture; at the same time, on first reading perhaps, this section can be skimmed in order to gain some familiarity with the problems without dwelling on the technical details.

Determinacy has some very varied definitions in the literature and is particularly difficult to define in a satisfactory way on process models such as CSP and CCS where the models have abstracted away the difference between one action $\overline{\mathsf{ba}}$—"pushing button a" causing another action ba—"button a is pushed". The definitions of determinism in CSP [32, p217] and CCS [24, p233] are dependent on the process equality they use (which we denote by $=_{pe}$) and can be stated as: P is deterministic if $x \xrightarrow{\mathsf{a}}_{\mathsf{P}} y$ and $x \xrightarrow{\mathsf{a}}_{\mathsf{P}} z$ imply $y =_{pe} z$.

Using this standard definition of determinism P and Q in Fig. 4 are deterministic processes and nP is not. But, as we will discuss later, if this definition

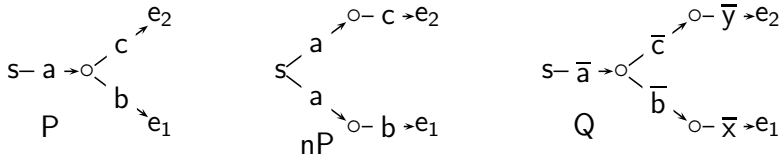

Fig. 4. Is Q deterministic?

is applied to broadcast processes we do not get the results we would expect, or the definition that appears in the literature, so a different definition must be found if it is going to give the desired results when instantiated into a concrete model of broadcast processes.

In fact our definition is going to be no more than a formalisation of Milner's [24, p232] comments 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."

Before we look at our formal definition let us apply Milner's comment to Q in Fig. 4. We assume that $\mathsf{P} \parallel_{\{\mathsf{a},\mathsf{b},\mathsf{c}\}} \mathsf{T}$ (i.e. P communicating via a, b and c with some test T) is a valid experiment. But if we perform this experiment twice with Q as the test, i.e. we perform $\mathsf{P} \parallel_{\{\mathsf{a},\mathsf{b},\mathsf{c}\}} \mathsf{Q}$, we do not necessarily get the same result, or behaviour, each time. Thus, following Milner's comment, Q should not be thought deterministic.

To formalise Milner's comment we first define the "same behaviour" or deterministic behaviour. If by starting from the same state and following the same sequence of actions, the process in question always finishes in the same state then we say it exhibits deterministic behaviour.

Definition 3.1 Deterministic behaviour, $det\_beh_{\Xi}(\mathsf{X})$: for any process given by LTS $\mathsf{X} = (N_{\mathsf{X}}, s_{\mathsf{X}}, T_{\mathsf{X}})$ with ρ any sequence of observable actions and $\{n, r, t\} \subseteq N_{\mathsf{X}}$:[5]

$$det\_beh_{\Xi}(\mathsf{X}) \triangleq n \stackrel{\rho}{\Longrightarrow} r \wedge n \stackrel{\rho}{\Longrightarrow} t \rightarrow r =_{\Xi} t$$

We will refer to an entity X as deterministic in Ξ if when placed in any context $[\_]_x \in \Xi$, $[\mathsf{X}]_x$ behaves deterministically.

Definition 3.2 An LTS X is **deterministic in** Ξ, written $det_{\Xi}(\mathsf{X})$, is given by:

[5] $\Longrightarrow$ is the observational semantics for X.

$$det_{\Xi}(\mathsf{X}) \triangleq \forall\, x \in \Xi.\, det\_beh_{\Xi}([\mathsf{X}]_x)$$

When the set of experiments is the set of deterministic contexts it is our interpretation of Milner's comment that a deterministic entity must:

behave deterministically in any deterministic context *DET*

When both entities and contexts are of the same kind (see Section 5) this cannot be used as a definition (since it is "circular"), nonetheless given a set D of deterministic systems DET is a property that can be checked:

$$\forall\, \mathsf{X} \in D.\, \forall\, x \in D.\, det\_beh_{\Xi}([\mathsf{X}]_x) \qquad \textit{Det-Cond}$$

Restricting the domain of the relational semantics to D, written $[\![\_]\!]_D$, means that an entity E is deterministic in D if and only if $[\![\mathsf{E}]\!]_D$ is a function. We choose to view deterministic entities as *implementations*. The meaning of a specification can be given by the set of implementations that satisfy the specification and with this interpretation it is again reasonable to view refinement as completing a semantics for specifications (see [17] for details and discussion).

We will consider examples of determinism where E is: one, an ADT with interaction via method calling (Section 4.2, Section 4.1); two, a process with interaction via output enabled broadcast (Section 5.1); and three, a state-based operation with interaction via shared memory (Section 6). In all of these three models of interaction our definition of determinism is the same as, or differs in uninteresting ways from, definitions of determinism found in the literature. A difference does occur, though, when we consider process algebraic, handshake-style interaction in Section 5.3.

4 Refinement for abstract data types

An ADT is an entity that interacts with programs. A program is a linear, unbranching sequence of actions which we can formally represent by an LTS, but for convenience we will refer to such an LTS by the sequence of actions themselves. The LTS can always be constructed from this sequence.

Each ADT-program interaction is a call to one of the ADT operations. Clearly an ADT-program interface is interactive. But the program-user interface could be either interactive or transactional. We will model sequential programs with a transactional program-user interface in Section 4.2 and with

an interactive program-user interface in Section 4.1. These constitute two *distinct* types of programs, *transactional programs* and *interactive programs*.

We assume the program-ADT interface to be interactive and private, i.e. it cannot be observed by the user. We further assume that the successful termination of the program can be achieved only if the ADT operations never fail to terminate. Thus if an ADT operation fails to terminate then the program must also fail to terminate.

We will give ADTs an event-based LTS semantics. See Fig. 5 for examples of the LTS semantics of two ADTs that we will use to illustrate the difference that the program-user interface makes to refinement.

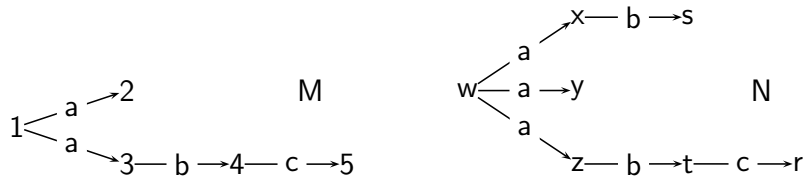

Fig. 5. M and N

4.1 Interactive user-program interface

If the user-program interface is interactive (case **I-I** in Fig. 3) then there is need for only one interface: that between the ADT and the program (user). As the program-ADT interface is interactive the program (user) can observe when each individual operation succeeds even if the program never terminates. Consequently for ADT entities we restrict the contexts to programs $(\overline{Names})^*$ thus:

Definition 4.1

$$\Xi_{\mathsf{P}} \triangleq \{[\_]_x \mid x \in (\overline{Names})^*\}$$

and

$$\mathsf{T}_{\mathsf{P}} \triangleq \{\mathsf{A} \ an\ LTS \mid \alpha\mathsf{A} \subseteq Names\}$$

With an interactive program-user interface information can be passed to the user while the program is running. Thus we allow the program to have any number of operations in this interface and the user can be informed of each successful operation of the ADT, even if the program subsequently fails to terminate.

Refinement in this case can be formalised by applying Definition 2.1 with contexts Ξ_P and observation function Tr^c, which gives complete traces, i.e. a

set over $Names^*$. See [6,27,7] for examples of such definitions. So, specialising our general refinement in this way we have:

Definition 4.2

$$\mathsf{A} \sqsubseteq_{Ip} \mathsf{C} \triangleq \forall\, x \in (\overline{Names})^*.\, Tr^c([\mathsf{C}]_x) \subseteq Tr^c([\mathsf{A}]_x)$$

There are two points to make for future reference:

(i) Looking at the example ADT in Fig. 5 with an interactive program-user interface we find M cannot be refined into N as:

$$(\mathsf{a}, \mathsf{b}) \notin Tr^c([\mathsf{M}]_{(\overline{\mathsf{a}},\overline{\mathsf{b}},\overline{\mathsf{c}})}) \text{ and } (\mathsf{a}, \mathsf{b}) \in Tr^c([\mathsf{N}]_{(\overline{\mathsf{a}},\overline{\mathsf{b}},\overline{\mathsf{c}})})$$

(ii) The deterministic contexts of an ADT are the programs $(\overline{Names})^*$ and hence from *DET* (Section 3) an ADT is deterministic if and only if the relation between programs and traces is a function. This is equivalent to the definition of deterministic transition diagrams or deterministic finite automaton:

> "no symbol can match the labels of two edges leaving one state" [4]
> *DFA*

Given that the context (a program) always decides upon a unique action (label on an edge) to attempt, then the behaviour of the system consisting of the program and deterministic ADT is determined.

4.2 Transactional user-program interface

Now we consider the case **I-T** (Fig. 3) where the program-user interface is transactional. Many definitions of ADT refinement [10,40,2,11,12] are based on the idea that observations are made only initially and, if the program terminates, at the point of termination.

In our transactional program-user interface we restrict the actions to $\bullet$. We use $\bullet$ to indicate that the program has started or has ended, so it appears no more than twice.

Data refinement with a transactional program-user interface can be formalised by applying Definition 2.1 with contexts Ξ_P as above and observation function Tr^c, but this time we note that Tr^c returns sets of sequences of length at most two (see [10,40,11,12,2] for examples of similar definitions). Two $\bullet$s appear if the program starts and terminates, but one $\bullet$ is absent if it starts but does not terminate. We have:

Definition 4.3

$$\mathsf{A} \sqsubseteq_{Tp} \mathsf{C} \triangleq \forall\, x \in (\bullet, (\overline{Names})^*, \bullet).\, Tr^c([\mathsf{C}]_x) \subseteq Tr^c([\mathsf{A}]_x)$$

There are two points to be made here also:

(i) With a transactional program-user interface we can show that M (Fig. 5) can be refined into N, as we can see by constructing the relevant relations:

$$[\![\mathsf{M}]\!] = \{((\bullet,\bullet),\{(\bullet,\bullet)\}),((\bullet,\overline{\mathsf{a}},\bullet),\{(\bullet,\bullet)\}),((\bullet,\overline{\mathsf{a}},\overline{\mathsf{b}},\bullet),\{(\bullet,\bullet),(\bullet)\}),$$
$$((\bullet,\overline{\mathsf{a}},\overline{\mathsf{b}},\overline{\mathsf{c}},\bullet),\{(\bullet,\bullet),(\bullet)\})\} \cup$$
$$\{(x,\{(\bullet)\}) \mid \forall\, x \notin \{(\bullet,\bullet),(\bullet,\overline{\mathsf{a}},\bullet),(\bullet,\overline{\mathsf{a}},\overline{\mathsf{b}},\bullet),(\bullet,\overline{\mathsf{a}},\overline{\mathsf{b}},\overline{\mathsf{c}},\bullet)\}\}$$

and by inspection $[\![\mathsf{M}]\!] = [\![\mathsf{N}]\!]$.

Thus $\sqsubseteq_{Tp}$ and $\sqsubseteq_{Ip}$ are not the same, as can be seen from the examples in Fig. 5, where $M \sqsubseteq_{Tp} N$ but not $M \sqsubseteq_{Ip} N$. This was first mentioned in [30].

(ii) It can be seen that applying Definition 3.2 to picks out the same set of ADTs as being deterministic whether we apply Definition 3.2 to ADTs that can be placed in interactional program-user interfaces or ADTs that can be placed in transactional program-user interfaces.

5 Processes

Both processes and ADTs can be given an event-based semantics, but process and ADT refinement are not the same. Programs can be modelled by an unbranching sequence of operation calls to an ADT and programs are the only valid contexts for an ADT, whereas processes can be placed in branching contexts. Thus processes will be given a distinct semantics to ADTs because of the change of contexts in which they can be placed [27].

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 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.

As we saw above, our definition of determinism, Definition 3.2, when applied to ADTs is the same as the informal *DFA* characterisation in Section 4.1. But we shall see that the *DFA* characterisation and our definition are not the same when applied to handshake processes. It is the *DFA* characterisation that is equivalent to, or used as, the definitions of deterministic handshake processes as found in [32,18,5,24] and deterministic broadcast processes as

found in [23]. As we shall show, the *DFA* definition does not always correspond to what we might reasonably think to be deterministic processes.

The reader may be perplexed about the time we spend talking about determinism in what follows (while the definitions of refinement are so simple). The point is that because we build a general model that permits us to compare determinism from different special models, we can see determinism from a very wide perspective, which we take advantage of.

In Section 5.1 we review broadcast processes, and consider what broadcast processes are deterministic, then in Section 5.3 we do the same for handshake processes.

Our interactive branching programs of Section 5.5 are classified as processes because they and their contexts can both branch. These programs (or processes) can be viewed as a restricted class of handshake processes which have active and passive events.

Since we are no longer dealing with transactional interfaces, we need make no distinction between context and user in what follows. Further, there is no longer any distinction between entities and contexts, in the sense that for any context $[\_]_X$, X is also an entity. Both entities and contexts are simply processes.

5.1 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 [14]. 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 all 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.

Because broadcast interactions are fundamentally different from the other interactions we consider we write $a!$ for $\overline{a}$ and $a?$ for a simply to remind the reader how to interpret events that appear in the figures.

Even restricting communication to point-to-point there is a variety of different ways to formalise broadcast communication. Some models of broadcast

systems [35,25,20,15] define parallel composition in such a way that output events cannot be blocked. The alternative approach, found in [36,26,33,22,21] and used here, is to keep parallel composition the same as defined for hand-shake operations and consider only entities, and thus contexts, that have input actions always enabled.

Definition 5.1

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

and

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

We can now apply Definition 2.1 (our general definition of refinement), based as it is on a widely accepted informal definition of refinement, to obtain a definition of the refinement of processes with broadcast interaction:

Definition 5.2

$$\sqsubseteq_{\mathsf{BC}} \triangleq \sqsubseteq_{\Xi_{\mathsf{BC}}}$$

5.2 Determinism and broadcasting

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

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 a? is not enabled:

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

It is frequently clearer to not show listening loops (see Fig. 6). Such LTSs can be interpreted as broadcast processes by leaving listening loops implicit.

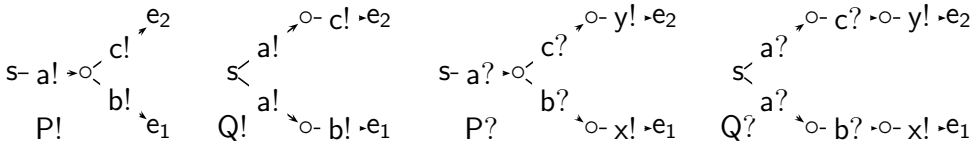

Fig. 6. $M_{BC}(\mathsf{P!}) =_{BC} M_{BC}(\mathsf{Q!})$ and $M_{BC}(\mathsf{P?}) \neq_{BC} M_{BC}(\mathsf{Q?})$

The effect of M_{BC} can be best understood by considering some examples. Consider Fig. 6. 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 b? event after the initial a? event then it must output x! not y! but $\mathsf{a?b?c?y!} \in M_{BC}(\mathsf{Q?})$ as the process, on hearing a?, can make one of two moves, one of which will lead to output y! being made. This is not the result that might be expected from

the handshake perspective where trace semantics are unable to distinguish P? and Q?.

P! can broadcast either b! or c!. As broadcast output is under local control no other process can block either of these events. Hence it seems unavoidable that we consider P! to be non-deterministic. Yet clearly P! and $M_{BC}(\mathsf{P!})$ are deterministic transition systems according to the informal *DFA* characterisation.

Clearly there is a mismatch between our intuitions on the one hand and the *DFA* characterisation on the other hand. Because of this mismatch we turn to the *DET* characterisation. Unfortunately as entities and contexts are the same type of thing *DET* cannot be turned into a definition (recall Section 3) but having defined a set of deterministic broadcast processes we can check they satisfy *Det-Cond*.

We define the set of deterministic broadcast processes, as in [36,26], as processes, ignoring listening loops (prior to applying M_{BC}), that branch on only input events with different names (and where $Names^?$ is $\{\mathsf{a}^? \mid \mathsf{a} \in Names\}$):

Definition 5.3 The set of deterministic broadcast processes, D_{BC}:

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

We leave for the interested reader to check that D_{BC} satisfies *Det-Cond* but draw the reader's attention to the fact that the definition of determinism in [36,26] is consistent with our abstract definition in Definition 3.2. In CBS all sequential processes are deterministic: "Speakers in parallel are the only source of non-determinism in CBS" [26]. An informal justification for this limitation is that branching outputs of a sequential process could not be implemented.

5.3 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 entities are:

Definition 5.4

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

and

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

Thus, our general refinement (Definition 2.1) specialises, for these processes, to little more than a rewording of must testing semantics ([9]) and, as has been shown in [27], it is equivalent to failure refinement. We put:

Definition 5.5

$$\sqsubseteq_{\mathsf{PA}} \triangleq \sqsubseteq_{\Xi_{\mathsf{PA}}}$$

5.4 Determinism and handshaking

Our definition of deterministic processes, just as for broadcast processes in Section 5.1, is very different to the *DFA* characterisation that is found in the process algebraic literature. We consider two simple examples of processes to investigate this.

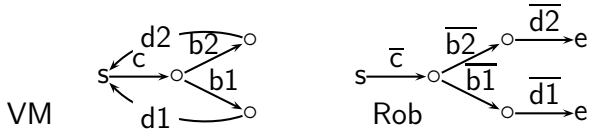

Fig. 7. Are VM and Rob deterministic?

The vending machine VM in Fig. 7 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 Rob in Fig. 7 requires some thought.

Non-determinism can arise naturally with concurrent processes, for example running processes $\mathsf{R1} \triangleq \overline{\mathsf{c}};\overline{\mathsf{b1}}$ and $\mathsf{R2} \triangleq \overline{\mathsf{b2}}$ in parallel with VM. The two processes R1 and R2 *race* to push different buttons and which button is pushed is not determined. We accept Hoare's view [18, 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.

Further, note that process algebras have chosen to ignore both time and causality, whereas broadcast systems have chosen to ignore just time.

In CSP, CCS and ACP Rob is deterministic but exhibits non-deterministic behaviour when interacting with VM. 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 there 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, 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 5.1 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.

Although the implement ability of processes such as Rob is not discussed in process algebras such as CSP, CCS or ACP it is well-known that such simple processes can be coded in the Occam programming language. As these concurrent processes can be executed on a single transputer and as transputers, like other digital computers, are finite-state deterministic machines they cannot exhibit non-deterministic[6] behaviour and consequently the Occam compiler has to determine which button is pushed. This could be described as the Occam compiler refining $[\mathsf{Rob}]_{\mathsf{VM}}$ and then implementing the "deterministic process" produced by the refinement. For this reason we view as *not imple-*

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

mentable the interpretation of Rob given by the process algebras cited.

This is the only model in which our definition 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 that we quoted in Section 3.

5.5 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, IBP (interactive branching programs) from [29], that combines process and program ideas.

It is common in the literature on handshake events ([18,32,24,5]) to treat events that synchronise in exactly the the same way, and not differentiate between 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. 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. 8 for an example of how this looks) and after this τ event only the single active event can be executed.

Definition 5.6

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

where

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

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

We put:

$$\sqsubseteq_{\mathsf{IBP}} \triangleq \sqsubseteq_{\Xi_{\mathsf{IBP}}}$$

5.6 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, $\overline{\mathsf{a}}$.

Definition 5.7 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

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

The IBP process $M_{IBP}(\mathsf{Rob})$ (Fig. 8) is a non-deterministic specification of the behaviour of Rob in Fig. 7 where the non-determinism arises from not specifying which button the robot will push.

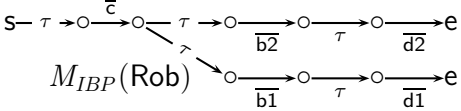

Fig. 8.

We informally define the set of deterministic IBP in the same way as the deterministic broadcast processes in Section 5.1. The deterministic IBP are the processes, prior to applying M_{IBP}, that branch on only passive events with different names.

Definition 5.8 The set of deterministic IBP, D_{IBP}:

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

$$\exists \overline{\mathsf{x}}, t. m \xrightarrow{\overline{\mathsf{x}}}_{\mathsf{P}} t \wedge (\tau = \mathsf{z} \wedge n = m \wedge \overline{\mathsf{y}} = \overline{\mathsf{x}} \wedge r = t)$$

$$\wedge\ n \xrightarrow{\mathsf{x}}_{\mathsf{P}} m \wedge n \xrightarrow{\mathsf{y}}_{\mathsf{P}} k \Rightarrow$$

$$(\mathsf{x} = \mathsf{y} \wedge m = k) \vee (\mathsf{x} \neq \mathsf{y} \wedge \mathsf{x} \in Act \wedge \mathsf{y} \in Act)\}$$

$[\mathsf{Rob}]_{\mathsf{VM}}$ (taking Rob and VM from Fig. 7) and both $[M_{IBP}(\mathsf{Rob})]_{M_{IBP}(\mathsf{VM})}$ and $M_{IBP}(\mathsf{Rob})$ (see Fig. 8) 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. 7.

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 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 *Det-Cond* but draw the reader's attention to the fact that this definition of determinism is consistent with our abstract definition in Definition 3.2. Thus IBP is a subset of handshaking-style processes in Section 5.3 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.

6 Operation refinement

This final section in this sequence of special models is something of an oddity as previously we have considered an entity as a set of operations (or events) but here our entity is a single entity. In addition what can be "seen" in this section are states from a set *State*.

Our entity E could be a method, procedure, function etc., its context X an ADT in some particular state and the user U a program. As the method-ADT interface is transactional (cases **T-T** and **T- I** in Fig. 2) we can view the context and user as the same entity with a single interface between it and the operation. All that is in the interface between an operation and its context is the state of the ADT. Thus we define contexts to be states and and operations move us from states to states, and what we can observe are the sequences of states we move through. Also, given this rather different setting, "putting in a context", written as ever as $[\_]_x$, will simply mean *applying* the operation to the context (current state) so as to allow us to see the new state we end up in.

The only things we can observe are sequences of what can be thought of as alternating pre-states and post-states as the use of an operation takes us from start state (context) to resulting state. In this way of modelling, we can think of our entity E as an operation in an ADT that can store information in a local (state) variable of type *State*. For example an operation E that maps the initial state $\mathsf{y} \in$ *State*, to a final state $\mathsf{x} \in$ *State* but fails to terminate from

initial state x is modelled by a relation e.g. $\mathsf{E} \triangleq \{(\mathsf{y}, \mathsf{x})\}$.

The detailed interpretation of an operation is formalised, as for previous examples, by fixing the contexts and observations. For example, we may allow contexts to be any *State* and restrict the observations to being pre- post state pairs, i.e. sequences of length two only.

Definition 6.1

$$\Xi_{\mathsf{isoz}} \triangleq \{[\_]_x \mid x \in State\}$$

and

$$\mathsf{T}_{\mathsf{isoz}} \triangleq \mathbb{P}(State \times State) \qquad O_{\mathsf{isoz}}([\mathsf{E}]_{\mathsf{x}}) \triangleq \{(\mathsf{x}, \mathsf{y}) \mid (\mathsf{x}, \mathsf{y}) \in \mathsf{x}, \mathsf{E}\}$$

Hence the operational semantics of E is represented by the relation:

$$[\![\mathsf{E}]\!]_{(\Xi_{\mathsf{isoz}}, O_{\mathsf{isoz}})} = \{(\mathsf{y}, (\mathsf{y}, \mathsf{x}))\}.$$

The "usual" relational semantics, (think of a Z or B operation, for example) is a relation from state to state. The advantage of state-to-state relations is that the semantics of a sequence of operations can be defined to be the sequential composition of the state to state semantics of the individual operations.

We can easily transform the relations above into state-to-state (partial) relations by simply removing the redundant pre-state from the observation. Thus $\{(\mathsf{y}, (\mathsf{y}, \mathsf{x}))\}$ becomes $\{(\mathsf{y}, \mathsf{x})\}$. This operational semantics is that used in ISO Z [1] (hence the subscript we have been using) and is silent as to whether the operation terminates or not.

7 Conclusions

Our definition of general refinement presented in this paper is parametrised on the set of contexts an entity can be placed in, and the observations that can be made by a user of the system thus formed.

We made use of contexts in distinct ways:

(i) Since general refinement has contexts Ξ as a parameter, by changing Ξ we were able to model different types of interaction [27];

(ii) We distinguished two sets of contexts for abstract data types, ADT: the *interactive programs*, with an interactive program-user interface; and the *transactional programs*, with a transactional program-user interface. Each set of contexts gives rise to a distinct refinement.

We also saw that consideration of what is meant by determinism was needed in order to give a full account of refinement.

In a companion paper [31] we will continue the story with a generalisation, which we call *vertical refinement* of what, in the literature, has been called action refinement or non-atomic refinement. By viewing a special model as a logical theory, vertical refinement will be seen as a theory morphism, formalised as a Galois connection.

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.

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] 13568, I., "Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics," Prentice-Hall International series in computer science, ISO/IEC, 2002, first edition.

[2] Abrial, J.-R., "The B-book: assigning programs to meanings," Cambridge University Press, New York, NY, USA, 1996.

[3] Abrial, J.-R., D. Cansell and D. Méry, *Refinement and reachability in Event B*, in: H. Treharne, S. King, M. C. Henson and S. Schneider, editors, *ZB05: Formal Specification and Development in Z and B*, Lecture Notes in Computer Science **3455** (2005), pp. 222–241.

[4] Aho, A. V., R. Sethi and J. D. Ullman, "Compilers: Principles, Techniques, and Tools," Addison-Wesley, 1986.

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

[6] Bolton, C. and J. Davies, *A singleton failures semantics for Communicating Sequential Processes*, Research Report PRG-RR-01-11, Oxford University Computing Laboratory (2001). URL citeseer.nj.nec.com/bolton01singleton.html

[7] Bolton, C. and J. Davies, *A singleton failures semantics for communicating sequential processes*, Formal Aspects of Computing **18** (2006), pp. 181–210. URL http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/s00165-005-0081-x

[8] Brinksma, E., A. Rensink and W. Vogler, *Applications of fair testing*, FORTE **69** (1996), iFIP Conference Proceedings.

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

[10] de Roever, W.-P. and K. Engelhardt, "Data Refinement: Model oriented proof methods and their comparison," Number 47 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1998.

[11] Derrick, J. and E. Boiten, "Refinement in Z and Object-Z: Foundations and Advanced Applications," Formal Approaches to Computing and Information Technology, Springer, 2001.
URL http://www.cs.ukc.ac.uk/pubs/2001/1200

[12] Derrick, J. and E. Boiten, *Relational concurrent refinement*, Formal Aspects of Computing **15** (2003), pp. 182–214.
URL http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/s00165-003-0007-4

[13] Dunne, S. and S. Conroy, *Process refinement in B*, in: H. Treharne, S. King, M. C. Henson and S. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users*, Lecture Notes in Computer Science **3455** (2005), pp. 45–64.

[14] Ene, C. and T. Muntean, *Expressiveness of Point-to-Point versus Broadcast Communications*, in: *Fundamentals of Computation Theory, 12th International Symposium FCT'99*, Lecture Notes in Computer Science **1684** (1999).

[15] Ene, C. and T. Muntean, *Testing Theories for Broadcasting Processes* (2004), submitted for publication, http://www.esil.univ-mrs.fr/.

[16] 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

[17] Hehner, E. C. R. and A. M. Gravell, *Refinement semantics and loop rules*, in: *World Congress on Formal Methods (2)*, 1999, pp. 1497–1510.
URL citeseer.nj.nec.com/328472.html

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

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

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

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

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

[23] Lynch, N. and F. Vaandrager, *Forward and backward simulations, part i: Untimed systems.*, Information and Computation 121(2) (1995), pp. 214–233.

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

[25] Prasad, K. V. S., *A calculus of value broadcasts*, in: *Parallel Architectures and Languages Europe*, 1993, pp. 391–402.
URL citeseer.nj.nec.com/article/prasad93calculus.html

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

[27] 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.

[28] Reeves, S. and D. Streader, *Atomic Components*, in: Z. Liu and K. Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004: First International Colloquium*, Lecture Notes in Computer Science **3407** (2004), pp. 128–139.
URL http://www.springerlink.com/openurl.asp?genre=article&id=N305G7A8GK2XDMBQ

[29] Reeves, S. and D. Streader, *Constructing programs or processes*, Journal of Universal Computer Science **11** (2005), pp. 2034–2045.
URL http://www.jucs.org/jucs_11_12/constructing_programs_or_processes

[30] Reeves, S. and D. Streader, *State- and Event-based refinement*, Technical report, University of Waikato (2006), computer Science Working Paper Series 09/2006, ISSN 1170-487X.
URL http://researchcommons.waikato.ac.nz/cms_papers/12/

[31] Reeves, S. and D. Streader, *General refinement, part two: flexible refinement*, Proceedings of Refine 2008, Electronic Notes in Theoretical Computer Science (2008).

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

[33] Segala, R., *A Process Algebraic View of I/O Automata*, Technical Report MIT/LCS/TR-557, Massachusetts Institute of Technology (1992).
URL citeseer.ist.psu.edu/article/segala92process.html

[34] Spivey, J. M., "The Z notation: A reference manual," Prentice-Hall International series in computer science, Prentice Hall, 1992, 2nd. edition.

[35] Tretmans, D., "A Formal Approach to Conformance Testing," Ph.D. thesis, Faculteit der Informatica (1992).

[36] Vaandrager, F. W., *On the relationship between process algebra and input/output automata*, in: *Logic in Computer Science*, 1991, pp. 387–398.
URL citeseer.ist.psu.edu/vaandrager91relationship.html

[37] Valmari, A. and M. Tienari, *Compositional Failure-based Semantics Models for Basic LOTOS*, Formal Aspects of Computing **7** (1995), pp. 440–468.
URL citeseer.nj.nec.com/valmari95compositional.html

[38] van Glabbeek, R. J., *Linear Time-Branching Time Spectrum I*, in: *CONCUR '90 Theories of Concurrency: Unification and Extension*, LNCS 458 (1990), pp. 278–297.

[39] van Glabbeek, R. J., *The Linear Time - Branching Time Spectrum II*, in: *International Conference on Concurrency Theory*, 1993, pp. 66–81.
URL citeseer.nj.nec.com/vanglabbeek93linear.html

[40] Woodcock, J. and J. Davies, "Using Z: Specification, Refinement and Proof," Prentice Hall, 1996.

Appendix

The basics of operational semantics

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, usually denoted by the symbol δ).

We assume a universe containing a set of names *Names* and their "matches" $\overline{Names} \triangleq \{\overline{\mathsf{a}} \mid \mathsf{a} \in Names\}$. *Names* will be used to give names to operations in a state-based system and names to events in an event-based system. In each case we need the matches to express the notion of "caller and called", or "passive and active". Note that if $\overline{a} \in \overline{Names}$, then $\overline{\overline{a}} = a \in Names$. A special event τ is introduced that models an event that can neither be seen nor blocked. We define $Names^{\tau} \triangleq Names \cup \overline{Names} \cup \{\tau\}$.

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 .1 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

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

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 the 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 [18,32], CCS [24], ACP [5], for broadcast systems IOA [22], CBS [26], for abstract data types [6] and for objects [12].

Definition .2 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

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

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 $\exists\, m.(n, \mathsf{a}, m) \in T_{\mathsf{A}}$. •

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 .3

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

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

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

Although in the body of the chapter we define our general model on the event-based operational semantics, the results can equally be applied to many state-based models, such as Event B, simply by using the mappings in Definition .3 to translate the semantic models where needed.

Since we use the event-based model to do most of the work in the chapter, we need further notational and definitional work, as follows.

As usual in the event-based world we formalise τ operations as unobservable by defining an observational semantics possessing no τ events.

Definition .4 An observational semantics $\Longrightarrow_{\mathsf{A}}$ for LTS A is given, where $\mathsf{x} \in Names \cup \overline{Names}$, by:

$$n \overset{\mathsf{x}}{\Longrightarrow}_{\mathsf{A}} m \triangleq \exists\, n', m'.n \overset{\tau}{\Longrightarrow}_{\mathsf{A}} n', n' \xrightarrow{\mathsf{x}}_{\mathsf{A}} m', m' \overset{\tau}{\Longrightarrow}_{\mathsf{A}} m$$

where

$$s \overset{\tau}{\Longrightarrow}_{\mathsf{A}} t \triangleq (\forall\, n > 0.\, \exists\, s_1, ..., s_{n-1}.s \xrightarrow{\tau}_{\mathsf{A}} s_1, s_1 \xrightarrow{\tau}_{\mathsf{A}} s_2, \ldots s_{n-1} \xrightarrow{\tau}_{\mathsf{A}} t) \vee s = t$$

Also

$$Abs(\mathsf{A}) \triangleq (N_{\mathsf{A}}, s_{\mathsf{A}}, \{n \xrightarrow{x} m \mid n \overset{x}{\Longrightarrow}_{\mathsf{A}} m\})$$

•

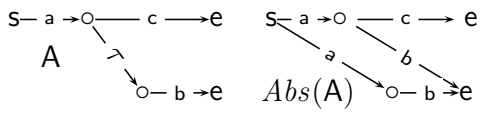

Fig. .1. Event abstraction

See Fig. .1 for an example.

Our observational semantics is not the same as in CCS [24] but, like CSP, has the advantage that it "succeeds in total concealment of internal events" [19, p.198]. Our definition comes from [8,37], and is very slightly different from the operational semantics of CSP simply because they provide the intuition in the original design whereas "the operational semantics of CSP was created to give an alternative view to the already existing denotational models" [32, p.178]. For a more detailed comparison and discussion of abstracting τ loops see [28].

Parallel composition is defined, as in CCS, to represent the point-to-point private communication between concurrent entities.

Definition .5 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 $\mathsf{x} \in Names^\tau$:

$$\frac{n \xrightarrow{\mathsf{x}}_\mathsf{A} l, \mathsf{x} \notin S \cup \overline{S}}{(n,m) \xrightarrow{\mathsf{x}}_{\mathsf{A}\|_S\mathsf{B}} (l,m)} \qquad \frac{n \xrightarrow{\mathsf{x}}_\mathsf{B} l, \mathsf{x} \notin S \cup \overline{S}}{(m,n) \xrightarrow{\mathsf{x}}_{\mathsf{A}\|_S\mathsf{B}} (m,l)} \qquad \frac{n \xrightarrow{\mathsf{a}}_\mathsf{A} l, m \xrightarrow{\overline{\mathsf{a}}}_\mathsf{B} k, \mathsf{a} \in S \cup \overline{S}}{(n,m) \xrightarrow{\tau}_{\mathsf{A}\|_S\mathsf{B}} (l,k)}$$

$$\mathsf{A} \|_S \mathsf{B} \triangleq (N_{\mathsf{A}\|_S\mathsf{B}}, s_{\mathsf{A}\|_S\mathsf{B}}, T_{\mathsf{A}\|_S\mathsf{B}})$$

Our definition of the entity context interface requires synchronisation on Act, all possible events in the entity.

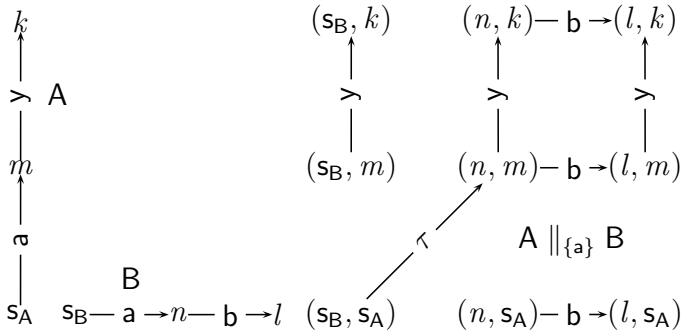

Fig. .2. Example $\mathsf{B} \|_{\{\mathsf{a}\}} \mathsf{A}$

Our parallel composition with synchronisation operator $\_\|_S\_$ enforces private communication between its operands on all events in the synchronisation set S. Thus any event in S that appears in one of the operands must either synchronise with an event from the other operand or be *blocked*. This can be understand by considering the example in Fig. .2.

To avoid confusion we assume that the event names that appear in each parallel component are unique. Thus when we write $\mathsf{A} \|_S \mathsf{B}$ we imply that $\alpha(\mathsf{A}) \cap \alpha(\mathsf{B}) = \varnothing$ and when we write $(\mathsf{A} \|_S \mathsf{B}) \|_T \mathsf{C}$ we imply that $S \cap T = \varnothing$.

Definition .6 Let ρ be a sequence of events. $|\ \rho\ |$ is the length of ρ, $()$ the empty sequence of events and $\mathsf{a} \lhd \rho$ the sequence built by adding the event a to the front of ρ. Write $\rho_0 \ll \rho$ for ρ_0 a prefix of ρ. We often write $\mathsf{a} \lhd ()$ as (a) and $\mathsf{a} \lhd (\mathsf{b})$ as (a, b) and so on.

Let $\forall\, n.n \xrightarrow{()} n,\ n \xrightarrow{\mathsf{a}\lhd\rho} o \triangleq \exists\, m.n \xrightarrow{\mathsf{a}} m \wedge m \xrightarrow{\rho} o,\ n \xrightarrow{\rho} \triangleq \exists\, m.n \xrightarrow{\rho} m$ and $\forall\, n.n \xRightarrow{()} n,\ n \xRightarrow{\mathsf{a}\lhd\rho} o \triangleq \exists\, m.n \xRightarrow{\mathsf{a}} m \wedge m \xRightarrow{\rho} o,\ n \xRightarrow{\rho} \triangleq \exists\, m.n \xRightarrow{\rho} m$

The observable traces of A are: $Tr(\mathsf{A}) \triangleq \{\rho \mid s_\mathsf{A} \xRightarrow{\rho}\}$. The complete observable traces of A are all the finite traces which take us to a state with no successor, and all the infinite traces, i.e. traces which have prefixes of every length:

$$Tr^c(\mathsf{A}) \triangleq$$

$$\{\rho \mid s_\mathsf{A} \xRightarrow{\rho} n \wedge \{\mathsf{a} \mid n \xRightarrow{\mathsf{a}}\} = \varnothing\} \cup \{\rho \mid s_\mathsf{A} \xRightarrow{\rho} \wedge\, \forall\, n.\, \exists\, \rho_0.\rho_0 \ll \rho \wedge |\ \rho_0\ | = n\}$$