Working Paper Series
ISSN 1177-777X

Flexible Refinement

Steve Reeves and David Streader

Working Paper: 02/2007
May 7, 2007

Flexible Refinement

Steve Reeves and David Streader

Department of Computer Science, University of Waikato, Hamilton, New Zealand
{stever,dstr}@cs.waikato.ac.nz

Abstract. To help make refinement more usable in practice we introduce a general, flexible 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.

Our general model is expressed in terms of an operational semantics, and by exploiting the well-known isomorphism between state-based relational semantics and event-based labelled transition semantics we were able to take particular models from both the state- and event-based literature, reflect on them and gradually evolve our general model. We are also able to view our general model both as a testing semantics and as a logical theory with refinement as implication.

Our general model can used as a bridge between different particular special models and using this bridge we compare the definition of determinism found in different special models. We do this because the reduction of nondeterminism underpins many definitions of refinement found in a variety of special models.

To our surprise we find that 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 in CCS and by formalising these intuitions we are able to define determinism in process algebra in such a way that it 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, that is an implementable subset of process algebra.

Later in the chapter we show explicitly how five special models, taken from the literature, are instances of our general model. This is done simply by fixing the sets of contexts and observations involved.

Next we define vertical refinement on our general model. Vertical refinement can be seen both as a generalisation of what, in the literature, has been called action refinement or non-atomic refinement. Alternatively, by viewing a layer as a logical theory, vertical refinement is a theory morphism, formalised as a Galois connection.

By constructing a vertical refinement between broadcast processes and interactive branching programs we can see how interactive branching programs can be implemented on a platform providing broadcast communication. But we have been unable to extend this theory morphism to implement all of process algebra using broadcast communication. Upon investigation we show the problem arises with the examples that caused the problem with the definition of determinism on process algebra.

Finally we illustrate the usefulness of our flexible general model by formally developing a single entity that contains events that use handshake communication and events that use broadcast communication.

1 Introduction

Refinement is the stepwise process of developing a specification towards, or perhaps into, an implementation. Each refinement step formalises a design decision and transforms a more abstract entity into a more concrete entity. 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 this spirit of flexibility, we might, for example, express properties of event-based models of entities within the refinement relation so as to allow development of nonterminating, interacting systems.

In order to express concepts most conveniently we give a characterisation of refinement at a very general level, which we discuss below, and then specialise this general theory to several (we give five examples) particular (special) theories, which we give later in the chapter.

Our general theory, in Section 2, centres around a parameterised 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 as presented in many places in the literature. Each of the particular models can in turn be seen as specialisations of the general theory, and so can many others. These various special theories come about by limiting the set of contexts and observations considered and the instantiation of various parameters. Notable examples of special models we deal with are: abstract data types (ADT); handshake processes such as in Communicating Sequential Processes (CSP, [1]) or the Calculus for Communicating Systems (CCS, [2]), or broadcast processes such as in the Calculus of Broadcasting Systems (CBS, [3]); and individual operations.

Our general refinement is formalised by a simple semantic model in terms of labelled transition systems (LTS). However, in order to combine the advantages of event-based and state-based approaches we are always going to keep in mind that there are mappings (both ways) between collections of named partial relations, a canonical state-based operational semantics, and LTS, a canonical event-based operational semantics. We keep this link in mind to avoid the results being constrained to the syntax of one particular pair of state- and event-based models.[1]

[1]See the Appendix for a summary of named partial relations and LTS definitions and the mappings between them, which are standard and appear widely throughout the literature. In this paper we will, it turns out, use the LTS semantics to work with, but the reader should never lose sight of the fact that we might just as well have used

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 [4], handshake processes with failure and trace semantics [1], broadcast semantics [3], etc. A similar situation exists in the state-based world where partial relations have been given many different interpretations. In Z and B [5–7] partial relations are interpreted as undefined outside of precondition and totally correct, while in [4] they are interpreted as guarded outside of precondition and totally correct, but in [8, Chapter 1-7] they are interpreted as partially correct.

The operational semantics can do this because both named partial relations and LTS are open to many different interpretations, so 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 [4, 1, 3, 5–8], different "meanings" have been given to the operational semantics 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", shown by $\_$, in. Informally speaking the context of an entity is no more than a definition of how the surrounding world interacts with the entity. Our general definition of refinement is thus parameterised by a set Ξ of possible contexts. 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.

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 [9, 10] 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 parameterised definition of refinement, i.e. select the set of contexts that an entity will be placed in, specialise the general definition by instantiating the context parameter accordingly and use the resulting special refinement. We have shown in [11] that this approach gives results that would be expected from the literature.

Let us re-cap: once you have decided on the set of contexts that your entities can be placed in you can use this set to construct an appropriate, specialised, definition of refinement. This definition of refinement is then used to complete the semantics of the LTS which will model your 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.

Figure Fig. 1 gives a picture of what we are about. On the left we have the general model. On the right, by specialising the general model in two different

the relational model. We may appear biased towards the event-based view, but we are not, in fact, since this is always immediately and trivially a state-based view too.

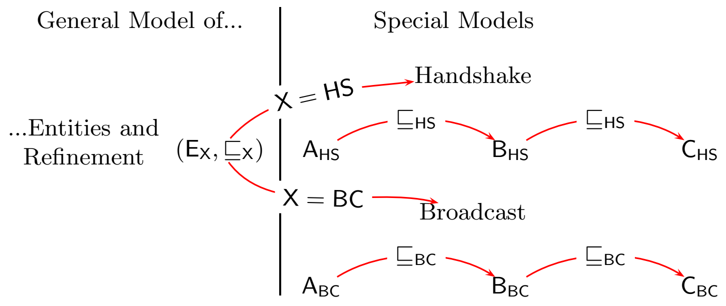


Fig. 1. Overview

ways (called HS and BC, to be explained later) we get two different special models (Handshake and Broadcast). Within each of these special models we have, as usual, the notion, as we move in steps from left to right, of moving from more abstract to more concrete entities (programs, processes etc.) via refinement.

As refinement is used to complete the meaning of the operational semantics the question "Which refinement, the one in [12] or the one in [13], is correct?" is not sensible. What it is helpful to ask is "In what situations is it safe to use the refinement in [12] and in what situations is it safe to use the refinement in [13]?"

What we wish to stress here is not, so much, this liberalisation, but that:

1. our semantic mapping between state-based and event-based operational semantics makes available a large number of event-based definitions of refinement in a large variety of state- and event-based models;
2. directly from our generalised definition of refinement between entities A and C we will have a clear statement of the contexts in which the concrete entity C is guaranteed to behave like the abstract entity A.

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 *nondeterminism* is defined in different special models. As we will see this apparently simple idea has proven difficult to define to everyone's satisfaction.

In Section 6 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 7 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 is the only one we consider that has abstracted away the *cause* and *response* nature of event synchronisation. Consequently, in Section 7.5, 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 nondeterminism can be removed from sequential entities by refinement.

In Section 8 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.

By the end of Section 9 we have developed a general model and shown that special instances of the general model give the results that we would expect from the literature, with the exception of determinism in the model of handshake processes (CSP/CCS). In this model our characterisation of determinism is, as we show, a formalisation of one of Milner's intuitions.

Before generalising even further we next introduce an additional sort of refinement, *visibility refinement*, a combination of extension refinement [14] and behavioural sub-typing [12], that can make visible (i.e. reverse hiding and restriction of various kinds) in the concrete entity events not visible in the abstract entity.

Having visibility refinement in our general model means we can use it in any of the "special models", including state-based models such as Z, B, and Event B. In Event B a very similar definition of refinement, with very few restrictions on the introduced events, has already been defined in [13]. Our definition, or rather the porting of a definition from [12], results in a refinement with no restrictions at all and hence is a liberalisation of the definition in [13].

In Section 10 we generalise our general model further by introducing *vertical refinement* between different special models. Viewing each 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 11. 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.

Finally in Section 11.1 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.

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

We will use the following natural notion of refinement that appears in many places in the literature [15, 8, 5, 16, 17, 7] 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.

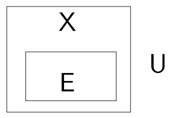


Fig. 2.

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 the set of events *Act* where $Act \subseteq Names$ (where *Names* is a set consisting of all possible event names). All E's events interact with X at the E—X interface (see Fig. 2). So, the events 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 events.

Recall we use $[\_]_{\mathsf{X}}$ to denote the context X and Ξ to denote some set of such contexts[2].

Our natural and common notion of refinement given above will be formalised in:

Definition 1 *Let Ξ be a set of contexts with which the entities A and C can communicate privately. Let Obs be a function defining what can be observed of the contexts:*

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

•

The definition of refinement is one of the central parts of this paper. It should be remembered that, as suggested by the picture Fig. 1, 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 specific sets of contexts.

Definition 2 *Entity equality is defined as:*

$$\mathsf{A} =_{(\Xi,Obs)} \mathsf{B} \triangleq \mathsf{A} \sqsubseteq_{(\Xi,Obs)} \mathsf{B} \wedge \mathsf{B} \sqsubseteq_{(\Xi,Obs)} \mathsf{A}$$

[2]Since we often use LTS as our canonical way of viewing entities, contexts and entities are often viewed as being composed by parallel composition. So:

$$[\_]_{\mathsf{X}} \triangleq \_ \parallel_{Act} \mathsf{X},$$

where X is some LTS, which we then gather together into sets of contexts:

$$\Xi \subseteq \{[\_]_{\mathsf{X}} \mid \mathsf{X} \text{ an LTS}\}$$

Further, let LTS $\mathsf{A} = (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$, $\{r, t\} \subseteq N_{\mathsf{A}}$ *then*

$$(r =_{(\Xi, Obs)} t) \triangleq (N_{\mathsf{A}}, r, T_{\mathsf{A}}) =_{(\Xi, Obs)} (N_{\mathsf{A}}, t, T_{\mathsf{A}})^{3}$$

•

In the rest of this work we will usually consider only total correctness (live) semantics and hence *Obs* 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 $=_{(\Xi, Tr^c)}$.

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, [18]) do just this.

Definition 3 *The relational semantics of an entity* A*:*

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

•

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 LTS A and C we have:

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

It should also be noted that this relational semantics is not the trivial named partial relation-based semantics which stands in isomorphism to the LTS presentation of an entity. *That* semantics (given in the Appendix in Definition 24) does include (named) relations, but is not itself a (single) relation and is not the whole semantic story for the entity. It is essentially just a way of viewing the labelled transitions of an LTS as transition relations. In particular, given such a semantics we cannot say what it represents or means (just, of course, as we cannot when given just an LTS); we need to know the contexts and observations possible for the entity (whether given as an LTS or via transition relations) in order to know what it means.

The relation in Definition 3 does have exactly this information, of course, so it *is* as usual the complete semantics of the entity.

[3]It is common in the process literature to refer to an LTS as a process and leave the equivalence relation = to be inferred from the surrounding text. Thus for some LTS S, use of S is an abbreviation for the equivalence class $\{R \mid S = R\}$. That $[\mathsf{E}]_{\mathsf{X}}$ defined on LTS can also be lifted to entities and contexts requires that = is congruent w.r.t. $[\_]_{\mathsf{X}}$, which it always will be for the models we consider.

3 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 nontermination alone. (We still have our first cup of coffee!)

From Fig. 2 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. In Fig. 3 we illustrate these four cases.

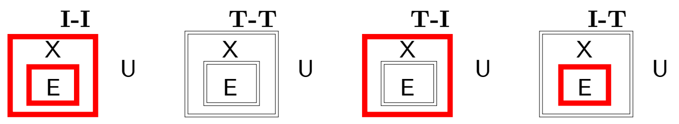


Fig. 3. Interactive interfaces ▬ and transactional interfaces ═ .

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 (details to follow).

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 events that our contexts Ξ may perform that do not synchronise with any event of the entity E. Using these events we can easily construct contexts $\widehat{\mathsf{X}}$ that after synchronising with E perform a distinct special observable event $\widehat{\mathsf{a}}$ that announces to U the fact that the a event 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 event 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.

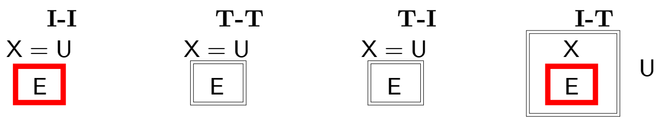


Fig. 4. With a sufficiently large set of contexts Ξ.

Later we will give more concrete examples of all four cases in Fig. 4, and in the **I-I**, **T-T** and **T-I** cases (left-hand three cases of Fig. 4) 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. 4) that requires both interfaces and this case occurs only in Section 6.2. 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 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

[4]In the relational semantics of [16, 19] 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.

if desired) which takes a closer look at what is meant by determinism. This crops up in various places in the rest of the chapter 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 LTSs (or named partial relations) standing alone do not tell the whole semantic story. By taking an LTS and considering what contexts it can appear in and what observations can subsequently be made of the LTS in those contexts we "complete" the meaning of the LTS and in particular fix the notion of refinement for it.

5 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 event $\overline{\mathsf{ba}}$—"pushing button a" causing another event ba—"button a is pushed". The definitions of determinism in CSP [20, p217] and CCS [2, 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 $y =_{pe} z$.

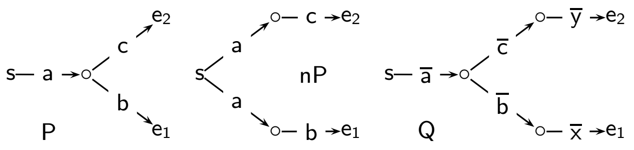


Fig. 5. Is Q deterministic ?

Using this standard definition of determinism P and Q in Fig. 5 are deterministic processes and nP is not. But, as we will discuss later, if we apply this definition to broadcast processes we do not get the results we would expect, or the definition that appears in the literature, so we must find a different definition for our abstract model 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 [2, 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. 5. We assume that $\mathsf{P} \parallel_{\{\mathsf{a,b,c}\}} \_$ is a valid experiment. But if we perform this experiment twice on 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 events, the LTS in question always finishes in the same state then we say it exhibits deterministic behaviour'.

Definition 4 Deterministic behaviour, *$det\_beh_{\Xi}(\mathsf{A})$: for any LTS $\mathsf{A} = (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$ with ρ any sequence of observable events and $\{n, r, t\} \subseteq N_{\mathsf{A}}$[5]:*

$$det\_beh_{\Xi}(\mathsf{A}) \triangleq n \overset{\rho}{\Longrightarrow} r \wedge n \overset{\rho}{\Longrightarrow} t \Rightarrow r =_{\Xi} t$$

•

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

Definition 5 *An LTS A is* **deterministic in** *Ξ, written $det_{\Xi}(\mathsf{A})$, is given by:*

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

•

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 7) 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:

[5] $\Longrightarrow$ is the observational semantics for A, as derived in Definition 27.

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

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 [21] for details and discussion).

We will consider examples of determinism where E is: one, an ADT with interaction via method calling (Section 6.2, Section 6.1); two, a process with interaction via output enabled broadcast (Section 7.1); and three, a state-based operation with interaction via shared memory (Section 8). 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 7.3.

6 Refinement for abstract data types

An ADT is an entity that interacts with programs. A program is a linear, unbranching sequence of events which we can formally represent by an LTS, but for convenience we will refer to such an LTS by the sequence of events 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 [22]. We will model sequential programs with a transactional program-user interface in Section 6.2 and with an interactive program-user interface in Section 6.1. 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 only be achieved 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. 6 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.

6.1 Interactive user-program interface

If the user-program interface is interactive (case **I-I** in Fig. 4) 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:

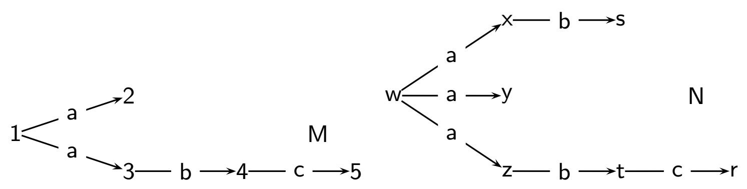


Fig. 6. M and N

Definition 6

$$\Xi_P = \{[\_]_x \mid x \in (\overline{Names})^*\}$$

●

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 1 with contexts Ξ_P and observation function Tr^c, where Tr^c returns a set over $Names^*$. See [15, 11, 4] for examples of such definitions. So, specialising our general refinement in this way we have:

Definition 7

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

●

There are two points to make for future reference:

1. Looking at the example ADT in Fig. 6 with an interactive program-user interface we find M cannot be refined into N as:

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

2. The deterministic contexts of an ADT are the programs $(\overline{Names})^*$ and hence from *DET* (Section 5) 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 automata:

   "no symbol can match the labels of two edges leaving one state" [23] *DFA*

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

6.2 Transactional user-program interface

This we described in case **I-T** (Fig. 4) where the program-user interface is transactional. Many definitions of ADT refinement [8, 5, 7, 17, 16] 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 events 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 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 [8, 5, 17, 16, 7] 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 8

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

$\bullet$

There are two points to be made here also:

1. With a transactional program-user interface we can show that M (Fig. 6) can be refined into N, as we can see by constructing the relevant relations:
$$[\![\mathsf{M}]\!] = \{((\bullet,\bullet),\{(\bullet,\bullet)\}),((\bullet,\bar{\mathsf{a}},\bullet),\{(\bullet,\bullet)\}),\ ((\bullet,\bar{\mathsf{a}},\bar{\mathsf{b}},\bullet),\{(\bullet,\bullet),(\bullet)\}),$$
$$((\bullet,\bar{\mathsf{a}},\bar{\mathsf{b}},\bar{\mathsf{c}},\bullet),\{(\bullet,\bullet),(\bullet)\})\} \cup$$
$$\{(x,\{(\bullet)\}) \mid \forall\, x \notin \{(\bullet,\bullet),(\bullet,\bar{\mathsf{a}},\bullet),(\bullet,\bar{\mathsf{a}},\bar{\mathsf{b}},\bullet),(\bullet,\bar{\mathsf{a}},\bar{\mathsf{b}},\bar{\mathsf{c}},\bullet)\}\}$$
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. 6, where $M \sqsubseteq_{Tp} N$ but not $M \sqsubseteq_{Ip} N$. This was first mentioned in [22].
2. It is quite easy to see that applying Definition 5 to ADTs that can be placed in transactional program–user interfaces picks out the same set of ADTs as being deterministic as applying Definition 5 to ADTs that can be placed in interactional program-user interfaces.

7 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 have a distinct semantics to ADTs because of the change of contexts in which they can be placed [11].

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 5, when applied to ADT is the same as the informal *DFA* characterisation in Section 6.1. But 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 [20, 1, 24, 2] and deterministic broadcast processes as found in [25].

But 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 7.1 we review broadcast processes, and consider what broadcast processes are deterministic, then in Section 7.3 we do the same for handshake processes.

Our interactive branching programs of Section 7.5 are classified as processes because they and their contexts can both branch. These programs/processes can be viewed as a restricted class of handshake processes and 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 $[\_]_{\mathsf{X}}$, X is also an entity. Both entities and contexts are simply processes.

7.1 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 [26]. 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 $\mathsf{a}!$ for $\bar{\mathsf{a}}$ and $\mathsf{a}?$ for a simply to remind the reader how to interpret events that appear in the figures. This is particularly helpful in Section 11 where we need to represent both broadcast and handshake interactions in the same figure.

Even restricting communication to point-to-point there is a variety of different ways to formalise broadcast communication. Some models of broadcast systems [27–30] define parallel composition in such a way that output events cannot be blocked. The alternative approach, found in [31, 3, 32–34] and used here, is to keep parallel composition as in Definition 28 and consider only entities, and thus contexts, that have input actions always enabled.

Definition 9

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

and

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

•

We can now apply Definition 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 10

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

•

7.2 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 $\mathsf{a}?$ is not enabled:

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

It is frequently clearer to not show listening loops (see Fig. 7). Such LTSs can be interpreted as broadcast processes by leaving listening loops implicit.

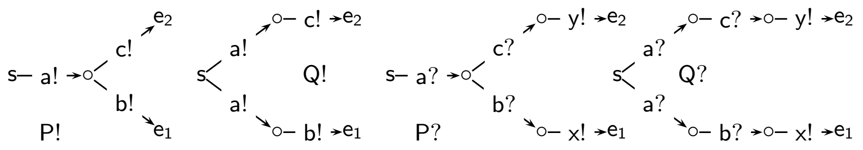


Fig. 7. $M_{BC}(\mathsf{P}!) =_{BC} M_{BC}(\mathsf{Q}!)$ and $M_{BC}(\mathsf{P}?) \neq_{BC} M_{BC}(\mathsf{Q}?)$

The effect of M_{BC} can be best understood by considering some examples. Consider Fig. 7. Processes $M_{BC}(\mathsf{P?})$ and $M_{BC}(\mathsf{Q?})$ are not trace equivalent, e.g. $\mathsf{a?b?c?y!} \notin M_{BC}(\mathsf{P?})$ because if $M_{BC}(\mathsf{P?})$ hears a $\mathsf{b?}$ event after the initial $\mathsf{a?}$ event then it must output $\mathsf{x!}$ not $\mathsf{y!}$ but $\mathsf{a?b?c?y!} \in M_{BC}(\mathsf{Q?})$ as the process, on hearing $\mathsf{a?}$, can make one of two moves, one of which will lead to output $\mathsf{y!}$ being made. This is not the result that might be expected from the handshake perspective where trace semantics are unable to distinguish $\mathsf{P?}$ and $\mathsf{Q?}$.

$\mathsf{P!}$ can broadcast either $\mathsf{b!}$ or $\mathsf{c!}$. As broadcast output is under local control no other process can block either of these events. Hence it seems unavoidable that we consider $\mathsf{P!}$ to be nondeterministic. Yet clearly $\mathsf{P!}$ 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 5) 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 [31, 3], as processes, ignoring listening loops (prior to applying M_{BC}), that branch on only input events with different names (and where $Act^?$ is $\{\mathsf{a}^? \mid \mathsf{a} \in Act\}$):

Definition 11 *The set of deterministic broadcast processes, D_{BC}:*

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

•

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 [31, 3] is consistent with our abstract definition in Definition 5. In CBS all sequential processes are deterministic: "Speakers in parallel are the only source of non-determinism in CBS" [3]. An informal justification for this limitation is that branching outputs of a sequential process could not be implemented.

7.3 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 12

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

and

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

•

Thus, our general refinement (Definition 1) specialises, for these processes, to little more than a rewording of must testing semantics ([35]) and, as has been shown in [11], it is equivalent to failure refinement. We put:

Definition 13

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

•

7.4 Determinism and handshaking

Our definition of deterministic processes, just as for broadcast processes in Section 7.1, 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.

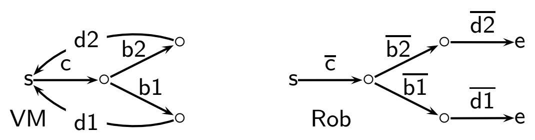


Fig. 8. Are VM and Rob deterministic?

The vending machine VM in Fig. 8 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. 8 requires some thought.

Nondeterminism 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 [1, p81] that: "*There is nothing mysterious about this kind of nondeterminism: 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 nondeterminism as arising from a deliberate decision to ignore time. Alternatively, nondeterminism 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 nondeterministic behaviour when interacting with VM. It is not clear from the literature whether the nondeterminism 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 nondeterminism 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 nondeterminism 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 nondeterminism there. Thus, since the reasons for the nondeterminism 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 nondeterminism (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 7.1 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 implementability 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 nondeterministic[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 implementable* 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 5.

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

7.5 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 [36], that combines process and program ideas.

It is common in the literature on handshake events ([1, 20, 2, 24]) 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. $\bar{\mathsf{a}}$) 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. 9 for an example of how this looks) and after this τ event only the single active event can be executed.

Definition 14

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

where

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

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

•

We put:

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

7.6 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, $\bar{\mathsf{a}}$.

Definition 15 *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. 9) is a nondeterministic specification of the behaviour of Rob in Fig. 8 where the nondeterminism arises from not specifying which button the robot will push.

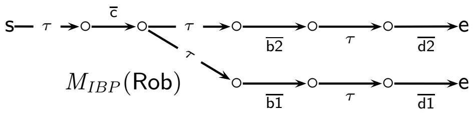


Fig. 9.

We informally define the set of deterministic IBP in the same way as the deterministic broadcast processes in Section 7.1. The deterministic IBP are the processes, prior to applying M_{IBP}, that branch on only passive events with different names.

Definition 16 *The set of deterministic IBP,* D_{IBP}*:*

$$\begin{aligned}D_{IBP} \triangleq \{\mathsf{P}\ \textit{an IBP} \mid\ & q \xrightarrow{\tau}_{\mathsf{P}} n \xrightarrow{\overline{\mathsf{y}}}_{\mathsf{P}} r \wedge q \xrightarrow{\mathsf{z}}_{\mathsf{P}} m \Rightarrow \\ & \quad \exists \overline{\mathsf{x}}, t.m \xrightarrow{\overline{\mathsf{x}}}_{\mathsf{P}} t \wedge (\tau = \mathsf{z} \wedge n = m \wedge \overline{\mathsf{y}} = \overline{\mathsf{x}} \wedge r = t) \\ & \wedge n \xrightarrow{\mathsf{x}}_{\mathsf{P}} m \wedge n \xrightarrow{\mathsf{y}}_{\mathsf{P}} k \Rightarrow \\ & \quad (\mathsf{x} = \mathsf{y} \wedge m = k) \vee (\mathsf{x} \neq \mathsf{y} \wedge \mathsf{x} \in Act \wedge \mathsf{y} \in Act)\}\end{aligned}$$

•

$[\mathsf{Rob}]_{\mathsf{VM}}$ (taking Rob and VM from Fig. 8) and both $[M_{IBP}(\mathsf{Rob})]_{M_{IBP}(\mathsf{VM})}$ and $M_{IBP}(\mathsf{Rob})$ (see Fig. 9) are nondeterministic. 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. 8.

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 5. Thus IBP is a subset of handshaking-style processes in Section 7.3 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.

8 Operation refinement

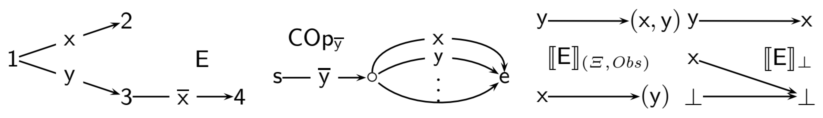
This final section in this sequence of special models is something of an oddity, but is here to (strongly) make the point that an LTS standing alone can have many different meanings, the complete meaning being given by defining the contexts and observations one may make of the LTS (which via our general definition means giving a definition of refinement).

We give a simple LTS, which can be given an obvious meaning (via various completions) as an ADT or a sort of process, as we have seen in the previous sections, but which, to make the point, we will complete in this section to form a *single operation* just by giving the contexts and observations one may make of this LTS when we wish to view it in this way.

Our entity E could be a method, procedure, function etc., its context X an ADT and the user U a program. As the method—ADT interface is transactional (cases **T-T** and **T- I** in Fig. 3) we can view the context and user as the same entity with a single interface between it and the operation.

As we show, with a little rewriting and given the right sort of contexts and observations, we can extract the more usual relational semantics of operations for Definition 3.

Let operation E be represented by the LTS on the left-hand side of Fig. 10.



1
x
2
y
3
$\overline{x}$
4
E
$\mathsf{COp}_{\overline{y}}$
s
$\overline{y}$
○
x
y
⋮
e
y
(x, y)
y
x
$[\![\mathsf{E}]\!]_{(\Xi,Obs)}$
x
$[\![\mathsf{E}]\!]_{\perp}$
x
(y)
⊥
⊥

Fig. 10. Entity E, $\mathsf{COp}_{\overline{y}}$—a context for E, $[\![\mathsf{E}]\!]_{(\Xi,Obs)}$ and $[\![\mathsf{E}]\!]_{\perp}$

To interpret such an LTS as an operation we use as contexts calls to the operation, see $\mathsf{COp}_{\overline{y}}$ in Fig. 10 for one example such a context, one that "uses" y and then waits for that to complete before moving to its ending state e, i.e. from the start state s, actively set the state to some unique value $\circ$ via $\mathsf{s}\xrightarrow{\overline{y}}\circ$ and then wait for the operation to terminate in any state, so for all $\mathsf{z} \in \mathit{Name}$ we have $\circ\xrightarrow{\mathsf{z}}\mathsf{e}$. To complete the semantics, the observation function returns the complete traces, Tr^c.

Definition 17

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

and

$$\mathsf{T}_{\mathsf{Op}} \triangleq \{\mathsf{COp}_{\mathsf{z}} \mid \mathsf{z} \in \overline{\mathit{Names}}\}$$

where

$$\mathsf{COp}_{\mathsf{z}} \triangleq (\{\mathsf{s}, \circ, \mathsf{e}\}, \{\mathsf{s}\}, \{(\mathsf{s}, \mathsf{z}, \circ)\} \cup \{(\circ, \mathsf{w}, \mathsf{e}) \mid \mathsf{w} \in \mathit{Names}\})$$

•

Another way of seeing this is to view E as having events that we view as elements of some set *State*, and it maps the (initial) member of *State*, y, to a (final) member of state *State*, x, and it does not terminate when started in x. In this way of modelling, we can think of our entity E as an operation of an ADT that can store information in a local variable called *State*. E starts in state 1 and when it synchronises with the program on event x it stores x in the *State* variable, moves to state 2 and then refuses to do anything else. If it synchronises with the program on event y it stores y in the *State* variable and moves to state 3. It then returns the value x to the program and terminates in state 4.

We can go further if we employ a little lateral thinking and allow ourselves some freedom in interpretation. The "usual" relational semantics (think of a Z operation, for example), Definition 3, is between the contexts Ξ in which the entity finds itself, which for operations can be summarised as some "starting" value in *State* (which is in turn its first event when viewed as an LTS), and what can be *observed* when the entity is executed in this context is a trace consisting of no more than two observations, the initial and final values held in *State*.

$$[\![\mathsf{E}]\!]_{\Xi_{\mathsf{Op}}} \triangleq \{(x, o) \mid [\_]_x \in \Xi_{\mathsf{Op}}, o \in Tr^c([\mathsf{E}]_x)\} = \{(\mathsf{COp}_{\overline{\mathsf{y}}}, (\mathsf{y}, \mathsf{x})), (\mathsf{COp}_{\overline{\mathsf{x}}}, (\mathsf{x}))\}$$

All that can be observed is the trace of observations, which is a pre- state/post-state pair when E terminates and just the pre-state when E does not terminate. As the pre-state is known from the domain of the relation (e.g. in the first pair in the example the $\mathsf{COp}_{\overline{\mathsf{y}}}$ tells us that the pre-state is y) it need, without loss of generality, not appear in the range, hence:

$$[\![\mathsf{E}]\!] \subseteq \mathit{State} \times (\mathit{State} \cup \{()\}) \textit{ can be recorded just as } \{(\mathsf{y}, \mathsf{x}), (\mathsf{x}, ())\}$$

By convention () is represented by $\perp$ and $\mathit{State} \cup \{()\}$ is written $\mathit{State}_\perp$. Again by convention $\perp$ is added to the domain of the relation and defined in various ways in the literature. To use $\perp$ in the domain to represent not-starting we choose to add only $(\perp, \perp)$.

Thus from the event-based LTS model we have constructed:

$$[\![\mathsf{E}]\!]_\perp \subseteq \mathit{State}_\perp \times \mathit{State}_\perp \textit{ which can be recorded just as } \{(\mathsf{y}, \mathsf{x}), (\mathsf{x}, \perp), (\perp, \perp)\}$$

one of the familiar state-based, pre-post relational semantics of a single operation E as found in the literature.

It is worth pointing out that the use of $\perp$ is frequently criticised because nontermination cannot be observed but if, as we do, you are content to allow the observation of any experiment to be a complete trace then each pair in the relation $[\![\mathsf{E}]\!]_\perp$ is there as a direct result of an observation. If on the other hand you reject the observation of complete traces then it is hard to see how liveness properties could be observed.

Our definition of refinement becomes subset of the relational semantics (recall Section 2):

$$\mathsf{A} \sqsubseteq \mathsf{C} \Leftrightarrow [\![\mathsf{C}]\!]_\perp \subseteq [\![\mathsf{A}]\!]_\perp$$

A context is deterministic when it provides a unique start state. From *DET* (Section 5) an operation E is deterministic if and only if the relation $[\![\mathsf{E}]\!]$ is a function. This is exactly as we would expect from the state-based literature. We do not claim our event-based view of operations to be radically new: it just gives us some confidence in our general model. It is little more that a restructuring of the familiar state-based view. It is interesting, however, that $\perp$ has appeared directly from what is observable of an operation.

9 Developing the General Model of Refinement—adding layers

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

So, each special model is viewed as a layer which has a distinct set of events that can be used to define the entities in the layer. A layer is formalised by a set of entities (and here we use a set of LTS used to represent the entities, but recall we could equally well chose to use a relational semantics) 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 LTS.

Definition 18 *A layer* L *is* $(\mathsf{E_L}, \sqsubseteq_\mathsf{L})$ *where* $\mathsf{E_L}$ *is a set of entities and* $\sqsubseteq_\mathsf{L} \subseteq \mathsf{E_L} \times \mathsf{E_L}$ *is a refinement relation* •

By considering only layers where the refinement relation is defined by Definition 1, *i.e.* $\sqsubseteq_\mathsf{L} \triangleq \sqsubseteq_{\Xi_\mathsf{L}}$, our layers can equally well be defined by the pair $(\mathsf{E_L}, \Xi_\mathsf{L})$ (with *Obs* being assumed to be Tr^c as usual).

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 $\mathsf{A_L} =_\mathsf{L} \mathsf{B_L} \Rightarrow Tr^c(\mathsf{A_L}) = Tr^c(\mathsf{B_L})$; and two, lift placing in a context from LTS to entities, i.e. $\mathsf{A_L} =_\mathsf{L} \mathsf{B_L} \Rightarrow \forall [\_]_x \in \Xi_\mathsf{L}.[\mathsf{A_L}]_x =_\mathsf{L} [\mathsf{B_L}]_x$, as is the case for all the models we consider.

Before going further with talking about layers, we introduce a further sort of refinement that can be used within a layer, *visibility refinement.*

The alphabet of an entity A, written $\alpha(\mathsf{A})$, is the set of events that it can engage in. Visibility refinement refers to extending the alphabet by making visible, and therefore usable, events not considered, because not visible, in the original abstract entity. We reverse the τ-abstraction and δ-abstraction of Definition 30 by extending refinement to introduce events in two quite separate ways [11, 37].

Firstly if δ–refinement holds, $\mathsf{A} \sqsubseteq_{\Xi\delta Del} \mathsf{C}$ (where $\alpha(\mathsf{A}) \cap Del = \varnothing$), 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 19 *δ–refinement and τ–refinement. For LTS A and C :*

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

The sets of all τ-refinements and δ-refinements are preorders:

$\sqsubseteq_{\Xi\tau} \triangleq \bigcup_{Hid} \sqsubseteq_{\Xi\tau Hid}$ $\sqsubseteq_{\Xi\delta} \triangleq \bigcup_{Del} \sqsubseteq_{\Xi\delta Del}$ $\sqsubseteq_{\Xi\tau\delta} \triangleq \sqsubseteq_{\Xi\tau} \cup \sqsubseteq_{\Xi\delta}$ •

See [37] for further details of δ–refinement and τ–refinement.

Here visibility refinement and ordinary ("simple") refinement will be grouped together and called *horizontal refinement* and we will consider them as occurring in one layer or level of abstraction, whereas vertical refinement will be refinement between two distinct layers. It may be of conceptual help to think of a protocol stack. Each layer possesses a *distinct* set of operations but nonetheless each layer implements the layer above it.

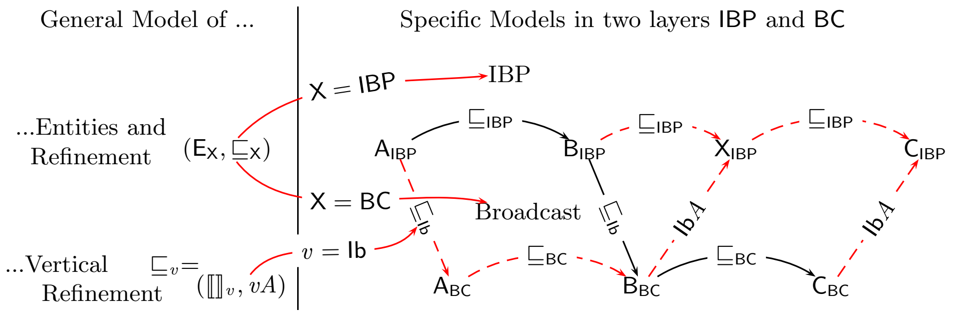
We make our general refinement more flexible in Section 10 by giving a general definition of *vertical refinement* between an abstract and a concrete layer. Our definition of vertical refinement can be seen as a generalisation of non-atomic refinement [38] or action refinement [39, 40] when we consider the LTS used to represent entities. But, when we consider predicates used to define the $\Xi \times Obs$ relations then vertical refinement is a theory morphism similar to those used in UTP [18, 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.

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 11 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.

Fig. 11 shows how our "single layer" general theory of Section 1 and Fig. 1 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.)



General Model of ...
Specific Models in two layers IBP and BC
...Entities and Refinement
$(\mathsf{E}_\mathsf{X}, \sqsubseteq_\mathsf{X})$
$\mathsf{X} = \mathsf{IBP}$
IBP
$\mathsf{X} = \mathsf{BC}$
Broadcast
...Vertical Refinement
$\sqsubseteq_v = ([\![]\!]_v, vA)$
$v = \mathsf{Ib}$
A_IBP
B_IBP
X_IBP
C_IBP
$\sqsubseteq_\mathsf{IBP}$
$\sqsubseteq_\mathsf{Ib}$
$\mathsf{Ib}A$
A_BC
B_BC
C_BC
$\sqsubseteq_\mathsf{BC}$

Fig. 11. Big picture

10 General vertical refinement

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 applying Definition 1 to define refinement $\sqsubseteq_{\Xi_\mathsf{L}}$ of the entities. Thus at this early step in the development a layer $(\mathsf{E}_\mathsf{L}, \sqsubseteq_{\Xi_\mathsf{L}})$ (or equivalently $(\mathsf{E}_\mathsf{L}, \Xi_\mathsf{L})$) has been fixed and an entity chosen to represent the specification. This specification is then developed using the defined refinement.

It is not at all unusual to want to build a more detailed description of the same system but based on a different set of low-level events that may interact in a different style to the original high-level events, thus creating a more abstract, high-level, layer and a more concrete, low-level, layer. Vertical refinement is one way to formally relate descriptions that appear in two distinct layers at two distinct levels of abstraction.

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* [38] or *action refinement* [39, 40].

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.”[40, section 7].

It is well-known [40] 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.

What we define is not intended to cover all situations of interest, but we highlight a situation where our approach might be of use in top-down design.

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}. First model the features needing the high-level events. Then 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.

Finally, we might use τ-refinement or δ-refinement, i.e. visibility refinement, to add additional features needing low-level events in α_{L}.

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.

$$\begin{tikzcd}
\mathsf{P_H} \arrow[rr, "\sqsubseteq_{\mathsf{H}}"] \arrow[dr, dotted, "\sqsubseteq_v"] & & \mathsf{Q_H} \arrow[rr, "\sqsubseteq_{\mathsf{H}}"] \arrow[dr, "\sqsubseteq_v"] & & vA[\![\mathsf{Q_H}]\!]_v \arrow[rr, "\sqsubseteq_{\mathsf{H}}"] & & vA(\mathsf{R_L}) \\
& [\![\mathsf{P_H}]\!]_v \arrow[rr, dotted, "\sqsubseteq_{\mathsf{L}}"] & & [\![\mathsf{Q_H}]\!]_v \arrow[ur, "vA"] \arrow[rr, "\sqsubseteq_{\mathsf{L}}"] & & \mathsf{R_L} \arrow[ur, "vA"] &
\end{tikzcd}$$

Fig. 12. Refinement $\longrightarrow$ 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. 12, 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. 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 to each layer.

The "mixing interaction type" situations we have described use low-level entities and contexts in the range of $[\![\_]\!]_v$. Hence $\sqsubseteq_{[\![\Xi_{\mathsf{H}}]\!]_v}$ is an appropriate refinement. Here we take $\sqsubseteq_{\mathsf{L}}$ to be $\sqsubseteq_{[\![\Xi_{\mathsf{H}}]\!]_v}$.
We will call a pair of semantic mappings $[\![\_]\!]_v$ and vA a vertical refinement if both:

1. low-level refinement can be interpreted as high-level refinement (see Fig. 12) $$[\![\mathsf{P_H}]\!]_v \sqsubseteq_{\mathsf{L}} \mathsf{R_L} \Rightarrow \mathsf{P_H} \sqsubseteq_{\mathsf{H}} vA(\mathsf{R_L})$$
2. high-level refinement can be interpreted as low-level refinement $$[\![\mathsf{P_H}]\!]_v \sqsubseteq_{\mathsf{L}} \mathsf{R_L} \Leftarrow \mathsf{P_H} \sqsubseteq_{\mathsf{H}} vA(\mathsf{R_L}).$$

Definition 20 *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:*

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

•

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

In Section 11, we apply our general definition of vertical refinement and 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 11.1, as we can "implement" only the deterministic IBPs.

10.1 Vertical refinement between LTS

We will define $[\![\_]\!]_v$ and vA to be mappings from LTS to LTS and will apply these mappings to sets of events (and again we use the event label on a transition to denote the LTS consisting of just that labelled transition). Recall that Act_{L} and Act_{H} etc. are the events that entities in layers L and H, respectively, can engage in:

Definition 21

$$Act_{\mathsf{L}} \triangleq [\![Act_{\mathsf{H}}]\!]_v^{\mathsf{HL}} \triangleq \{\mathsf{a} \mid \exists \mathsf{x} \in Act_{\mathsf{H}}, \mathsf{a} \in \alpha([\![\mathsf{x}]\!]_v^{\mathsf{HL}})\}$$
$$vA^{\mathsf{HL}}(Act_{\mathsf{L}}) \triangleq \{\mathsf{a} \mid \exists \mathsf{x} \in Act_{\mathsf{L}}, \mathsf{a} \in \alpha(vA^{\mathsf{HL}}(\mathsf{x}))\}$$

and similarly for $\overline{Act}$ •

Consequently it is easy to see that $Act_{\mathsf{H}} = vA^{\mathsf{HL}}(Act_{\mathsf{L}})$.

The entities in any layer are represented by equivalence classes of LTS not just a single LTS. If the equalities $=_{\mathsf{H}}$ and $=_{\mathsf{L}}$ are *congruent* w.r.t. the relevant semantic mappings $[\![\_]\!]_v$ and vA then we are able to lift the semantic mapping from the LTS to entities (equivalence classes of LTS).

Monotonicity with respect to the preorders defining an adjunction is a well-known property of adjunctions [41, p151]. Thus as our mappings from LTS to LTS are an adjoint (Definition 20). They are monotonic and the monotonicity is sufficient to lift refinement defined on LTS to refinement defined on entities, i.e. equivalence classes of LTS.

11 $(\mathsf{T}_{\mathsf{IBP}}, \sqsubseteq_{\mathsf{IBP}}) \sqsubseteq_v (\mathsf{T}_{\mathsf{BC}}, \sqsubseteq_{\mathsf{BC}})$

In this section we will define a particular vertical refinement between high-level IBP entities from Section 7.5 and low-level broadcast processes of Section 7.1. We will then show that we have been unable to extend the high-level to all handshake process of Section 7.3. 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. 13) into three parts. The try event tb! is performed, subsequently either aborting (rb?) if the context cannot interact on b, or succeeding (ab?) if the context can interact on b. The mapping for the passive event b can be seen in right-hand side of Fig. 13.

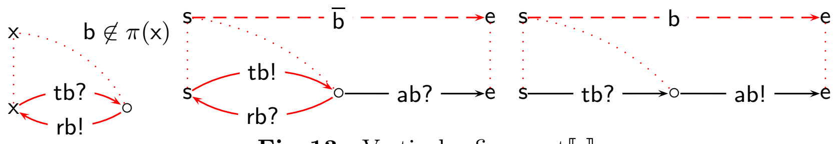


Fig. 13. Vertical refinement$[\![_]\!]_B$

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 (see left-hand side of Fig. 13).

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 [39].

We need some care in interpreting the events of Fig. 13. 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 22 *Let* A *be an LTS* $(N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$.

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

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

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

•

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. 13.

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

Definition 23 *Let* A *be an LTS* $(N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$.

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

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

•

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

11.1 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 the following example[7]:

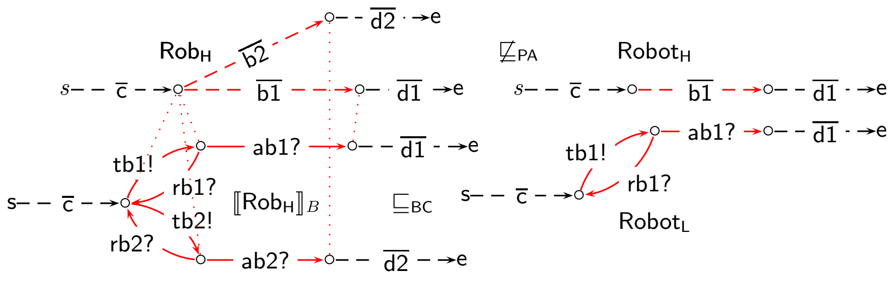


Fig. 14. $[\![\mathsf{Rob_H}]\!]_B \sqsubseteq_{\mathsf{BC}} \mathsf{Robot_L}$ but $\mathsf{Rob_H} \not\sqsubseteq_{\mathsf{PA}} \mathsf{Robot_H}$ and $M_{IBP}(\mathsf{Rob_H}) \sqsubseteq_{\mathsf{IBP}} \mathsf{Robot_H}$

$[\![\mathsf{Rob_H}]\!]_B$ in Fig. 14 is a nondeterministic broadcast process. In particular which button, b1 or b2, it tries to push first is not determined. Hence when offered both buttons by VM its behaviour is nondeterministic. Process Robot$_\mathsf{L}$ is a refinement of $[\![\mathsf{Rob_H}]\!]_B$ that will try button b1 only.

However, we can work harder to gain success with VM. The vending machine VM in Fig. 8 is defined with handshake interactions. This can be vertically refined into an entity with broadcast interactions, VMv in Fig. 15.

We can continue our example with VMv above to show the flexibility of our refinement. For the vending machine VM in Fig. 8 we wish to add an error event,

[7]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. 13.

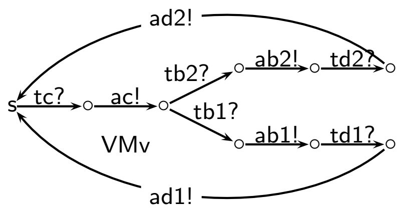


Fig. 15. (Fig. 8) VM $\sqsubseteq_v$ VMv

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!.

By applying visibility refinement we can uncover new broadcast events. The cr! event is an event that was never considered in the original specification (it was never seen nor performed) consequently it must be made visible by δ-refinement to give VMvd in Fig. 15.

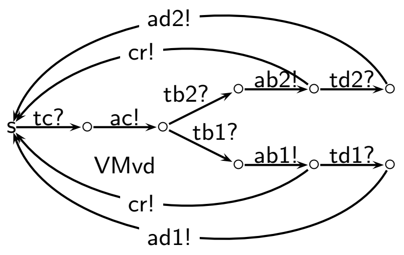


Fig. 16. (Fig. 15)VMv $\sqsubseteq_{BC\delta\{\mathsf{cr}\}}$ VMvd

A more compact way to view this process is VMb in Fig. 17 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. 16 and leave the reader to expand the dashed lines in Fig. 13.

Having made visible the return of coin event we now have an entity that is nondeterministic, 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. 18.

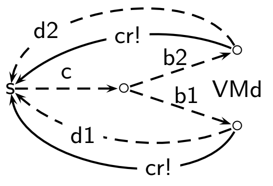


Fig. 17. $\mathsf{VM} \sqsubseteq_{BC\delta\{\mathsf{cr}\}} \mathsf{VMb}$

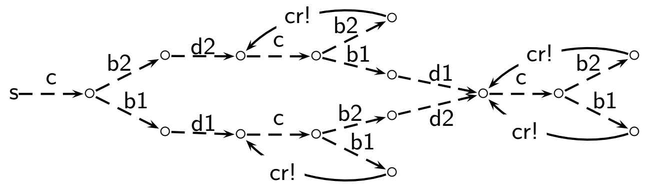


Fig. 18. $\mathsf{VMvd} \sqsubseteq_{BC} \mathsf{VM2}$

12 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. Our definition of general refinement (refinement within a layer, Definition 1 from [11]) is parameterised on the set of contexts and the observations that can be made.

We make use of contexts in distinct ways:

1. Since general refinement has contexts Ξ as a parameter (Definition 1), by changing Ξ we are able to model different types of interaction [11];
2. We distinguish 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 introduce two types of visibility refinement that make refinement more flexible by allowing new events to be added.

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 10) 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". To do this we have been forced (at some length, and more than once) to consider what it means for a handshake entity to be deterministic.

Finally we give a simple example using both vertical refinement and visibility refinement to specify an entity with events with different styles of interaction.

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 automata 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 24 *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 [1, 20], CCS [2], ACP [24], for broadcast systems IOA [33], CBS [3], for abstract data types [15] and for objects [16].

Definition 25 *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 26

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

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 26 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 27 *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

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

•

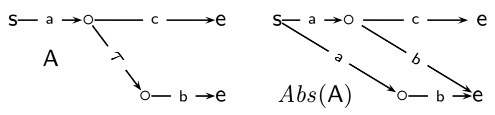


Fig. 19. Event abstraction

Our observational semantics is not the same as in CCS [2] but, like CSP, has the advantage that it "succeeds in total concealment of internal events" [18, p.198]. Our definition comes from [42, 43], 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" [20, p.178]. For a more detailed comparison and discussion of abstracting τ loops see [44].

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

Definition 28 *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*$^\tau$*:*

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

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

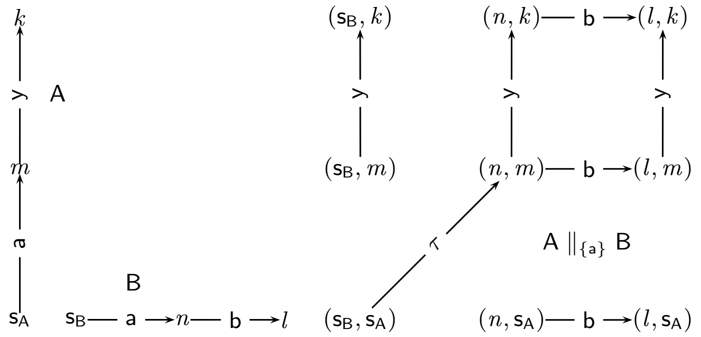


Fig. 20. Example $\mathsf{B} \|_{\{\mathsf{a}\}} \mathsf{A}$

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. 20.

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

Definition 29 *Let ρ be a sequence of events. $\mid \rho \mid$ 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:

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

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

•

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 30 *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:

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

•

References

1. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science (1985)
2. Milner, R.: Communication and Concurrency. Prentice-Hall International (1989)
3. Prasad, K.V.S.: A calculus of broadcasting systems. Science of Computer Programing **25** (1995) 285–327
4. Bolton, C., Davies, J.: A singleton failures semantics for Communicating Sequential Processes. Formal Aspects of Computing **18** (2006) 181–210
5. Woodcock, J., Davies, J.: Using Z: Specification, Refinement and Proof. Prentice Hall (1996)
6. Spivey, J.M.: The Z notation: A reference manual. 2nd. edn. Prentice Hall (1992)
7. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
8. de Roever, W.P., Engelhardt, K.: Data Refinement: Model oriented proof methods and their comparison. Cambridge Tracts in theoretical computer science 47 (1998)
9. van Glabbeek, R.J.: Linear Time-Branching Time Spectrum I. In: CONCUR '90 Theories of Concurrency: Unification and Extension. LNCS 458, Springer-Verlag (1990) 278–297
10. van Glabbeek, R.J.: The Linear Time - Branching Time Spectrum II. In: International Conference on Concurrency Theory. (1993) 66–81
11. Reeves, S., Streader, D.: Comparison of Data and Process Refinement. In Dong, J.S., Woodcock, J.C.P., eds.: ICFEM 2003. LNCS 2885. Springer-Verlag (2003) 266–285
12. Fischer, C., Wehrheim, H.: Behavioural subtyping relations for object-oriented formalisms. LNCS **1816** (2000) 469–483
13. Abrial, J.R., Cansell, D., Méry, D.: Refinement and reachability in Event B. In Treharne, H., King, S., Henson, M.C., Schneider, S., eds.: ZB05: Formal Specification and Development in Z and B. Volume 3455 of Lecture Notes in Computer Science., Springer (2005) 222–241
14. Brinksma, E., Scollo, G.: Formal notions of implementation and conformance in LOTOS. Technical Report INF-86-13, Twente University of Technology, Department of Informatics, Enschede, The Netherlands (1986)
15. Bolton, C., Davies, J.: A singleton failures semantics for Communicating Sequential Processes. Research Report PRG-RR-01-11, Oxford University Computing Laboratory (2001)
16. Derrick, J., Boiten, E.: Relational concurrent refinement. Formal Aspects of Computing **15** (2003) 182–214
17. Derrick, J., Boiten, E.: Refinement in Z and Object-Z: Foundations and Advanced Applications. Formal Approaches to Computing and Information Technology. Springer (2001)
18. Hoare, C., Jifeng, H.: Unifying Theories of Programming. Prentice Hall International Series in Computer Science (1998)
19. Dunne, S., Conroy, S.: Process refinement in B. In: ZB 2005: Formal Specification and Development in Z and B. Volume 3455 of LNCS., Springer (2005) 45–64
20. Roscoe, A.: The Theory and Practice of Concurrency. Prentice Hall International Series in Computer Science (1997)
21. Hehner, E.C.R., Gravell, A.M.: Refinement semantics and loop rules. In: World Congress on Formal Methods (2). (1999) 1497–1510

22. Reeves, S., Streader, D.: State- and Event-based refinement. Technical report, University of Waikato (2006) Computer Science Working Paper Series 09/2006, ISSN 1170-487X, http://www.cs.waikato.ac.nz/∼dstr.
23. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
24. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge Tracts in Theoretical Computer Science 18 (1990)
25. Lynch, N., Vaandrager, F.: Forward and backward simulations, part i: Untimed systems. Information and Computation 121(2) (1995) 214–233
26. Ene, C., Muntean, T.: Expressiveness of Point-to-Point versus Broadcast Communications. Volume LNCS 1684., Springer-Verlag (1999) FCT'99.
27. Tretmans, D.: A Formal Approach to Conformance Testing. PhD thesis, Faculteit der Informatica (1992)
28. Prasad, K.V.S.: A calculus of value broadcasts. In: Parallel Architectures and Languages Europe. (1993) 391–402
29. Kumar, R., Heymann, M.: Masked prioritized synchronization for interaction and control of discrete event systems. IEEE Transactions on Automatic Control **45** (2000) 1970–1982
30. Ene, C., Muntean, T.: Testing Theories for Broadcasting Processes. (2004) Submitted for publication, http://www.esil.univ-mrs.fr/∼cene.
31. Vaandrager, F.W.: On the relationship between process algebra and input/output automata. In: Logic in Computer Science. (1991) 387–398
32. Segala, R.: A Process Algebraic View of I/O Automata. Technical Report MIT/LCS/TR-557, Massachusetts Institute of Technology (1992)
33. Lynch, N., Tuttle, M.: An introduction to input/output automata. CWI-Quarterly (1989) 2(3):219–246
34. Lynch, N., Segala, R.: A Comparison of Simulation Techniques and Algebraic Techniques for Verifying Concurrent Systems. Formal Aspects of Computing Journal **7** (1995) 231–265
35. de Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science **34** (84)
36. Reeves, S., Streader, D.: Stepwise Refinement of Processes. Technical report, University of Waikato (2005) Computer Science Technical Report 07/2005 , http://www.cs.waikato.ac.nz/∼dstr.
37. Reeves, S., Streader, D.: Liberalising event b without changing it. Technical report, University of Waikato (2006) Computer Science Working Paper Series 07/2006, ISSN 1170-487X, http://www.cs.waikato.ac.nz/∼dstr.
38. Derrick, J., Boiten, E.A.: Non-atomic refinement in z. In: FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II, Springer-Verlag (1999) 1477–1496
39. Rensink, A., Gorrieri, R.: Vertical implementation. Information and Computation **170** (2001) 95–133
40. Gorrieri, R., Rensink, A.: Action refinement (2001)
41. Taylor, P.: Practical Foundations of Mathematics. Cambridge University Press (1999) Cambridge studies in advanced mathematics 59.
42. Brinksma, E., Rensink, A., Vogler, W.: Applications of fair testing. FORTE **69** (1996) IFIP Conference Proceedings.
43. Valmari, A., Tienari, M.: Compositional Failure-based Semantics Models for Basic LOTOS. Formal Aspects of Computing **7** (1995) 440–468
44. Reeves, S., Streader, D.: Atomic Components. In: ICTAC 2004. LNCS 3407. Springer-Verlag (2004) 128–139