Journal of Universal Computer Science, vol. 11, no. 12 (2005), 2034-2045
submitted: 29/9/05, accepted: 4/10/05, appeared: 28/12/05 © J.UCS

Constructing Programs or Processes[1]

Steve Reeves
(Department of Computer Science, University of Waikato, Hamilton, New Zealand
Email: stever@cs.waikato.ac.nz)

David Streader
(Department of Computer Science, University of Waikato, Hamilton, New Zealand
Email: dstr@cs.waikato.ac.nz)

Abstract: We define *interacting sequential programs*, motivated originally by constructivist considerations. We use them to investigate notions of implementation and determinism. Process algebras do not define what can be implemented and what cannot. As we demonstrate it is problematic to do so on the set of all processes. Guided by constructivist notions we have constructed interacting sequential programs which we claim can be readily implemented and are a subset of processes.

Key Words: Process algebra, determinism, cause, refinement, constructive
Category: F.1

1 Introduction

1.1 An overture

Per Martin-Löf [Martin-Löf 1985] takes as a touchstone of constructivity the "proofs as programs" idea (and the "types as specifications" and "propositions as specification" ideas). So, a proof of an implication should be a computable (proof-transforming) function which constructs a proof of the consequent from a proof of the antecedent. Similarly, a proof of a conjunction should be a pair consisting of proofs of the left and right predicates in the conjunction. A proof on an existential predicate should consist of a pair too: the first element is an example of one of the things whose existence is claimed and the second element is a proof that this things has the properties claimed for it. Following this line, predicates can be thought of as specifications.

It turns out that the notion of proof and the notion of program coincide in this sort of theory. So, the decision as to whether something is true comes down to the question as to whether there is a proof for it. In other words, a predicate is true if there is, viewing it as a specification, an implementation of it.

Within Martin-Löf's theories we can also reason about equivalence (or other logical relations) between predicates (via the *universes*). That is, within the theories there is a sub-theory which allows reasoning about specifications. Also, in the higher universes, we can write specification-transforming functions.

[1] C. S. Calude, H. Ishihara (eds.). *Constructivity, Computability, and Logic. A Collection of Papers in Honour of the 60th Birthday of Douglas Bridges.*

So, viewing use of the theories through the "programming glasses" we see a typical development consisting of reasoning at the specification level ($i.e.$ manipulation of predicates) followed, once the specification is in an appropriate form, by a drop into the final program ($i.e.$ the construction of the proof). Thus, the whole development cycle of a program is exactly mirrored by the development cycle of a proof, and we can observe the idea of work going on at different levels which become progressively closer to the program.

In fact, in what follows we shall not be explicitly working directly with the formalities of Martin-Löf's framework, but we will be using the constructive ideas upon which that theory is based and some of the technicalities it uses ($e.g.$ evaluation of terms being "lazy") in order to motivate certain analogies which guide and inform our direction and the choices we make.

We have programs as proofs and specifications as propositions. The programs are executable (they evaluate lazily to some normal form which is what we choose to count as a value, inhabiting the lowest universe) and the specifications, even in normal form, do not count as values (we are here thinking of them as objects in some universe above the lowest one where the "values", like numbers, reside). So, the programs are executable (using rules of equality, to give values) whereas the specifications are not (even though they can be reduced to their normal form, they do not reduce to values in the lowest universe). Further, specifications (propositions) may not implementable either because they contain no programs (have no proofs).

So, we say that programs can be reduced to their normal forms (values) in the lowest universe, which we call execution, and they are always, of course, implementable.

A specification may or may not be implementable (or, read as a proposition, may or may not be true, $i.e.$ provable), which is to say it may or may not contain programs. So, of course, some specifications do not specify things which can be implemented (just as some propositions are not true), which means that they contain no programs. Specifications which can be implemented contain programs (which can be executed), and although a specification can be reduced to its normal form, this will be an object in a universe above the lowest one (where the values reside) so specifications can never be executed though they may be implementable.

A process is analogous to a proposition (specification) rather than a proof in that it may have many traces (implementations) because it is nondeterministic.

Putting two processes in synchronous composition means that under the analogy we have something like application at the program (proof) level also happening at the specification (proposition) level.

In the appropriate universe we can think of a term which takes a pair of specifications and returns another specification, $i.e.$ a function for combining specifications.

It would be strange (and perhaps unacceptable) if this function took two specifications (propositions) which were implementable ("had programs", or as propositions they had proofs) and combined them to form a specification that had no programs, $i.e.$

went from implementable specifications to give a non-implementable one (or went from a pair of true propositions and combined them to give a false one).

By analogy, what is the situation with determinism? In what follows, though it may not always be explicit, we are motivated by, and try to follow, the ideas above, which guide thinking about specifications and programs and implementability by identifying them with propositions and proofs and constructivity. In particular we seek to preserve determinism in the same way as constructivity is preserved within philosophies and theories like Martin-Löf's. This goal motivates and informs what follows.

1.2 Interactive processes

For programs that are concurrent or interactive (as opposed to "transactional", where inputs are consumed and transformed to outputs) the picture is less well-developed. By "an interactive process" we mean a program that, in addition to consuming parameters and returning values, interacts with the world around it via named actions. Formalisms such as CSP, CCS and ACP lack the clear distinction between a specification and an implementation.

Here we wish to formalise what interacting sequential programs can be implemented, on modern computers. By "implemented" we do not mean "what specifications can be refined into".

Given that sequential computers are finite state deterministic machines, what can be implemented on such machines must be related to "what we mean by deterministic". Unfortunately the determinism in state-based formalisms of transactional programs is not the same as determinism in event-based formalisms such as CSP, CCS and ACP. The determinism of such processes will be shown to be *may determinism* in as much as a process is deterministic if there is a context that may determine how it behaves. In contrast to this the determinism of state-based abstract data types (ADT) is *must determinism* in that an ADT is deterministic if and only if any context, $i.e.$ every program, must determine how it behaves.

We have also found that what is meant by sequential is not as obvious as we had initially thought. Clearly a sequential program can only perform one action at a time. But, can a sequential program offer to perform more than one action at a time and allow the implemented context in which it is placed decide what it performs?

Could we implement a program that offered to either pop a value from a stack or to push the value $\mathsf{1}$ onto the same stack, $\overline{\mathsf{pop}} + \overline{\mathsf{push}(1)}$? We will define interacting sequential programs (isp) based on the assumption that programs do not possess, as primitive, the ability to offer their contexts a choice of methods they might call.

As isp terms turn out to be a subset of the deterministic finite state processes of CSP (or CCS or ACP) we will compare isp with process algebra-style processes and argue that the limitation on process algebra has little effect in practice.

2 Semantics of interacting terminating processes

Interacting processes are given an action-based semantics by labelling a state transition with an action. The observable actions a can only be performed when the process is executed in a context that includes a parallel process that is ready to execute $\overline{\mathsf{a}}$ the "other half"' of the action a. In handshake models of processes the execution of actions a and $\overline{\mathsf{a}}$ are not under local (their own) control and are blocked from execution whenever the context they are in is not ready to execute their "other half".

A special action τ is introduced that models an action that cannot be seen. Importantly in process algebras CSP, CCS and ACP an action can be blocked if and only if it is observable [Roscoe 1997, Milner 1989, Baeten and Weijland 1990].

We assume a universe containing a set of passive actions $Act \stackrel{\text{def}}{=} \{\mathsf{a}|\mathsf{a} \in Names\}$ from which we build active actions $\overline{Act} \stackrel{\text{def}}{=} \{\overline{\mathsf{a}}|\mathsf{a} \in Names\}$. We define $Obs \stackrel{\text{def}}{=} Act \cup \overline{Act}$ and $Act^\tau \stackrel{\text{def}}{=} Obs \cup \{\tau\}$.

$S \subseteq Obs$ and $D \subseteq Obs$	
Actions	$act = \delta|\overline{\mathsf{a}}|\mathsf{a}$
Sequential	$pr = Skip|act|pr\delta_D|pr + pr|act;pr$
Parallel	$parp = pr|parp \,\|_S\, parp$

Figure 1: Process terms

The operational semantics of processes are widely defined by labelled transition systems (LTS).

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

We write $x \xrightarrow{\mathsf{a}} y$ for $(x, \mathsf{a}, y) \in T_\mathsf{A}$ where A is obvious from context, $n \xrightarrow{\mathsf{a}}$ for $\exists m.(n, \mathsf{a}, m) \in T_\mathsf{A}$,

Let ρ be a sequence of actions $\rho_1\rho_2\rho_3 \ldots \rho_x$. We write $s_\mathsf{A} \xrightarrow{\rho} y$ iff $(s_\mathsf{A}, \rho_1, n_2)$, $(n_2, \rho_2, n_3), \ldots (n_x, \rho_x, y) \in T_\mathsf{A}$ and $Tr_\mathsf{A} \stackrel{\text{def}}{=} \{\rho|s_\mathsf{A} \xrightarrow{\rho} y\}$.
The complete traces of A are: $Tr^c(\mathsf{A}) \stackrel{\text{def}}{=} \{\rho|(s_\mathsf{A} \xrightarrow{\rho} n \wedge \{\mathsf{a}|n \xrightarrow{\mathsf{a}}\} = \emptyset)\}$.

We write $\mathsf{A} \sqsubseteq_x \mathsf{C}$ for C is a refinement of A using the refinement relation $\sqsubseteq_x$. We interpret the meaning of a specification to be given by the set of implementations that it can be refined into and hence the meaning of a specification is given by a definition of refinement.

Refusals are defined: $Ref(\rho, \mathsf{C}) \stackrel{\text{def}}{=} \{\{\mathsf{a}|n \stackrel{\mathsf{a}}{\not\longrightarrow}\}|s_\mathsf{C} \xrightarrow{\rho} n\}$ and failure refinement [Roscoe 1997]: $\mathsf{A} \sqsubseteq_F \mathsf{C} \stackrel{\text{def}}{=} \forall\rho.Ref(\rho, \mathsf{C}) \subseteq Ref(\rho, \mathsf{A})$.

Singleton refusals are defined: $SRef(\rho, \mathsf{C}) \stackrel{\text{def}}{=} \{\{\mathsf{a}\} | s_{\mathsf{C}} \xrightarrow{\rho} n \wedge n \stackrel{\mathsf{a}}{\not\longrightarrow}\}$ and singleton failure refinement [Bolton and Davies 2001]: $\mathsf{A} \sqsubseteq_{sF} \mathsf{C} \stackrel{\text{def}}{=} \forall \rho . SRef(\rho, \mathsf{C}) \subseteq SRef(\rho, \mathsf{A})$.

When the terms in [Fig. 1] are used to define processes they have the operational semantics defined in [Fig. 2].

LTS		
Action	$\mathsf{a};P \xrightarrow{\mathsf{a}} P$	$\bar{\mathsf{a}};P \xrightarrow{\bar{\mathsf{a}}} P$
Choice	$\dfrac{p_1 \xrightarrow{\alpha} p_2}{p_1 + p_3 \xrightarrow{\alpha} p_2}$	$\dfrac{p_1 \xrightarrow{\alpha} p_2}{p_3 + p_1 \xrightarrow{\alpha} p_2}$
Parallel	$\dfrac{p \xrightarrow{\mathsf{a}} q, p_1 \xrightarrow{\bar{\mathsf{a}}} q_1, \mathsf{a} \in S}{p \parallel_S p_1 \xrightarrow{\tau} q \parallel_S q_1}$	$\dfrac{p \parallel_S p_1 \xrightarrow{\alpha} q}{p_1 \parallel_S p \xrightarrow{\alpha} q}$
	$\dfrac{p \xrightarrow{\mathsf{a}} q, \mathsf{a} \notin S}{p \parallel_S p_1 \xrightarrow{\mathsf{a}} q \parallel_S p_1}$	$\dfrac{p \xrightarrow{\bar{\mathsf{a}}} q, \mathsf{a} \notin S}{p \parallel_S p_1 \xrightarrow{\bar{\mathsf{a}}} q \parallel_S p_1}$

Figure 2: Operational semantics of processes

Our parallel composition with synchronisation operator $\_ \parallel_S \_$ enforces private communication between its operands on all actions in the synchronisation set S. Thus any action in S that appears in one of the operands must either: synchronise with an action from the other operand; or be *blocked*.

[Fig. 2] defines what is called a strong semantics, $i.e.$ a semantics that treats τ actions just like an observable action. In order to model τ actions as unobservable we will define, in [Section 2.2], how to abstract them to produce what is called the observational semantics.

2.1 Deterministic behaviour

Our process terms are defined in [Fig. 1] and $\mathsf{a} + \mathsf{b}$ is a process that allows its context to decide if a or b is to be executed. This is deterministic behaviour.

Definition 2 A *is deterministic iff* $n \xrightarrow{\alpha} \wedge\, n \xrightarrow{\beta} \Rightarrow \alpha \neq \beta$ •

The behaviour of term $\mathsf{a} + \mathsf{a}$ is nondeterministic, as which a action is executed cannot be decided by its context.

The view of Hoare is [Hoare 1985, p81]: "*There is nothing mysterious about this kind of nondeterminism: it arises from a deliberate decision to ignore the factors which influence the selection.*"

Hoare makes it quite clear that nondeterministic sequential processes are not intended to be implemented [Hoare 1985, p82] "*Nondeterminism has been introduced here in its purest and simplest form by the binary operator,* $\sqcap$. *Of course,* $\sqcap$ *is not intended as a useful operator for* implementing *a process.*" [Hoare's emphasis]

Later we will consider if nondeterministic parallel processes are intended to be specifications or not, but next we will define how to build an observational semantics.

2.2 Abstraction

Our definition of observational semantics is quite separate from the definition of strong equality/refinement. This allows us to use the same observational semantics with distinct strong semantics.

Definition 3 *Observational semantics* $\Longrightarrow$*:*

$$\begin{aligned} s \overset{\tau}{\Longrightarrow} t &\overset{\mathrm{def}}{=} s \xrightarrow{\tau} s_1, s_1 \xrightarrow{\tau} s_2, \ldots s_{n-1} \xrightarrow{\tau} t \\ n \overset{\mathsf{a}}{\Longrightarrow} m &\overset{\mathrm{def}}{=} n \overset{\tau}{\Longrightarrow} n', n' \xrightarrow{\mathsf{a}} m', m' \overset{\tau}{\Longrightarrow} m \wedge \mathsf{a} \in Act \\ Abs(\mathsf{A}) &\overset{\mathrm{def}}{=} (N_{\mathsf{A}}, s_{\mathsf{A}}, \{n \xrightarrow{x} m \mid n \overset{x}{\Longrightarrow} m\}). \end{aligned}$$

•

Our observational semantics is not the same as in CCS [Milner 1989] as we, like CSP, use failure semantics and thus $Abs(\_)$ removes all τ actions (see example to the right). As we only consider terminating processes here we do not need to consider τ loops or infinite sequences of τ actions.

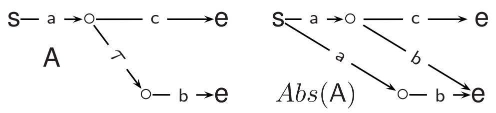

Figure 3: Action abstraction

Definition 3, or an equivalent definition, has appeared in [Brinksma et al. 1996, Valmari and Tienari 1995], and see [Reeves and Streader 2004] for a comparison with the literature.

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

$$\mathsf{A} \sqsubseteq_{aX} \mathsf{C} \overset{\mathrm{def}}{=} Abs(\mathsf{A}) \sqsubseteq_X Abs(\mathsf{C})$$

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

Clearly given our definition of parallel composition with private communication [Fig. 1], parallel composition may introduce τ actions and the observational semantics (Definition 3) we use to model them as unobservable means that nondeterminism can be introduced (see [Fig. 3]).

2.3 Vending machines and robots.

The process algebras such as CSP have been use to define LTS that define the operational semantics of both parallel programs and processes such as vending machines and robots.

Let $\mathsf{VM} \stackrel{\text{def}}{=} \mathsf{c}; (\mathsf{b1};\mathsf{d1} + \mathsf{b2};\mathsf{d2})$ and note it has operational semantics $\mathsf{VM_{pr}}$ in [Fig. 4]. VM is easy to understand as a machine that accepts a coin (c) and then reacts to either button one ($\mathsf{b1}$) or button two ($\mathsf{b2}$) being pushed and subsequently enables the removal of drink one ($\mathsf{d1}$) or drink two ($\mathsf{d2}$). Hopefully it is easy to see that VM is a realistic representation of a real machine that after having a coin inserted offers its context the option of pushing one of two buttons.

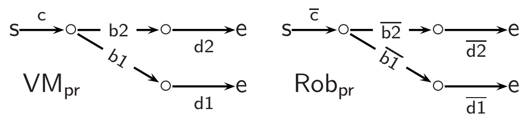

Figure 4: $\mathsf{VM_{pr}}$ and $\mathsf{Rob_{pr}}$

A robot can choose what button is pushed by synchronising with one of the button pushing actions and blocking the other by not synchronising with it. Hence it will choose to push button $\mathsf{b1}$ and block button $\mathsf{b2}$ in $(\overline{\mathsf{c}};\overline{\mathsf{b1}}) \parallel_{\{\mathsf{b1},\mathsf{b2},\mathsf{c},\mathsf{d1},\mathsf{d2}\}} \mathsf{VM}$.

But how realistic is $\mathsf{Rob} \stackrel{\text{def}}{=} \overline{\mathsf{c}}; (\overline{\mathsf{b1}};\overline{\mathsf{d1}} + \overline{\mathsf{b2}};\overline{\mathsf{d2}})$ with operational semantics $\mathsf{Rob_{pr}}$ in [Fig. 4] as either a robot or as a program? The robot Rob is supposed to offer the vending machine the option to choose which button it will push. Is this realistic?

Put another way, could you offer a simple vending machine the ability to choose what buttons you were going to push? If we consider Rob as a program then we cannot view the button pushing as method calling as with current languages programs can only try to call one method at a time. They cannot offer to call either method $\mathsf{b1}$ or to call method $\mathsf{b2}$ and ask the ADT or object with these methods to select which it will call.

If we execute processes $\mathsf{R1} \stackrel{\text{def}}{=} \overline{\mathsf{c}};\overline{\mathsf{b1}}$ and $\mathsf{R2} \stackrel{\text{def}}{=} \overline{\mathsf{b2}}$ in parallel with VM then which button is pushed depends on depends on which robot (R1 or R2) is fastest and consequently we call this form of nondeterminism *race nondeterminism*. From this example we conclude that nondeterminism can arise quite naturally from the execution of easy to implement processes run in parallel.

We are left with two questions:

one Is the nondeterminism of $\mathsf{Rob} \parallel_{\{\mathsf{b1},\mathsf{b2},\mathsf{c},\mathsf{d1},\mathsf{d2}\}} \mathsf{VM}$ a result of the parallel execution of processes that we can implement or, like sequential processes [Section 2.1], is it to be interpreted as a specification that can be satisfied by observationally distinct implementations?

two If the nondeterminism of $\mathsf{Rob} \parallel_{\{\mathsf{b1},\mathsf{b2},\mathsf{c},\mathsf{d1},\mathsf{d2}\}} \mathsf{VM}$ arises from the deliberate decision to ignore certain factors, then what are they?

In order to answer these questions we will define an operational semantics that interprets the terms of [Fig. 1] as interacting sequential programs. These programs are based on the familiar idea of active actions representing the calling of a method and passive actions representing a called method. We claim that these programs would be easy to implement.

As the operational semantics of our interacting sequential programs is a restriction of the operational semantics of processes we can use this restricted set of LTS in the generalised testing semantics of [Reeves and Streader 2003] to build a refinement relation for our interacting sequential programs. But first we will review generalised testing semantics.

3 Generalised testing semantics

It is clear that different equivalences and refinements are appropriate when modelling different kinds of actions and a common way to formalise the situation is by defining a testing semantics and then to use the testing semantics to define equality and refinement: for a survey see [van Glabbeek 2001].

To define refinement of a process P we first combine it with some context $[\_]$ from some set of valid contexts Ξ then observe all possible executions. We are interested in the executions that may occur without any help from additional concurrent processes. Thus we are interested in the possible execution of τ actions, the problem being that handshake formalisms treat τ actions as unobservable. The solution in [de Nicola and Hennessy 1984] was to introduce a special action ω that could be observed but never blocked and could only appear in a testing process.

A key observation of [Bolton and Davies 2001] is that different kinds of things can be placed in differing contexts and that this can be used to define different refinement semantics for different kinds of things. This is made explicit in [Reeves and Streader 2003] where a generalised refinement relation, parameterised on the set of legal contexts, is defined. But unfortunately, as pointed out in [Reeves and Streader 2003], a testing semantics that uses just one special action ω would not give the singleton refinement of [Bolton and Davies 2001].

We will assume the existence of a set of actions $\mathsf{a}^! \in Obs^!$ in our testing process $LTS^!$ that do not synchronise with any action in the processes to be tested, *i.e.* $Act^! \cap S = \emptyset$ in Definition 4. Using these actions we can easily construct contexts that after synchronising with the process under test always perform a distinct special observable action $\mathsf{a}^! \notin S$ that announces the fact that the a action has been performed.

If $n \xrightarrow{\mathsf{a}}_{\mathsf{A}} m$ then $n \xrightarrow{\mathsf{a}}_{\mathsf{A}^!} z \xrightarrow{\mathsf{a}!}_{\mathsf{A}^!} m$ where z is a not a node in A.

Such testing processes have the effect of making visible any action that the testing process synchronises with. Here we use a simplified version of the generalised refinement found in [Reeves and Streader 2003]:

Definition 4 *Let* $\Xi \subseteq \{(\_ \parallel_S \mathsf{x}) | \mathsf{x}$ *is an* LTS *extended as above with ! actions*$\}$, *then*

$$\mathsf{A} \sqsubseteq_{\Xi} \mathsf{C} \stackrel{\text{def}}{=} \forall_{[\_]_{\mathsf{x}} \in \Xi}. Tr^c([\mathsf{C}]_{\mathsf{x}}) \subseteq Tr^c([\mathsf{A}]_{\mathsf{x}}) \qquad \bullet$$

This general definition of refinement can be made more concrete by fixing the contexts in which the things are to be placed. In [Reeves and Streader 2003] it is shown that the generalised refinement with *all* LTS is equivalent to failure refinement [Hoare 1985] and with only programs (traces of events) legal, is equivalent to singleton failure refinement [Bolton and Davies 2001].

4 Interacting sequential programs

We will define an interacting sequential program *isp* to have the following informal properties:

1. *There is no effect without a cause;*
2. *Sequential programs can only try to call one method at a time.*

We model a method of an ADT or vending machine's actions as being passive actions. We model the actions of a program, the calling of a method or of a robot, as active actions. We view the active actions as causing the performance of the passive actions. Thus one action of any synchronising pair of actions is an active action that causes the other passive action to occur. The active actions are written with the name overlined ($e.g.$ $\overline{\mathsf{a}}$) and the passive actions with no overline ($e.g.$ a).

As the active actions of our programs are the calling of a method, if a program tries to call a method that cannot be called then the program terminates. That is to say calling a method is *committing*: once started the caller cannot back off. In order to formalise this we change the operational semantics of active actions $\overline{\mathsf{a}}$: the only change we make to what is otherwise a standard process semantics (see [Fig. 5]).

LTS_{sp}	
Actions	$\mathsf{a};P \xrightarrow{\mathsf{a}} P$ $\overline{\mathsf{a}};P \xrightarrow{\tau} \hat{\mathsf{a}};P \xrightarrow{\overline{\mathsf{a}}} P$
For $\_ + \_$, $\_;\_$ and $\_ \parallel_S \_$ see [Fig. 2]	

Figure 5: Operational semantics of isp

Our model is a generalisation of the programs and ADT of [Bolton and Davies 2001] in that there they restrict processes to be programs that are only a trace of active actions and ADT that only have passive actions. Here our programs can have a mixture of active and passive actions.

The isp process $\mathsf{Rob} \stackrel{\mathrm{def}}{=} \overline{\mathsf{c}};(\overline{\mathsf{b1}};\overline{\mathsf{d1}} + \overline{\mathsf{b2}};\overline{\mathsf{d2}})$ is a nondeterministic specification where the nondeterminism arises from not specifying which button it will push (method it will call) (see [Fig. 6]).

Nondeterminism can arise naturally with concurrent deterministic processes, for example running processes R1 $\stackrel{\text{def}}{=}$ $\overline{\mathsf{c}};\overline{\mathsf{b1}}$ and R2 $\stackrel{\text{def}}{=}$ $\overline{\mathsf{b2}}$ in parallel with VM. The reason we gave for this nondeterminism was that the two processes R1 and R2 race to push different buttons and which button is pushed is not determined. This implies that R1 $\|_{\{\}}$ R2 should be nondeterministic, which it is in our model, $i.e.$ using [Fig. 5] semantics. But if we use the process semantics in [Fig. 2] or CSP semantics then R1 $\|_{\{\}}$ R2 is a deterministic process.

Using isp semantics [Fig. 5] the behaviour of both Rob $\|_{\{\mathsf{b1,b2,c,d1,d2}\}}$ VM and Rob see [Fig. 6], are nondeterministic. This is not because distinct sequential processes are racing to perform active actions but because Rob is failing to choose what active action it will perform. What is more Rob can now be refined into a deterministic isp whereas it could not using the process semantics of [Fig. 2].

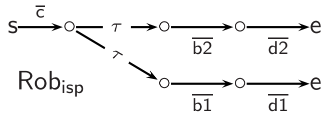

Figure 6: $\mathsf{Rob}_{\mathsf{isp}}$

We can use isp terms to define a set of contexts and then apply the generalised definition in [Section 3] to define refinement. Let A and C be terms from [Fig. 1]. Then it is clear that refinement built from process semantics in [Fig. 2] will be failure refinement, from [Hennessy 1988, Reeves and Streader 2003] .

Lemma 1 *The testing refinement built from isp semantics in [Fig. 5] is singleton failure refinement:* $\mathsf{A} \sqsubseteq_{sF} \mathsf{C} \iff \mathsf{A} \sqsubseteq_{\Xi_{isp}} \mathsf{C}$.

Proof Sketch.

Let us write Ξ_{sF} for the tests that are programs. It is known that these tests generate singelton failure refinment [Bolton and Davies 2001]. As $\Xi_{sF} \subseteq \Xi_{isp}$ then clearly $\mathsf{A} \sqsubseteq_{\Xi_{isp}} \mathsf{C} \Rightarrow \mathsf{A} \sqsubseteq_{\Xi_{sF}} \mathsf{C}$

To establish that $\mathsf{A} \sqsubseteq_{\Xi_{isp}} \mathsf{C} \Leftarrow \mathsf{A} \sqsubseteq_{\Xi_{sF}} \mathsf{C}$ we assume $\mathsf{A} \sqsubseteq_{\Xi_{sF}} \mathsf{C}$ and for some new context x we have $\rho \in Tr^c([\mathsf{C}]_x)$.

As the context x is an isp it must be in a state from which either one active action $\overline{\mathsf{a}}$ is enabled or a set X of passive actions are enabled.

Case 1 - one active action $\overline{\mathsf{a}}$ is enabled. Hence $(\rho, \{\overline{\mathsf{a}}\}) \in sF(\mathsf{C})$ and as $\mathsf{A} \sqsubseteq_{\Xi_{sF}} \mathsf{C}$ we can conclude $(\rho, \{\overline{\mathsf{a}}\}) \in sF(\mathsf{A})$. From which we have $\rho \in Tr^c([\mathsf{A}]_x)$.

Case 2 - a set X of passive actions are enabled and hence $(\rho, X) \in F(\mathsf{C})$. Because C is an isp this test is redundant as we will now show.

As C is an isp it must be in a state from which either:

Case 2a - one active action $\overline{\mathsf{b}}$ is enabled. $\rho \in Tr^c([\mathsf{C}]_x)$ is true if and only if $\rho\overline{\mathsf{b}} \in Tr^c([\mathsf{C}]_{x\mathsf{b}})$ where context $x\mathsf{b}$ is built from context x by adding action b to all nodes reached by ρ. Hence $\exists Y.(\rho\overline{\mathsf{b}}, Y) \in sF(\mathsf{C})$. As $\mathsf{A} \sqsubseteq_{\Xi_{sF}} \mathsf{C}$ we can conclude $(\rho\overline{\mathsf{b}}, Y) \in sF(\mathsf{A})$ and $\rho\overline{\mathsf{b}} \in Tr^c([\mathsf{A}]_{x\mathsf{b}})$. And finally $\rho \in Tr^c([\mathsf{A}]_x)$.

Case 2b - a set Z of passive actions are enabled. $\rho \in Tr^c([\mathsf{C}]_x)$ is true if and only if $\rho\mathsf{z} \in Tr^c([\mathsf{C}]_{x\bar{\mathsf{z}}})$ where context $x\bar{\mathsf{z}}$ is built from context x by adding action $\bar{\mathsf{z}}$, where $\mathsf{z} \in Z$, to all nodes reached by ρ. Hence $\exists Y.(\rho\mathsf{z}, Y) \in sF(\mathsf{C})$. As $\mathsf{A} \sqsubseteq_{\Xi_{sF}} \mathsf{C}$ we can conclude $(\rho\mathsf{z}, Y) \in sF(\mathsf{A})$ and $\rho\mathsf{z} \in Tr^c([\mathsf{A}]_{x\bar{\mathsf{z}}})$. And finally $\rho \in Tr^c([\mathsf{A}]_x)$.

We conclude $\rho \in Tr^c([\mathsf{A}]_x)$ as it is true in all cases. Hence we have $Tr^c([\mathsf{C}]_x) \subseteq Tr^c([\mathsf{A}]_x)$ for all isp contexts x. Thus by definition $\mathsf{A} \sqsubseteq_{\Xi_{sF}} \mathsf{C}$ •

5 Conclusion

In [Section 2] we defined a small set of process terms and an operational semantics, similar to that of CCS and CSP process terms. By making a small change to this process semantics we were able to model interacting sequential programs in [Section 4]. For interacting sequential programs, synchronising action pairs consist of one active action calling the other passive action. This model we claim is readily implementable and using the generalised refinement of [Reeves and Streader 2003] we construct a definition of refinement that we show to be singleton failure refinement. Singleton failure refinement is known to be the semantics of ADT and programs [Bolton and Davies 2001].

The only place where our parallel composition, unlike that for process algebra, introduces nondeterminism is when there are two active processes racing to perform actions. Hence in our (untimed) model this naturally causes nondeterminism.

We have so far found the analogy (between constructivity, implementability and determinism) fruitful as a way of suggesting questions that do not usually get asked in the process algebra world. It remains to be seen whether we can go further and give a model for some process algebra in a constructive logic as Martin-Löf did for functional programs.

References

[Martin-Löf 1985] Martin-Löf, P.: Constructive Mathematics and Computer Science. Mathematical Logic and Programming Languages. Eds. C.A.R. Hoare and J.C. Shepherdson. Prentice Hall International, Englewood Cliffs, N.J. (1985)

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

[Milner 1989] Milner, R.: Communication and Concurrency. Prentice-Hall International (1989)

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

[Bolton and Davies 2001] Bolton, C., Davies, J.: A singleton failures semantics for Communicating Sequential Processes. Research Report PRG-RR-01-11, Oxford University Computing Laboratory (2001)

[Hoare 1985] Hoare, C.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science (1985)

[Brinksma et al. 1996] Brinksma, E., Rensink, A., Vogler, W.: Applications of fair testing. In: FORTE. (1996) 145–160

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

[Reeves and Streader 2004] Reeves, S., Streader, D.: Atomic Components. Technical report, University of Waikato (2004) Computer Science Technical Report 01/2004, http://www.cs.waikato.ac.nz/~dstr.

[Reeves and Streader 2003] Reeves, S., Streader, D.: Comparison of data and process refinement. In Woodcock, J., Dong, J., eds.: Proceedings of ICFEM 2003. Number 2885 in LNCS. Springer-Verlag (2003) 266–285

[van Glabbeek 2001] van Glabbeek, R.L.: The linear time - branching time spectrum I. the semantics of concrete sequential processes. In Bergstra, J., Ponse, A., Smolka, S., eds.: Handbook of Process Algebra. Elsevier Science, Amsterdam, The Netherlands (2001) 3–99

[de Nicola and Hennessy 1984] de Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science **34** (84)

[Hennessy 1988] Hennessy, M.: Algebraic Theory of Processes. The MIT Press (1988)