State-based and process-based value-passing

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

Abstract

State-based and process-based formalisms each come with their own distinct set of assumptions and properties. To combine them in a useful way it is important to be sure of these assumptions in order that the formalisms are combined in ways which have, or which allow, the intended combined properties. Consequently we cannot necessarily expect to take one state-based formalism and one process-based formalism and combine them and get something sensible, especially since the act of combining can have subtle consequences.

Here we concentrate on value-passing, how it is treated in each formalism, and how the formalisms can be combined so as to preserve certain properties. Specifically, the aim is to take from the many process-based formalisms definitions that will best fit with our chosen state-based formalism, namely Z, so that the fit is simple, has no unintended consequences and is as elegant as possible.

Keywords: transition systems, Z, value-passing.

1 Introduction

Z is very expressive but leaves much unformalized (though this may also be seen, in some lights, as a strength). For example, Z has no formal ability to combine operation schemas into an abstract data type that encapsulates the data. Further, Z has a practical tradition where it is the responsibility of the specifier not to specify things which do not make sense according to the intended (perhaps informal) model. For example, when intending to bundle operations together so as to model some of the ideas behind abstract data types (ADTs, as is often done when discussing data refinement for Z) we would not want to specify input operations that overly constrain the value to be input, *e.g.* a 'pop' operation which only allows the value 2 to be put read from a stack.

Process algebras, on the whole, come from a more formal, abstract, 'theoretical' tradition which tends to mean that there are as many distinct, well-defined algebras as there are ways of varying definitions while still retaining a sound theory. Equivalence is an important property in any algebra, so there are many different possibilities for equivalence in the process-based world. In particular we should not be trapped into thinking that the equivalence that we are most familiar with, or which is most widely used, is the best for a given problem.

To give some idea of how much choice there is, we note that in [22] a survey of 155 testing semantics (often used to characterize equivalence) can be found (and these only deal with atomic actions). So, there is rather a large task to carry out when choosing the right equivalence even in this simple situation. In this paper, to make matters worse, we are interested in value-passing actions.

The usual (failures) value-passing semantics of CSP treats "a?1", "a?2","a!1" and "a!2" as distinct atomic observable actions. A consequence of failure (and singleton failure) equivalences using the atomic actions in their refusal sets is that an observer can select the value to be passed to an action **before** the action is performed. That is to say, a program can perform "pop?2" which will only pop a value off a stack if the value is to be popped is 2.

Clearly this is not the normal ADT operational semantics of applying pop to a stack. We claim that *the normal ADT operational semantics for value-passing* is that an input operation cannot use the value to be input to prevent the action being performed (and, of course, we can choose to use Z in exactly this way if we are intending to model aspects of ADTs using Z).

Now, failure and singleton failure equivalences are just two of very, very many (at least 155!) process equivalences and, if we want an equivalence which *does* meet our claim about ADTs, then we can look around and find, eventually, in [7] that they define **named failure equivalence** (our name for it) which *has* the normal ADT operational semantics for value-passing.

Of course, as shown in [2], we can keep CSP's (singleton failure) value-passing semantics and define a denotational

model for Z ADTs, but that clearly means we have to delineate a subset of all of CSP as having meaning in this situation (and this is exactly what [2] does), since some CSP (under failure semantics) will not correspond to Z ADTs. This then leaves open the possibility of that subset losing some desirable properties, like being closed under CSP's operators, which may or may not be seen as a bad thing.

If, as here, we are interested primarily in operational semantics we can then, by standard means, express the operational semantics of Z ADTs via (a subset of) labelled transition systems (LTSs, of which more later). On this subset both CSP's and Z ADT's value-passing semantics coincide, but it is a somewhat tortuous route.

To spell the situation out, this approach has several drawbacks:

1. it complicates the otherwise very simple operational interpretation of Z
2. it requires syntactic constraints on what CSP terms constitute an ADT (process) and what constitutes a program (context)
3. the definition of ADTs will not be closed under the usual CSP operators
4. mapping Z directly to a denotational semantics limits the equality we can use and prevents us making use of the large amount of work relating operational semantics with testing semantics [21, 22, 20].

As we will see, choosing a different equivalence (*i.e.* different semantics) for CSP, one where value-passing fits the ADT requirements better, eases the situation. In particular all CSP terms now have the right semantics, so the whole language is "used" and we now have closure under the operators.

We close this introduction with some more comments:

Mixing Z specifications with CSP's internal and external choice[5, 3] makes defining an operational semantics quite problematic. Given that as Roscoe said [15] [p.178] "the operational semantics of CSP was created to give an alternative view to the already existing denotational models rather that providing the intuition in the original design" it should not be surprising that the CSP operational semantics [15] [Ch. 7] is different to both CCS [14] and ACP [1]. Significantly CSP hiding does not distribute through CSP choice, whereas in ACP the renaming of an action as a τ action does distribute through ACP choice. The reason being that CSP uses τ to model "nondeterministic state". This is **not** the same as CCS/ACP where τ is used to model an unobservable action.

Failure and divergence semantics have been defined for Z specifications in [17, 5, 3]. This we believe to be unfortunate, as we agree with Leduc [12] that failure and divergence semantics is "not adequate w.r.t. the operational interpretation" of divergence.

Here we define an operational semantics via labelled transition systems (LTS) for Z, which is essentially the same as in [16]. We then define an isomorphism between the operational semantics and Z's partial relation semantics for operations, defined as sets of bindings [18]. This isomorphism is no more than a simple syntactic reordering of the underlying set theoretic formalizations.

We postpone considering which equivalence/preorder we are interested in until after we have related the two semantic models.

1.1 Common operational semantics

We first map a Z specification to a labelled transition system (definition below), just as process algebras are mapped to such systems in [1]. Here we will treat Z and value-passing process algebras as alternative styles of specification, not as specifying different kinds of things. Having built a LTS, the Z or the process algebra can be forgotten. This allows us to make use of the extensive existing work defining equivalences and refinement relations of LTSs. The work includes denotational semantics, testing semantics and many full abstraction proofs. See [21, 22, 20] for surveys of many such results. We believe that understanding the testing characterization of process equivalences/preorders can both guide and inform the definition of equivalences/preorders for ADTs.

It has been shown in [20] that failure semantics can be characterized by a testing semantics and that amending this by the addition of 'lights' to show the availability of an action results in a testing semantics that characterizes ready trace semantics. The contexts (tests) of [3] characterize failure refinement/equivalence. [16] extends these contexts by adding a "pre" construct, and all we need to do is show that the semantics of this construct is the same as that of a 'light', and we have, for free, a proof of full abstraction with ready trace refinement/equivalence.

1.2 Value-passing events

Process semantics is usually built on a transition system, where each transition is labelled with an event and whether or not two events synchronize is controlled by the the event name. An input *event* $\mathsf{a}?v$ consists of the value v being input by a and the output *event* $\mathsf{a}!v$ consists of the value v being output by a. We will be refer to a as the **name** of events $\mathsf{a}!v$ and $\mathsf{a}?v$

A process that calls an output event (*i.e.* it 'demands' that something performs output) cannot select the value to be output: what that is is up to the something performing the output. A program (which is just another process) with

event named $\overline{\mathsf{pop}}$ that calls the event named pop from an ADT (a stack in this case) must accept the value returned. A program cannot call $\overline{\mathsf{pop}?1}$ with the requirement that it would only be executed if the value on the top of the stack is 1. Similarly, if $\mathsf{push}?x$ for some x is required of the ADT then the ADT must accept the (well-typed) value pushed and, for example, $\mathsf{push}?1$ is not allowed as a 'condition' in the definition of the ADT.

Assumption 1 *The value to be output by an event is under local control,* i.e. *the input event that it synchronizes with cannot select or restrict, the value.*

This assumption is satisfied by a process algebra ([7]) that has input events $\mathsf{a}?x$ where x is a variable and where named failure semantics (the refusal set is a set of names not events) is used.

To satisfy this assumption we restrict what we accept as a Z specification of an ADT.

The contexts in which ADTs are placed are the programs that execute them. These programs consist of sequences of operations (defined by the ADT), whereas processes are able to be placed in contexts that may have a branching structure of operations.

Assumption 2 *ADTs are placed in contexts that are programs and programs consist of sequences (not trees) of operations.*

Assumption 2 is the motivation behind singleton failure semantics ([2]) which is a variation on failure semantics where the refusal set must be a singleton set.

Here we are going to combine Assumption 1 and Assumption 2, which not surprisingly will give a semantics which is a variation on failure semantics where the refusal set must be a singleton set containing the names of actions.

We introduce healthiness conditions on a Z ADT A so as to satisfy Assumption 1 and similar healthiness conditions on both the relational semantics $[\![\mathsf{A}]\!]_R$ and the operational (LTS) semantics $[\![\mathsf{A}]\!]_g$. We then define obvious mappings ${}_g[\![\_]\!]_R$ from LTS to relations and its inverse ${}_R[\![\_]\!]_g$. We establish that these mappings commute

$$ {}_R[\![[\![\mathsf{A}]\!]_R]\!]_g = [\![\mathsf{A}]\!]_g $$

for any A. Finally, we will see that refinement being subset of relations corresponds, under our mapping, to refinement being subset of singleton named failures.

We do not see these results as surprising, given that very similar results were shown in [2]. Our approach differs (see Section 8) in that, firstly, we more closely follow Assumption 1 and, secondly, we are more able to mix the use of Z with a process algebra without needing to extend what could be specified directly in Z, as we shall see.

Because we are interested in combining different formalisms at the semantic level we do not wish to restrict the LTSs we are able to consider hence we use the semantic mapping from Z to LTSs of [17, 16, 4] .

2 Terminology

In this section we define the notation we will use. It is a combination of notation from ACP [1] and Z [19]. We assume a universe of observable event names *Act* over which a will range. Let $io \stackrel{\text{def}}{=} \{!v \mid v : V\} \cup \{?v \mid v : V\}$, $Events \stackrel{\text{def}}{=} \{\mathsf{a}io_{\mathsf{a}} \mid \mathsf{a} \in Act \wedge io_{\mathsf{a}} \in io\}$, $Events^\tau \stackrel{\text{def}}{=} Events \cup \{\tau\}$, $a \in Events^\tau$ and $\rho \in (Events^\tau)^*$. Also let $\overline{Act} \stackrel{\text{def}}{=} \{\overline{\mathsf{a}} \mid \mathsf{a} \in Act\}$.

We write ρ_{n} for the sequence of event names in ρ. We will write ρ_{in} for the sequence of input values built from ρ by regarding output events to have blank _ inputs. Similarly we write ρ_{out} for the sequence of output values. There is clearly an isomorphism between ρ and $\langle \rho_{\mathsf{n}}, \rho_{in}, \rho_{out} \rangle$.

Definition 1 *labelled Transition Systems (LTSs)*
$\mathsf{A} \stackrel{\text{def}}{=} (Nodes_{\mathsf{A}}, Tran_{\mathsf{A}}, s_{\mathsf{A}})$ *where* $Nodes_{\mathsf{A}} \stackrel{\text{def}}{=} \{\eta \mid \eta : Var \mapsto Val\}$, $s_{\mathsf{A}} \in Nodes_{\mathsf{A}}$ *and*
$Tran_{\mathsf{A}} \stackrel{\text{def}}{=} \{n \xrightarrow{a} m \mid n, m \in Nodes_{\mathsf{A}} \wedge a \in Events^\tau\}$.

We lift "$-$" to sets of transitions and to labelled transition systems in the obvious way. Any single labelled transition system will either have transitions labelled from $Events^\tau$ or transitions labelled from $\overline{Events}^\tau$ (*i.e.* if it is a context or program).

We write $\rho\ |_n$ for the n^{th} element of ρ and $\rho\ \restriction_n$ for the first n elements of ρ. We write $\rho \restriction X$ for the sequence ρ with all elements not in set X removed, so $prefix(\rho) \stackrel{\text{def}}{=} \{\rho\ \restriction_n \mid n <| \rho\ |\}$.

Where A is obvious from context, we write: $n \xrightarrow{a} m$ for $(n, a, m) \in Tran_{\mathsf{A}}$, $\pi(s) \stackrel{\text{def}}{=} \{a \mid s \xrightarrow{a}\}$ $\pi_n(s) \stackrel{\text{def}}{=} \{a \mid s \xrightarrow{a!x} \vee\ s \xrightarrow{a?x}\}$ $n \xrightarrow{\rho} m$ for $\exists_{m_1 \ldots m_i} .(m_1, \rho\ |_1, m_2), \ldots (m_i, \rho\ |_i, m_{i+1}) \in Tran_{\mathsf{A}} \wedge n = m_1 \wedge m = m_{i+1} \wedge |\ \rho\ | = i$.

Assume a set of state variables *Var* and let η range over *evaluations* $\stackrel{\text{def}}{=} Var \to Val$.

Let $\widetilde{X_E}(s) \subseteq Events - \pi(s)$, $\widetilde{a_E}(s) \in Events - \pi(s)$ $\widetilde{X_n}(s) \subseteq Act - \pi_n(s)$, $\widetilde{a}(s) \in Act - \pi_n(s)$

Traces $Tr(\mathsf{A}) \stackrel{\text{def}}{=} \{\rho \mid s_{\mathsf{A}} \xrightarrow{\rho}\}$ and complete traces $Tr^c(\mathsf{A}) \stackrel{\text{def}}{=} \{\rho \mid s_{\mathsf{A}} \xrightarrow{\rho} n \wedge (\pi(n) = \varnothing \vee |\ \rho\ | = \infty)\}$.

$F(\mathsf{A}) = \{\langle \rho, \widetilde{X_E}(s) \rangle \mid s_{\mathsf{A}} \xrightarrow{\rho} s\}$ failure [9]

$sF(\mathsf{A}) = \{\langle \rho, \widetilde{a_E}(s) \rangle \mid s_{\mathsf{A}} \xrightarrow{\rho} s\}$ singleton failure [2]

$nF(\mathsf{A}) = \{\langle \rho, \widetilde{X_n}(s) \rangle \mid s_{\mathsf{A}} \xrightarrow{\rho} s\}$ named failure [7]

$nsF(\mathsf{A}) = \{\langle \rho, \widetilde{a}(s) \rangle \mid s_{\mathsf{A}} \xrightarrow{\rho} s\}$ named singleton failure

$(\mathsf{A})\delta_X \stackrel{\text{def}}{=} (Nodes_{\mathsf{A}}, Tran_{(\mathsf{A})\delta_X}, s_{\mathsf{A}})$ where $Tran_{(\mathsf{A})\delta_X} \stackrel{\text{def}}{=} \{n \xrightarrow{\mathsf{a}} m \mid n \xrightarrow{\mathsf{a}} m \in Tran_{(\mathsf{A})} \wedge \mathsf{a} \notin X\}$.

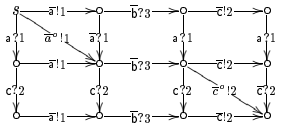
$(\mathsf{A})\tau_X \stackrel{\text{def}}{=} (Nodes_{\mathsf{A}}, Tran_{(\mathsf{A})\tau_X}, s_{\mathsf{A}})$ where $Tran_{(\mathsf{A})\tau_X} \stackrel{\text{def}}{=} \{n \xrightarrow{\mathsf{a}} m \mid n \xrightarrow{\mathsf{a}} m \in Tran_{(\mathsf{A})} \wedge \mathsf{a} \notin X\} \cup \{n \xrightarrow{\tau} m \mid n \xrightarrow{\mathsf{a}} m \in Tran_{(\mathsf{A})} \wedge \mathsf{a} \in X\}$.

The synchronization function γ_X which maps $(a, b) \mapsto c$ adds c, representing the synchronization of a and b, where a and b could be performed concurrently see [1] for detailed discussion and Figure 1 for an example.

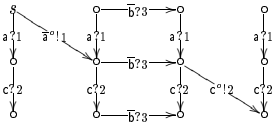
We treat the synchronization of y and $\overline{y}$ as giving the observable $\overline{y}$. In detail (which the reader may skip), in order to do this, and allow the deletion of unsynchronized $\overline{y}$ actions, we first map them to $\overline{y}^o$ (so γ_Y contains $(\mathsf{y}!v, \overline{\mathsf{y}}?v) \mapsto \overline{y}^o?v$ and $(\mathsf{y}?v, \overline{\mathsf{y}}!v) \mapsto \overline{y}^o!v$) then delete $\overline{\mathsf{y}}$ via $\delta_{\overline{Y}}$ and then rename $\overline{y}^o$ to $\overline{\mathsf{y}}$ via Ren_Y. All this is brought together in the following definition (*e.g.* Figure 1):

$$\_ \|_Y \_ \stackrel{\text{def}}{=} (((\_ \|_{\gamma_Y} \_)\delta_{\overline{Y}})Ren_Y)\delta_{Act}$$

$\mathsf{P} = \overline{\mathsf{a}}!1\overline{\mathsf{b}}?2\overline{\mathsf{c}}!2 \|_{\gamma_{\{\overline{\mathsf{a}},\overline{\mathsf{c}}\}}} \mathsf{a}?1\mathsf{c}?2$



$\mathsf{Q} = (\mathsf{P})\delta_{\{\overline{\mathsf{a}},\overline{\mathsf{c}}\}}$



$\mathsf{R} = (\mathsf{Q})Ren_{\{\overline{\mathsf{a}},\overline{\mathsf{c}}\}}\delta_{Act}$

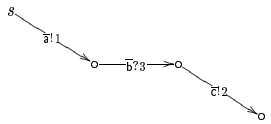


Figure 1. $\overline{\mathsf{a}}!1\overline{\mathsf{b}}?x\overline{\mathsf{c}}!2 \|_{\{\overline{\mathsf{a}},\overline{\mathsf{c}}\}} \mathsf{a}?y\mathsf{c}?x$

3 Z and abstract data types

An abstract data type (ADT, defined using Z) A consists of a state schema $State_\mathsf{A}$, an initialization schema $init_\mathsf{A}$ and a set of operation schemas Op_A. A Z schema can be freely interpreted as a set of bindings *i.e.* mappings from names of observations to values, or the predicate that defines the set. We write:

$$\mathsf{A} \stackrel{\text{def}}{=} \langle State_\mathsf{A}, init_\mathsf{A}, Op_\mathsf{A}\rangle$$

To match the usual process algebra convention we will restrict what we regard as data types in two ways. Firstly, events have either input or output but not both. Secondly, the initialization schema must define a single state. These restrictions ease the definitions.

For any operation a we will interpret a and $\overline{\mathsf{a}}$ that do not pass values as "sugar" for $\mathsf{a}!\epsilon$ and $\overline{\mathsf{a}}?\epsilon$ where ϵ is a special unreferenced variable of singleton type $\epsilon : \{*\}$.

We write $name_\mathsf{A}$ for the names of the operation schemas of A and io_a for the input or output value of operation a of ADT A (using the context to disambiguate if necessary). Finally $Events_\mathsf{A} \stackrel{\text{def}}{=} \{\mathsf{a}io_\mathsf{a} \mid \mathsf{a} \in name_\mathsf{A}\}$.

For an example, see Figure 2 where $name_\mathsf{A} = \{\mathsf{a}, \mathsf{b}\}$, $Events_\mathsf{A} = \{\mathsf{a}!*\} \cup \{\mathsf{b}!v \mid v : V\}$, $io_\mathsf{a} = !\epsilon$ and $io_\mathsf{b} = !x$.

To satisfy Assumption 1 we require:

Healthiness 1 *An operation schema* $S = [d : D;\ d' : D';\ x? : V \mid P(d, d', x?)]$ *with input must satisfy* $\forall d \bullet (\exists v \bullet \exists d' \bullet P(d, d', v) \Leftrightarrow \forall v \bullet \exists d' \bullet P(d, d', v))$.

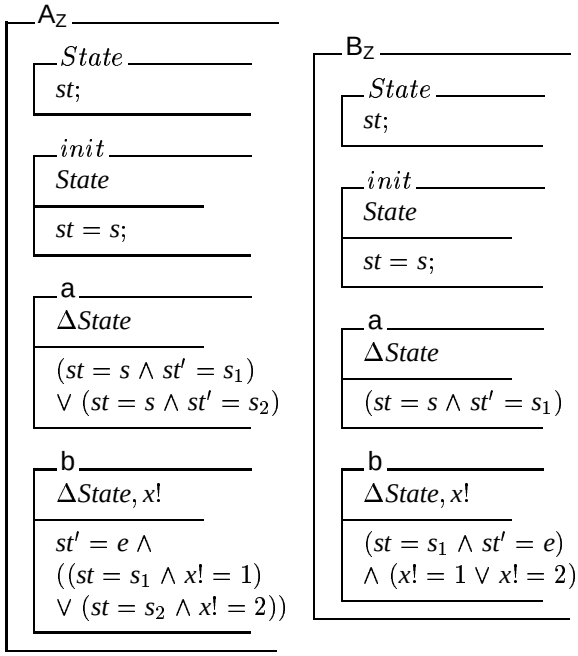


Figure 2. A_Z **and** B_Z

4 Z relational semantics

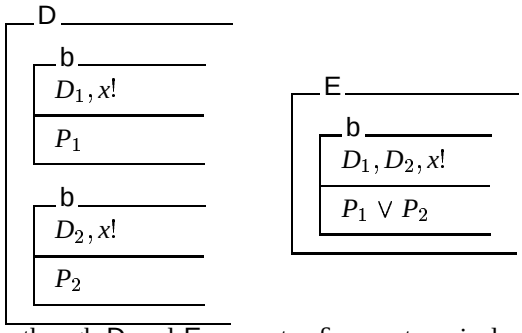
As is well known, a relational semantics can be given for Z (see [2]) which gives each operation a a partial relation $[\![\mathsf{a}]\!]_R \subseteq (State \times input) \times (State \times output)$. From this we define:

$$[\![Op_\mathsf{A}]\!]_R \stackrel{\text{def}}{=} \{\langle \mathsf{a}, [\![\mathsf{a}]\!]_R\rangle \mid \mathsf{a} \in name_\mathsf{A}\}$$

$$[\![\mathsf{A}]\!]_R \stackrel{\text{def}}{=} \langle State_\mathsf{A}, init_\mathsf{A}, [\![Op_\mathsf{A}]\!]_R\rangle$$

Using this we find that, for example in Figure 2, $[\![\mathsf{A}_Z]\!]_R \neq [\![\mathsf{B}_Z]\!]_R$ although, as we shall see, they are refinement equivalent.

Later on we will need to consider ADTs like D below, which looks very odd. In a program like $\overline{\mathsf{b}}$ the natural question is "which b is being called?". The answer is "it does not matter", so we can think of E as equivalent to D, and so the question does not arise.



So, though D and E are not refinement equivalent under [24, 3] (as their definition of refinement uses an indexed set of operations), they are semantically equivalent and refinement equivalent according to our definitions. (However, this distinction we do not feel to be important as both definitions coincide when a data type has operational schemas with distinct names.)

4.1 Refinement on relational semantics

Bolton and Davies [2] adopt a 'guarded outside of precondition' interpretation in their construction of a total relation for each operation. These total relations are then composed to give a relational semantics of *programs* ρ_{n}, *i.e.* sequences of names of 'called' events, *e.g.* $\overline{\mathsf{push}}\,\overline{\mathsf{pop}}\,\overline{\mathsf{push}}\,\overline{\mathsf{pop}}$. Data refinement $\mathsf{A} \sqsubseteq_R \mathsf{C}$ is defined to hold if, for any program ρ_{n} calling operations from C, its semantics is a subset of ρ calling operations from A.

The semantics of the programs

$$\rho_{\mathsf{n}} \stackrel{\text{def}}{=} \mathit{init}, \overline{\mathsf{a}^1_{\mathsf{n}}}\,\overline{\mathsf{a}^2_{\mathsf{n}}} \ldots \mathit{final}$$

on a data type is constructed from the relational semantics of the operations, plus an initialization and finalization relation.

The semantics of a program is a relation[1] $\mathit{input}^* \times \mathit{output}^*$ from a sequences of inputs to a sequence of outputs. For details of this construction see [24, 2, 3] here we only need the result of the construction.

$$\begin{aligned} {}_{\mathsf{A}}[\![\rho_{\mathsf{n}}]\!]_R \stackrel{\text{def}}{=} \{\langle \rho_{in}, \rho_{out}\rangle \mid\ & (|\,\rho_{in}\,| = |\,\rho_{out}\,| \wedge\ \rho \in \mathit{Tr}^c(\mathsf{A})) \vee \\ & (|\,\rho_{in}\,| > |\,\rho_{out}\,| \wedge\ \rho\,|_{|\rho_{out}|} \in \mathit{Tr}^c(\mathsf{A}))\} \end{aligned}$$

$$\mathsf{A} \sqsubseteq_R \mathsf{C} \stackrel{\text{def}}{=} \forall\, \rho_{\mathsf{n}} \cdot {}_{\mathsf{A}}[\![\rho_{\mathsf{n}}]\!]_R \supseteq {}_{\mathsf{C}}[\![\rho_{\mathsf{n}}]\!]_R$$

The relational semantics of any *program* calling operations for A_Z is going to be the same as if it were run calling operations for B_Z (see Figure 3).

5 Operational semantics

The LTS semantics of a Z ADT A is defined by:

[1] The relation in [2] is actually given on $(\{\top, \bot\} \times \mathit{input}^*) \times (\{\top, \bot\} \times \mathit{output}^*)$ from $\top$ or $\bot$ and a sequence of inputs to $\top$ or $\bot$ and a sequence of outputs. But the distinction between $\bot$ and $\top$ is redundant as they can be inferred from the relative lengths of the input and output sequences.

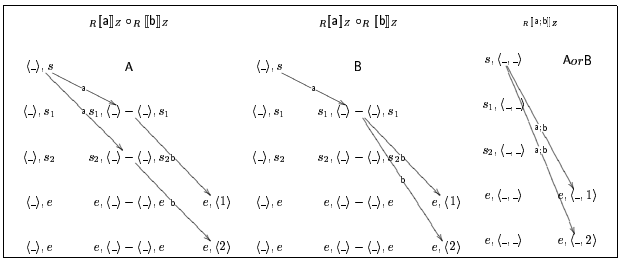


Figure 3. Z relational semantics

$$[\![\mathsf{A}]\!]_g \stackrel{\text{def}}{=} \langle \mathit{State}_{\mathsf{A}}, \mathit{init}_{\mathsf{A}}, [\![\mathit{Op}_{\mathsf{A}}]\!]_g \rangle$$

$$[\![\mathit{Op}_{\mathsf{A}}]\!]_g \stackrel{\text{def}}{=} \{x \xrightarrow{\mathsf{a}i o_{\mathsf{a}}} y \mid (\mathsf{a}i o_{\mathsf{a}} \in \mathit{Events}^{\tau}_{\mathsf{A}}) \wedge x \in \mathit{State}_{\mathsf{A}} \wedge \mathsf{a} \wedge y \in \mathit{State}'_{\mathsf{A}})\}.$$

For an example Figure 4 gives the LTS semantics of the ADTs in Figure 2.

The relation between $[\![\mathsf{A}]\!]_R$ and $[\![\mathsf{A}]\!]_g$ is straightforward. The nodes of $[\![\mathsf{A}]\!]_g$ and the states of $[\![\mathsf{A}]\!]_R$ are Z bindings. The meaning of an operation a $[\![\mathsf{a}]\!]_R$ is a relation between evaluations, labelled with a, which hence defines a set of a transitions of $[\![\mathsf{A}]\!]_g$. The initialization schema is restricted to a unique evaluation/node.

$${}_g[\![[\![\mathsf{A}]\!]_g]\!]_R \stackrel{\text{def}}{=} \langle \mathit{State}_{\mathsf{A}}, \mathit{init}_{\mathsf{A},g}\, [\![[\![\mathit{Op}_{\mathsf{A}}]\!]_g]\!]_R \rangle$$

$$\begin{aligned} {}_g[\![[\![\mathit{Op}_{\mathsf{A}}]\!]_g]\!]_R \stackrel{\text{def}}{=}\ & \{\langle \mathsf{a}, \langle \langle x, \_\rangle, \langle y, v\rangle\rangle\rangle \mid x \xrightarrow{\mathsf{a}!v} y\} \\ & \cup \{\langle \mathsf{a}, \langle \langle x, v\rangle, \langle y, \_\rangle\rangle\rangle \mid x \xrightarrow{\mathsf{a}?v} y\} \end{aligned}$$

$${}_R[\![\_]\!]_g \stackrel{\text{def}}{=} ({}_g[\![\_]\!]_R)^{-1}$$

Lemma 1 ${}_R[\![[\![\mathsf{A}]\!]_R]\!]_g = [\![\mathsf{A}]\!]_g$ *and* ${}_g[\![[\![\mathsf{A}]\!]_g]\!]_R = [\![\mathsf{A}]\!]_R$

5.1 Refinement on operational semantics

We define data refinement on the operational semantics as subset of the named singleton failure semantics:

$$\mathsf{A} \sqsubseteq_g \mathsf{C} \stackrel{\text{def}}{=} \mathit{nsF}([\![\mathsf{A}]\!]_g) \supseteq \mathit{nsF}([\![\mathsf{C}]\!]_g)$$

For example, $[\![\mathsf{A}_Z]\!]_g$ and $[\![\mathsf{B}_Z]\!]_g$ are not singleton failure equivalent but are named singleton failure equivalent.

Lemma 2 $\mathsf{A} \sqsubseteq_R \mathsf{C} \Leftrightarrow \mathsf{A} \sqsubseteq_g \mathsf{C}$

Proof see Lemmas 5 and 6 •

6 Testing semantics

It is usual, in the process algebra approach, to define contexts as process terms. We split this into two parts. First we

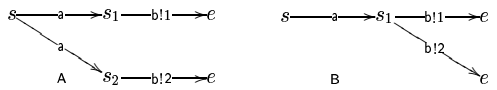


Figure 4. $\mathsf{A} = [\![\mathsf{A}_Z]\!]_g$ **and** $\mathsf{B} = [\![\mathsf{B}_Z]\!]_g$

will define terms for $[\_]\_$, with a hole for an ADT and a hole for a context. This defines the interaction between the ADT and the context. Below (Section 6.0.1) we will say what contexts represent value-passing programs.

Placing T in a context x is written $[\mathsf{T}]_x$ and must model the synchronization between operations of T (such as m) and operations of contexts (such as calling m, *i.e.* $\overline{\mathsf{m}}$).

The resulting synchronized operations may be private, *i.e.* τ, actions. Any action of the context that is not private is observable by an "independent observer". We are going to quantify over "all" contexts and it is easy to amend any context by adding actions that make observable any of the unobservable synchronizations. Consequently, although communication may be unobservable, we will treat it as observable. We will define the synchronization of y and $\overline{\mathsf{y}}$ to be $\overline{\mathsf{y}}$. In order to allow the deletion of unsynchronized $\overline{\mathsf{y}}$ actions we use $((\_ \parallel_{\gamma_Y} \_)\delta_{\overline{Y}})Ren_Y$ (see Section 2 above).

We assume that all observable actions of T require synchronization with some other thing in order to be performed. Hence, no observable action of T can be performed on its own (formalized by $(\_)\delta_{Act}$). So, we have

$$[\mathsf{T}]_x \stackrel{\text{def}}{=} \_ \parallel_Y \_ \stackrel{\text{def}}{=} ((\_ \parallel_{\gamma_Y} \_)\delta_{\overline{Y}})Ren_Y)\delta_{Act}$$

Further, we assume that we can wait long enough so that if something observable will eventually happen we do see it. We can only view our things via their synchronization with the context and we can view all synchronization with the context. This amounts to an observation being a *complete trace* (the set of observable traces is not prefix closed).

Hence

$$Obs([\mathsf{T}]_x) \stackrel{\text{def}}{=} Tr^c([\mathsf{T}]_x).$$

6.0.1 What terms are contexts?

Contexts are terms that correspond to value-passing programs that satisfy Assumption 1 and Assumption 2. These can only use a variable in an expression in an output action if it previously appears in an input action.

Let $\mathsf{a} \in Act$ and let us define operations as:

$$Oper_i \stackrel{\text{def}}{=} \{s \xrightarrow{\overline{\mathsf{a}}?x_i} r\} \cup \{s \xrightarrow{\overline{\mathsf{a}}!e_i(x_1,\ldots x_{j<i})} r\}$$

$$Exp\_prog \stackrel{\text{def}}{=} \{op_1 \ldots op_n \mid op_i \in Oper_i\}$$

Our terms, or *expression programs* $\rho_e \in Exp\_prog$, are quite different to the programs ρ_{n} of Section 4.1 In the appendix we will define ${}_{\mathsf{A}}[\![\rho_e]\!]_R$, the relational semantics of ρ_e, and hence data refinement as: $\mathsf{A} \sqsubseteq_{eR} \mathsf{C} \stackrel{\text{def}}{=} \forall \rho_{ex} \cdot {}_{\mathsf{A}}[\![\rho_e]\!]_R \supseteq {}_{\mathsf{C}}[\![\rho_e]\!]_R$. Using this we prove that $\mathsf{A} \sqsubseteq_{eR} \mathsf{C}$ if and only if $\mathsf{A} \sqsubseteq_R \mathsf{C}$ (see Lemma 4).

6.1 Refinement on testing semantics

A single observation of T in a context X is a complete trace of $[\mathsf{T}]_X$ and will be interpreted as $\top$ (success) if and only if it is also a complete trace of the context X. Being interested in nondeterminism we assume that an observation consists of a set of single observations of the same thing and context. Such observations are given one of the following three interpretations: $\{\top\}$—always succeed; $\{\top, \bot\}$—may succeed or may fail; and $\{\bot\}$—always fail.

Of the three power domains on the two point lattice $\top > \bot$ we are only interested in the *Smyth* power domain

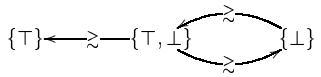


Figure 5. Smyth

Definition 2 . $[\_] \stackrel{\text{def}}{=} \{[\_]_x \mid x \in Exp\_prog\}$

$\top \in I([\mathsf{A}]_x) \Leftrightarrow \exists_{\rho \in Obs([\mathsf{A}]_x)} \cdot \rho \in Tr^c(x)$

$\bot \in I([\mathsf{A}]_x) \Leftrightarrow \exists_{\rho \in Obs([\mathsf{A}]_x)} \cdot \rho \in prefix(Tr^c(x))$ *and nothing else is in* $I([\mathsf{A}]_x)$.

$Obs([\mathsf{C}]_x) \gtrsim Obs([\mathsf{A}]_x) \stackrel{\text{def}}{=} I([\mathsf{C}]_x) > I([\mathsf{A}]_x) \vee (I([\mathsf{A}]_x) = I([\mathsf{C}]_x) \wedge Obs([\mathsf{A}]_x) \supseteq Obs([\mathsf{C}]_x))$

$\mathsf{A} \sqsubseteq_{Test} \mathsf{C} \stackrel{\text{def}}{=} \forall_{[\_]_x \in [\_]} .Obs([\mathsf{C}]_x) \gtrsim Obs([\mathsf{A}]_x).$ •

Lemma 3 $\mathsf{A} \sqsubseteq_R \mathsf{C} \Leftrightarrow \mathsf{A} \sqsubseteq_{Test} \mathsf{C}$

Proof see Lemmas 5 and 8 •

7 Mixing formalisms

In ACP, process terms are given an operational semantics and the operators are defined on the semantic domain. This is defined so that the semantic mapping distributes through the operators and consequently the operators have that same interpretation whether considering process terms or their semantics. Similarly operators that could be used to combine two state-based representations or two process-based representations and had the same interpretation on the underlying semantic model would, we believe, make specifications using both representations easier to understand. Ideally we would further like to be able to transform state-based and process-based representations into each other.

It is easy to see that the finite LTS can be converted into process terms and into a Z ADT by using an enumerated data type to represent state. It would be desirable that these semantic mappings should also distribute through all process operators.

With these goals in mind we will next investigate some of the process operators found in the literature.

The schema operation $\|_Z$ defined in [3] to compose two operation schemas can be lifted to the composition of two ADTs by simply composing operation schemas with the same name. The parallel composition of CSP ($\|_{CSP}$) and Object-Z ($\|_Z$) have subtly different interpretations. But the

ACP definition of parallel composition ($\|_\gamma$) is parameterized on a synchronization algebra γ and we can define γ to give either $\|_{CSP}$ or $\|_Z$. Consequently by using $\|_\gamma$ in place of $\|_{CSP}$ we do not need to have different notions of parallel composition in the state and action based approaches.

We interpret Z ADTs as processes for which the state is encapsulated and we interpret their operation schemas as a style of "symbolic transitions". With CSP and the symbolic transitions of processes defined in [8, 11] their is no explicit representation of state, and input variables $?x$ have meaning outside of the symbolic transition in which they appear. Alternatively, in the input/output automata of [6, 13] and the objects of [3] their is an explicit representation of state, and input variables are local to the transition (or operation schema). *Explicit state symbolic transitions* do not appear to have been considered in detail in the process algebra literature but appear to offer a very natural connection between the state and action based approaches.

In a formalism that has an explicit representation of both state and actions the process operators can be applied in at least three distinct ways. One, outside the data encapsulation, between two ADTs with separate **local states** $\mathsf{A} + \mathsf{B}$; two, inside the data encapsulation between ADTs/operation schemas that can **share state** $\mathsf{a} + \mathsf{b}$ and three, to build an operation schema (symbolic transition) from other operation schemas. All three, to some degree, have been used in [3].

7.1 Shared-state operators

The schema operations $[]$, $\|_Z$ for choice and parallel composition as defined in [3] compose two operation schemas and return a single operation schema, whereas the process algebra style of choice and parallel composition [8, 11] compose two symbolic transitions and return a process not a single symbolic transition.

The informal interpretation of an ADT (or object) is that a context can "choose" to execute one of the enabled actions. In AplusB Figure 6 choosing action a does not prevent a subsequent choice of b. This property is not true of choice as formalized in CSP, CCS or ACP, but the choice defined in [23] does satisfy this property. Consequently if we were to formalise operational schema as symbolic transitions then AplusB could only equal $\mathsf{a} + \mathsf{b}$ if we uesd the choice in [23].

Symbolic actions in process algebras can distinguish successful termination from deadlock and a symbolic transition is seen as a special type of process. Whereas in state-based approaches like Object Z an operation schema cannot distinguish successful termination from deadlock and on it own is not an object. One way to unify the approaches would be to add an explicit successful termination to both Z ADTs and operation schemas. The importance of this is not new and can be seen in Unifying Theories of Programming [10].

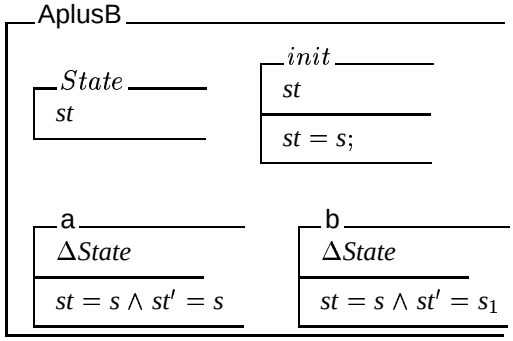


Figure 6. Is AplusB **=** $\mathsf{a} + \mathsf{b}$ **?**

7.2 Local state operators

The following definition identifies the initial states of A and B. Although this is conceptually simple it is not so easy to define in Z. To make the definition easier we have assumed that the state of an ADT is *lifted* over $\bot$.

ACP's definition of choice requires "root unwinding" which would have been both messy to define in Z and, we feel, less natural than the definition from [23] that we have used.

$$\begin{aligned}
\mathsf{A} + \mathsf{B} \stackrel{\text{def}}{=}\ & \langle State_{\mathsf{A}+\mathsf{B}}, init_{\mathsf{A}+\mathsf{B}}, Op_{\mathsf{A}+\mathsf{B}} \rangle \text{ where} \\
State_{\mathsf{A}+\mathsf{B}} \stackrel{\text{def}}{=}\ & [State_{\mathsf{A}}; State_{\mathsf{B}} \mid init_{\mathsf{A}} \Leftrightarrow init_{\mathsf{B}} \wedge \\
& \quad \neg\ init_{\mathsf{A}} \Rightarrow State_{\mathsf{B}} = \bot \wedge \\
& \quad \neg\ init_{\mathsf{B}} \Rightarrow State_{\mathsf{A}} = \bot] \\
init_{\mathsf{A}+\mathsf{B}} \stackrel{\text{def}}{=}\ & init_{\mathsf{A}} \\
Op_{\mathsf{A}+\mathsf{B}} \stackrel{\text{def}}{=}\ & Op_{\mathsf{A}} \cup Op_{\mathsf{B}}.
\end{aligned}$$

For an example of $\mathsf{A} + \mathsf{B}$ see Figure 7.

8 Z to CSP in [2]

Bolton and Davies [2] define a CSP semantics for Z ADTs $[\![\_]\!]_{CSP}$.

They build CSP processes by taking the internal choice between output values. This can be motivated by Assumption 1. But, in [2] their ADTs (their name CDTs) are not subject to the constraint Healthiness 1 and hence operations such as push?1 can be defined, hence their approach does not completely satisfy Assumption 1. However, their semantic function has a range on which singleton failure and named singleton failure semantics give the same definition of refinement/equality.

For example, in the CSP semantics from [2] $[\![\mathsf{A}_Z]\!]_{CSP} = \mathsf{A}(\mathsf{s})$ and $[\![\mathsf{B}_Z]\!]_{CSP} = \mathsf{B}(\mathsf{s})$ where:

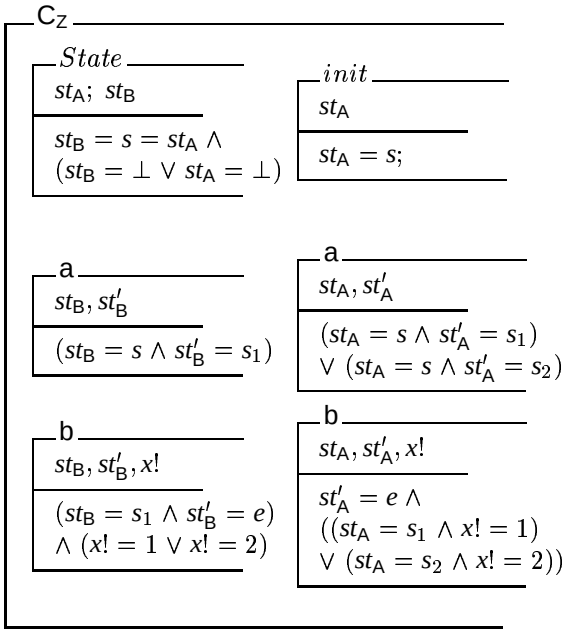


Figure 7. $\mathsf{C_Z} = \mathsf{A_Z} + \mathsf{B_Z}$

$$
\begin{aligned}
\mathsf{A(s)} &\stackrel{\text{def}}{=} \mathsf{a} \rightarrow \mathsf{A(s_1)} \sqcap \mathsf{a} \rightarrow \mathsf{A(s_1)} \\
\mathsf{A(s_1)} &\stackrel{\text{def}}{=} \mathsf{b!1} \rightarrow \mathsf{A(e)} \\
\mathsf{A(s_2)} &\stackrel{\text{def}}{=} \mathsf{b!2} \rightarrow \mathsf{A(e)} \\
\mathsf{B(s)} &\stackrel{\text{def}}{=} \mathsf{a} \rightarrow \mathsf{B(s_1)} \\
\mathsf{B(s_1)} &\stackrel{\text{def}}{=} \mathsf{b!1} \rightarrow \mathsf{B(e)} \sqcap \mathsf{b!2} \rightarrow \mathsf{B(e)}.
\end{aligned}
$$

The operational semantics of $\mathsf{B(s)}$, as defined in [15], is failure equivalent to A not B (from Figure 4).

Choice ($+$ or $\Box$) is **not closed** on the range of this CSP semantics *e.g.* The operational semantics of $\mathsf{a} \rightarrow (\mathsf{b!1} \rightarrow \mathit{nil} \Box \mathsf{b!2} \rightarrow \mathit{nil})$ is B from Figure 4 and this is not the operational semantics of $[\![\mathsf{X_Z}]\!]_{CSP}$ for any Z specification $\mathsf{X_Z}$.

9 Conclusions

We believe that the most understandable and useful connection between state-based Z and process-based formalisms is the simple isomorphism between Z's relational semantics and a process algebra's operational semantics. Given this, deciding to combine Z with CSP (as commonly interpreted) we believe to show a poor choice as far as the process algebra component is concerned because (1) its equivalences do not have the normal value-passing semantics; and (2) its use of τ actions is not based on the simple operational intuitions found in CCS/ACP; and (3) the operational interpretation of divergence is not adequate ([12]).

We have amended the named failure semantics of [7] into named singleton failure following [2], where failure semantics was amended into singleton failure. Then we selected the choice operator from [23] which we believe is closer to the normal state-based intuition and have shown how the operator can be defined on Z ADTs. The advantages are:

1. a simpler operational semantics for Z
2. process operators that are closed on the operational semantics of Z
3. hiding distributes through the process operators—this should ease the definition of a weak congruances

References

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

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

[3] J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology. Springer, May 2001.

[4] J. Derrick, E. Boiten, H. Bowman, and M. Steen. Specifying and Refining Internal Operations in Z. *Formal Aspects of Computing*, 10:125–159, December 1998.

[5] C. Fischer. CSP-OZ: a combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 423–438, Canterbury, UK, 1997. Chapman and Hall, London.

[6] S. Garland, N. Lynch, and M. Vaziri. IOA: A language for specifying, programming, and validating distributed systems. Technical report, MIT Laboratory fro Computer Science, 1997.

[7] M. Hennessy and A. Ingolfsdottir. A theory of communicating processes with value passing. *Information and computation*, 107:202–236, 1993.

[8] M. Hennessy and H. Lin. Symbolic bisimulation. *Theoretical Computer Science*, pages 353–389, 1995. 138.

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

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

[11] A. Ingolfsdottir and H. Lin. A Symbolic Approach to Value-passing Processes. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, chapter 7. Elsevier Science, Amsterdam, The Netherlands, 2001.

[12] G. Leduc. Failure-based Congruences, Unfair Divergences and New Testing Theory. In S. T. Vuong and S. T. Chanson, editors, *PSTV*, volume 1 of *IFIP Conference Proceedings*. Chapman & Hall, 1994.

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

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

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

[16] G. Smith. A Fully Abstract Semantics of Classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[17] G. Smith. A semantic integration of object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313, pages 62–81. Springer-Verlag, 1997.

[18] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[19] J. M. Spivey. *The Z notation: A reference manual*. Prentice Hall, 1989.

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

[21] R. J. van Glabbeek. Full Abstraction in Structural Operational Semantics (extended abstract). In *Algebraic Methodology and Software Technology*, pages 75–82, 1993.

[22] R. J. van Glabbeek. The Linear Time - Branching Time Spectrum II. In *International Conference on Concurrency Theory*, pages 66–81, 1993.

[23] G. Winskel and M. Nielsen. Models for concurrency. Technical Report DAIMI PB 429, Computer Science Dept. Aarhus Universty, 1992.

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

A Proofs

$\mathsf{A}[\![\rho_e]\!]_R \stackrel{\text{def}}{=} \{\langle \rho_{in}, \rho_{out} \mid_n \rangle \mid \langle \rho_{in}, \rho_{out} \rangle \in \beta_{io}(\rho_e) \wedge \rho \mid_n \in Tr^c(\mathsf{A})\}$

Lemma 4 $\forall \rho_{ex} \cdot \mathsf{A}[\![\rho_{ex}]\!]_R \supseteq \mathsf{C}_Z[\![\rho_{ex}]\!]_R \Leftrightarrow \forall \rho_n \cdot \mathsf{A}[\![\rho_n]\!]_R \supseteq \mathsf{C}_Z[\![\rho_n]\!]_R$

Proof Each ρ_e has a unique underlying ρ_n and clearly $\mathsf{A}[\![\rho_e]\!]_R \subseteq_{\mathsf{A}} [\![\rho_n]\!]_R$. But we can see that:

$dom(\mathsf{A}[\![\rho_e]\!]_R) \subseteq dom(\mathsf{A}[\![\rho_n]\!]_R)$

The relation formed by $\mathsf{A}[\![\rho_n]\!]_R$ restricted to $dom(\mathsf{A}[\![\rho_e]\!]_R)$ is identical to

$\mathsf{A}[\![\rho_e]\!]_R$ from which we can conclude our result. •

$\mathsf{A} \sqsubseteq_{DT} \mathsf{C} \stackrel{\text{def}}{=} \forall_{[\_]_a \in [\_]} . Obs([\mathsf{C}]_a) \subseteq Obs([\mathsf{A}]_a)$.

Lemma 5 $\mathsf{A} \sqsubseteq_{DT} \mathsf{C} \Leftrightarrow \mathsf{A} \sqsubseteq_R \mathsf{C}$

Proof 1. $\mathsf{A} \sqsubseteq_{DT} \mathsf{C} \Rightarrow \mathsf{A} \sqsubseteq_R \mathsf{C}$

$\mathsf{A} \sqsubseteq_{DT} \mathsf{C} \Leftrightarrow \forall_{[\_]_a \in [\_]} . Obs([\mathsf{C}]_a) \subseteq Obs([\mathsf{A}]_a)$

$\forall_{\rho_e} . r \in Obs([\mathsf{C}]_{\rho_e}) \Rightarrow r \in Obs([\mathsf{A}]_{\rho_e})$

$\forall_{\rho_e} . r \in Tr^c([\mathsf{C}]_{\rho_e}) \Rightarrow r \in Tr^c([\mathsf{A}]_{\rho_e})$ [1]

Case $|\rho_e| = |r|$ then $r = \rho$

$\rho \in Tr^c([\mathsf{C}]_{\rho_e}) \Leftrightarrow \langle \rho_{in}, \rho_{out} \rangle \in \mathsf{C}_Z[\![\rho_e]\!]_R$

Case $|\rho_e| > |r|$ then $r = \rho \mid_{|r|}$

$\rho \mid_{|r|} \in Tr^c([\mathsf{C}]_{\rho_e}) \Leftrightarrow \langle \rho_{in}, \rho_{out} \mid_{|r|} \rangle \in \mathsf{C}_Z[\![\rho_e]\!]_R$. [2]

From [1] and [2] $\forall \rho_n . x \in \mathsf{C}_Z[\![\rho_e]\!]_R \Rightarrow x \in \mathsf{A}_Z[\![\rho_e]\!]_R$

Hence $\forall \rho_e . \mathsf{C}_Z[\![\rho_e]\!]_R \subseteq \mathsf{A}_Z[\![\rho_e]\!]_R$

From Lemma 4: $\forall \rho_n . \mathsf{C}_Z[\![\rho_n]\!]_R \subseteq \mathsf{A}_Z[\![\rho_n]\!]_R \stackrel{\text{def}}{=} \mathsf{A}_Z \sqsubseteq_R \mathsf{C}_Z$

2. $\mathsf{A} \sqsubseteq_{DT} \mathsf{C} \Leftarrow \mathsf{A} \sqsubseteq_R \mathsf{C}$ reverse above. •

Lemma 6 $\mathsf{A} \sqsubseteq_{DT} \mathsf{C} \Leftrightarrow \mathsf{A} \sqsubseteq_g \mathsf{C}$

Proof

1. $\forall_{[\_]_a \in [\_]} . Obs([\mathsf{C}]_a) \subseteq Obs([\mathsf{A}]_a) \Leftarrow \mathsf{A} \sqsubseteq_g \mathsf{C}$:

Let $[\_]_a \stackrel{\text{def}}{=} (\_ \parallel_{\alpha(C)} \rho_{in})\delta_{Act}$ and $\widehat{\rho_{in}} \stackrel{\text{def}}{=} \rho_{in} \restriction_{\overline{\alpha(\mathsf{C})}}$.

For $X \in \overline{lts}$ and $\mathsf{C} \in lts$ then from the construction of $[\mathsf{C}]_X = (\mathsf{C} \parallel_{\alpha(\mathsf{C})} X)\delta_{Act}$ we can see that:

$\sigma \in Tr^c([\mathsf{C}]_X) \Leftrightarrow \sigma \in Tr^c(X) \wedge \overline{\sigma} \restriction_{\alpha(\mathsf{C})} \in Tr^c(\mathsf{C})$ [1]

¿From $\mathsf{A} \sqsubseteq_g \mathsf{C}$ we have $o \in Tr_c(\mathsf{C}) \Rightarrow o \in Tr_c(\mathsf{A})$. [2]

From [1] and [2] we have

$\sigma \in Tr^c([\mathsf{C}]_X) \Rightarrow \sigma \in Tr^c(X) \wedge \overline{\sigma} \restriction_{\alpha(\mathsf{A})} \in Tr^c(\mathsf{A})$

hence $o \in Tr_c([\mathsf{C}]_a) \Rightarrow o \in Tr_c([\mathsf{A}]_a)$ from [1]

hence $Obs([\mathsf{C}]_a) \subseteq Obs([\mathsf{A}]_a)$

2. $\mathsf{A} \sqsubseteq_{DT} \mathsf{C} \Rightarrow \mathsf{A} \sqsubseteq_g \mathsf{C}$:

$\langle \rho_{in}, \{a\} \rangle \in nsF(\mathsf{C}) \Leftrightarrow \forall_{x:io} \overline{\rho_{in}} \in Obs([\mathsf{C}]_{\overline{\rho_{in}a x}})$

As $\forall_{x:io} \overline{\rho_{in}} \in Obs([\mathsf{C}]_{\overline{\rho_{in}a x}}) \Rightarrow \forall_{x:io} \overline{\rho_{in}} \in Obs([\mathsf{A}]_{\overline{\rho_{in}a x}})$

then $nsF(\mathsf{C}) \subseteq nsF(\mathsf{A})$. •

Lemma 7 $I([\mathsf{C}]_a) > I([\mathsf{A}]_a) \Rightarrow Obs([\mathsf{C}]_a) \subseteq Obs([\mathsf{A}]_a)$.

Proof From the definition of $\gtrsim$ see Figure 5 the only counter example to the lemma could occur when $I([\mathsf{A}]_\rho) = \{\bot\} \wedge I([\mathsf{C}]_\rho) \neq \{\bot\}$. [1]

So we will show that this part of the relation $\gtrsim$ is empty. We do this by giving a construction from a context for which the relation holds and the constraint [1] is satisfied to a context for which the relation fails to hold.

Assume $I([\mathsf{A}]_\rho) = \{\bot\}$ then $\rho \notin Obs([\mathsf{A}]_\rho)$ and as $I([\mathsf{C}]_\rho) \neq \{\bot\}$ then $\rho \in Obs([\mathsf{C}]_\rho)$. Hence $Obs([\mathsf{C}]_\rho) \not\subseteq Obs([\mathsf{A}]_\rho)$ i.e. $Tr^c([\mathsf{C}]_\rho) \not\subseteq Tr^c([\mathsf{A}]_\rho)$. Select an a such that $\rho a \notin Tr^c([\mathsf{C}]_{\rho a})$ hence $Tr^c([\mathsf{C}]_{\rho a}) = Tr^c([\mathsf{C}]_\rho)$. As $I([\mathsf{A}]_{\rho a}) = \{\bot\} = I([\mathsf{C}]_{\rho a})$ and $Obs([\mathsf{A}]_{\rho a}) \not\supseteq Obs([\mathsf{C}]_{\rho a})$ we have $Obs([\mathsf{A}]_{\rho a}) \not\gtrsim Obs([\mathsf{C}]_{\rho a})$.

If $I([\mathsf{A}]_\rho) = \{\bot\} \wedge I([\mathsf{C}]_\rho) \neq \{\bot\}$ $I([\mathsf{C}]_\rho) \xrightarrow{\gtrsim} I([\mathsf{A}]_\rho)$ then $Obs([\mathsf{A}]_{\rho a}) \not\gtrsim Obs([\mathsf{C}]_{\rho a})$ •

Lemma 8 $\mathsf{A} \sqsubseteq_{Test} \mathsf{C} \Leftrightarrow \mathsf{A} \sqsubseteq_{DT} \mathsf{C}$.

Proof

1.$\mathsf{A} \sqsubseteq_{Test} \mathsf{C} \Rightarrow \mathsf{A} \sqsubseteq_{DT} \mathsf{C}$

By definition if $Obs([\mathsf{C}]_a) \gtrsim Obs([\mathsf{A}]_a)$ then $I([\mathsf{C}]_a) > I([\mathsf{A}]_a) \vee (I([\mathsf{A}]_a) = I([\mathsf{C}]_a) \wedge Obs([\mathsf{A}]_a) \supseteq Obs([\mathsf{C}]_a))$. As if $I([\mathsf{C}]_a) > I([\mathsf{A}]_a)$ then from Lemma 7 $Obs([\mathsf{C}]_a) \subseteq Obs([\mathsf{A}]_a)$.

Hence $Obs([\mathsf{C}]_a) \gtrsim Obs([\mathsf{A}]_a) \Rightarrow Obs([\mathsf{C}]_a) \subseteq Obs([\mathsf{A}]_a)$.

$\forall_{[\_]_a \in [\_]^A} . Obs([\mathsf{C}]_a) \gtrsim Obs([\mathsf{A}]_a) \Rightarrow Obs([\mathsf{C}]_a) \subseteq Obs([\mathsf{A}]_a)$

2. Similarly $Obs([\mathsf{C}]_a) \subseteq Obs([\mathsf{A}]_a) \Rightarrow Obs([\mathsf{C}]_a) \gtrsim Obs([\mathsf{A}]_a)$

$\mathsf{A} \sqsubseteq_{Test} \mathsf{C} \Leftarrow \mathsf{A} \sqsubseteq_{DT} \mathsf{C}$

From 1. and 2.

$\mathsf{A} \sqsubseteq_{Test} \mathsf{C} \Leftrightarrow \mathsf{A} \sqsubseteq_{DT} \mathsf{C}$. •