Programming Research Group

A SINGLETON FAILURES SEMANTICS FOR COMMUNICATING SEQUENTIAL PROCESSES

Christie Bolton, Jim Davies

PRG-RR-01-11

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

This paper defines a new denotational semantics for the algebraic specification language of Communicating Sequential Processes (CSP). The semantics lies between the existing traces and failures models of CSP, providing a treatment of nondeterminism in terms of singleton failures. Although the semantics does not represent a congruence upon the full language, it is adequate for sequential tests of nondeterministic processes.

This semantics corresponds exactly to the relational semantics used for data refinement in Z: an abstract data type is refined in the relational semantics precisely when the corresponding process is refined in the singleton failures semantics. Proofs about data types can therefore be conducted in either semantic domain.

1 Introduction

In this paper, we define the *singleton failures* model, an alternative denotational semantic model for the language of Communicating Sequential Processes (CSP) [Hoa85]. We motivate the need for such a model by presenting a translation from *abstract data types* expressed in the state-based specification language Z [Spi92] to their corresponding CSP processes. We prove that the relational refinement of such data types does not coincide with any of the established refinement orderings on CSP; data refinement [HHS87, WD96] corresponds precisely with the refinement ordering induced by our singleton failures semantic model.

Data types are given a sequential relational semantics; for a given sequence of inputs, or operation calls, the semantics tells us which sequences of outputs might appear. There is a clear link between this relational semantics and the behavioural semantics of processes. We may combine sequences of inputs and outputs to produce trace information, and—by considering the points at which errors occur and are propagated—obtain information about failures or divergences. However we prove that the information content of the relational semantics is *strictly less* than that of the failures model [Ros98]. The sequential nature of the semantics means that whilst we have information about what operations may *individually* be refused after a given sequence of interaction we do not have information about what *combinations* of operations may be refused. The singleton failures semantic model is consistent with the relational semantics of data types; refusal information is recorded separately for each event.

The singleton failures semantic model lies between two existing models: the finite traces model T, and the stable failures model F [Ros98]. It distinguishes between pairs of processes that would be identified in T, yet identifies others that would be distinguished in F. Although the semantics does not represent a congruence upon the full language we show that it is adequate for sequential tests of nondeterministic processes and is an ideal basis for an investigation of data refinement.

The paper begins with a brief review of the theory of CSP: the process language, and the existing semantic models. This is followed in Section 3 by a explanation of the new semantic model, the *Singleton Failures Model*. We present axioms and laws for this model, and demonstrate that it is not a congruence for the language. We explain how the singleton failures refinement may easily be verified using the model checker FDR [For97] and conclude that this model may be employed to verify automatically the relational refinement of data types.

In Section 4 we formally define our data types and give them a blocking semantics [BD99]: operations may not be called outside their preconditions. We present a translation from our abstract data types to their process equivalents and prove that data refinement coincides exactly with singleton failures refinement. We then go on, in Section 5 to explore the similarities and differences between our work and other links between state-based and behavioural formalisms. We conclude the paper with a discussion of the use and applicability of our proposed semantic model.

2 Notation

A *process*, as defined in [Hoa85], is a pattern of communication. We may use processes to represent components in terms of their communicating behaviour, building up descriptions using the standard operators of the CSP language.

Processes themselves are defined in terms of *events*: synchronous, atomic communications between a process and its environment. Compound events may be constructed using '.' the dot operator; a family of compound events is called a *channel*, and may be used to represent the passing of a value between components.

2.1 Syntactic definitions

Processes may be defined by sets of mutually-recursive equations; these equations may be indexed to allow parameterised definitions. Parameters may be used to represent aspects of the process state, and may appear in *guards*: we write $bool \;\&\; P$ to denote the process that behaves as P if Boolean *bool* is true, and can perform no events otherwise.

The atomic processes *Skip* and *Stop* both denote termination; *Skip* denotes successful termination—the end of a pattern of communication—whereas *Stop* denotes deadlock. The most basic operator is the *prefix operator*; if P is a process and a is an event then $a \rightarrow P$ denotes a process that is initially ready to engage in a. If this event occurs it will subsequently behave as process P.

There are two types of *choice operators* denoting internal choice and external choice. An *internal choice* is resolved between the processes involved without reference to the environment whereas an *external choice* may be influenced by the environment and is resolved by the first event to occur. These operators each come in two forms: the binary operator; and the choice over an indexed set of processes. Given processes P and Q and a set of processes $R(i)$ indexed over some set I, the following processes are as described:

$P \sqcap Q$	an internal choice between P and Q;
$P \,\Box\, Q$	an external choice between P and Q;
$\sqcap\, i : I \bullet R(i)$	an internal choice over the indexed set of processes;
$\Box\, i : I \bullet R(i)$	an external choice over the indexed set of processes.

We may compose processes *sequentially*. If P and Q are both processes, then the process $P \,{}_{9}^{\circ}\, Q$ first behaves as P and then, if P successfully terminates, behaves as Q. Alternatively we may compose processes in *parallel*. We write $P \;{}_X\!\|_Y\; Q$ to denote the process in which P and Q can evolve independently but must synchronise on every event in the set $X \cap Y$. In an *interleaving* parallel combination, no synchronisation is required; in the combination $P \;|||\; Q$, the two processes evolve independently. As with ordinary parallel combination, termination requires the agreement of both parties.

Finally, we introduce the *hiding operator*: if P is a process and A is a set of events then $P \setminus A$ is a process that behaves as P, except that both the requirement to synchronise upon and the ability to observe events from the set A has been removed.

2.2 Established semantic models

In this paper we are concerned with two established semantic models: the traces model (T) and the stable failures model (F) as defined in [Ros98]. Both the traces model and the stable failures model record all possible sequences of interaction. The difference between these models is that the traces model does not provide an adequate treatment of non-determinism. A third model presented by Roscoe [Ros98] is the failures-divergences model (D). The failures-divergences model not only records all possible sequences of interaction and provides an adequate treatment of non-determinism; it also identifies potential *livelock*, the possibility of the process entering into an infinite sequence of internal actions. Livelock, or *divergence* is generally introduced by hiding. Since one of the restrictions we impose in this paper is that hiding may not be performed if so doing might cause the system to diverge we will not concern ourselves with this model.

A *trace* records a history of communication between a process and the environment. It is expressed as a sequence of events in which the process has engaged up to a particular point in time with the head of the sequence being the first event that was communicated. We write $T\llbracket P \rrbracket$ to denote the set of all finite traces of a process P. Given processes P and Q, event a and sets of events X and Y, the denotational semantics for the basic CSP operators within the traces model are as follows:

$$
\begin{aligned}
T\llbracket Stop \rrbracket &= \{\langle\rangle\} \\
T\llbracket a \rightarrow P \rrbracket &= \{\langle\rangle\} \cup \{ s : \operatorname{seq} Event \mid s \in T\llbracket P \rrbracket \bullet \langle a \rangle \frown s \} \\
T\llbracket P \mathbin{\Box} Q \rrbracket &= T\llbracket P \rrbracket \cup T\llbracket Q \rrbracket \\
T\llbracket P \sqcap Q \rrbracket &= T\llbracket P \rrbracket \cup T\llbracket Q \rrbracket \\
T\llbracket P \;{}_X\|_Y\; Q \rrbracket &= \{ tr : \operatorname{seq} Event \mid \\
&\qquad \operatorname{ran} tr \subseteq X \cup Y \;\wedge\; tr \upharpoonright X \in T\llbracket P \rrbracket \;\wedge\; tr \upharpoonright Y \in T\llbracket Q \rrbracket \} \\
T\llbracket P \setminus X \rrbracket &= \{ tr : \operatorname{seq} Event \mid tr \in T\llbracket P \rrbracket \bullet tr \setminus X \}
\end{aligned}
$$

A *failure* is a pair in which the first element is the trace of a process and the second is a refusal set—a set of events from which the process may refuse to communicate anything no matter how long it is offered—of the process after the given trace. Given processes P and Q, event a and sets of events X and Y, the denotational semantics for the basic CSP operators within the stable failures model are defined as follows:

$$
\begin{aligned}
F\llbracket Stop \rrbracket &= \{ ref : \mathbb{P}\, Event \bullet (\langle\rangle, ref) \} \\
F\llbracket a \rightarrow P \rrbracket &= \{ ref : \mathbb{P}\, Event \mid a \notin ref \bullet (\langle\rangle, ref) \} \\
&\quad \cup \\
&\quad \{ tr : \operatorname{seq} Event;\; ref : \mathbb{P}\, Event \mid \\
&\qquad (tr, ref) \in F\llbracket P \rrbracket \bullet (\langle a \rangle \frown tr, ref) \} \\
F\llbracket P \mathbin{\Box} Q \rrbracket &= (F\llbracket P \rrbracket \cap F\llbracket Q \rrbracket) \cap \{ tr : \operatorname{seq} Event;\; ref : \mathbb{P}\, Event \mid tr = \langle\rangle \} \\
&\quad \cup \\
&\quad (F\llbracket P \rrbracket \cup F\llbracket Q \rrbracket) \cap \{ tr : \operatorname{seq} Event;\; ref : \mathbb{P}\, Event \mid tr \neq \langle\rangle \}
\end{aligned}
$$

$$
\begin{aligned}
F\,[\![P \sqcap Q]\!] &= F\,[\![P]\!] \,\cup\, F\,[\![Q]\!] \\
F\,[\![P\ {}_X\|_Y\ Q]\!] &= \{tr : \mathrm{seq}(X \cup Y);\ ref_P, ref_Q : \mathbb{P}\,Event \mid \\
&\qquad\qquad ref_P \setminus (X \cap Y) \;=\; ref_Q \setminus (X \cap Y) \\
&\qquad\qquad \wedge \\
&\qquad\qquad (tr \restriction X, ref_P) \in F\,[\![P]\!] \;\wedge\; (tr \restriction Y, ref_Q) \in F\,[\![Q]\!] \;\bullet \\
&\qquad (tr, ref_P \cup ref_Q)\} \\
F\,[\![P \setminus X]\!] &= \{tr : \mathrm{seq}\,Event;\ ref : \mathbb{P}\,Event \mid \\
&\qquad (tr, X \cup ref) \in F\,[\![P]\!] \bullet (tr \setminus X, ref)\}.
\end{aligned}
$$

The refinement orderings associated with the traces model and the stable failures model are both based on reverse containment. Given processes P and Q, P is refined by Q within the traces model precisely when every trace of Q is also a trace of P. Similarly, P is refined by Q within the stable failures model precisely when every failure of Q is also a failure of P.

$$
P \sqsubseteq_T Q \;\Leftrightarrow\; T\,[\![Q]\!] \subseteq T\,[\![P]\!] \qquad\qquad P \sqsubseteq_F Q \;\Leftrightarrow\; F\,[\![Q]\!] \subseteq F\,[\![P]\!]
$$

3 Singleton failures semantic model

In this section we formally define the *singleton failures* semantics model (S), an alternative semantic model for CSP whose information content is greater than that of the traces model but less than that of the stable failures model. We record information about possible sequences of interaction and record which events might be refused after each such sequence. However the difference between this model and the stable failures model is that we will throw out all information about the availability after each sequence of interaction of *combinations* of operations.

3.1 Refinement

Like the traces model and the stable failures model, refinement within the singleton failures model is based on reverse-containment: one process is refined by another within the singleton failures model precisely when all the singleton failures of the second are also singleton failures of the first. More formally, given process P and Q:

$$
P \sqsubseteq_S Q \quad\Leftrightarrow\quad S\,[\![Q]\!] \subseteq S\,[\![P]\!].
$$

We say that two processes are equivalent under the singleton failures model when each refines the other.

$$
P \equiv_S Q \quad\Leftrightarrow\quad P \sqsubseteq_S Q \;\wedge\; Q \sqsubseteq_S P
$$

3.2 Denotational semantics

The singleton failures semantics is essentially a projection of the stable failures semantics. Each process is associated with a set of trace-refusal pairs in which the cardinality of the refusal sets is at most one.

3.2.1 Basic operators

We define the singleton failures semantics for the process *Stop*, the prefix operator and the internal and external choice operations as follows:

$$
\begin{aligned}
S\,[\![Stop]\!] \;&=\; \{\, ref : \mathbb{P}\,Event \;\mid\; \#ref \le 1 \;\bullet\; (\langle\rangle, ref)\,\} \\[1ex]
S\,[\![a \to P]\!] \;&=\; \{\, ref : \mathbb{P}\,Event \;\mid\; \#ref \le 1 \land ref \ne \{a\} \;\bullet\; (\langle\rangle, ref)\,\} \\
&\quad\; \cup \\
&\quad\; \{\, s : \mathrm{seq}\,Event;\; ref : \mathbb{P}\,Event \mid \\
&\qquad\quad (s, ref) \in S\,[\![P]\!] \;\bullet\; (\langle a\rangle \frown s, ref)\,\} \\[1ex]
S\,[\![P \,\Box\, Q]\!] \;&=\; (\,S\,[\![P]\!] \;\cap\; S\,[\![Q]\!]\,) \;\cap\; \{s : \mathrm{seq}\,Event;\; ref : \mathbb{P}\,Event \;\mid\; s = \langle\rangle\,\} \\
&\quad\; \cup \\
&\quad\; (\,S\,[\![P]\!] \;\cup\; S\,[\![Q]\!]\,) \;\cap\; \{s : \mathrm{seq}\,Event;\; ref : \mathbb{P}\,Event \;\mid\; s \ne \langle\rangle\,\} \\[1ex]
S\,[\![P \sqcap Q]\!] \;&=\; S\,[\![P]\!] \cup S\,[\![Q]\!]
\end{aligned}
$$

Laws We see that for the process *Stop*, the prefix operator and the internal and external choice operators our singleton failures semantic model is a projection of the stable failures semantic model. We record only those trace-refusal pairs in which the cardinality of the refusal set is at most one. Defining the projection function *slice* which takes sets of pairs in which the first element of each pair is a sequence of events and the second is a set of events and returns the set containing those pairs in which the cardinality of the second element is at most one as follows:

$$
\begin{array}{|l}
slice : \mathbb{P}(\mathrm{seq}\,Event \times \mathbb{P}\,Event) \to \mathbb{P}(\mathrm{seq}\,Event \times \mathbb{P}\,Event) \\
\hline
\forall\, failures : \mathbb{P}(\mathrm{seq}\,Event \times \mathbb{P}\,Event) \;\bullet \\
\qquad slice\,(\,failures\,) \;=\; failures \;\cap\; \{\, s : \mathrm{seq}\,Event;\; X : \mathbb{P}\,Event \mid \#X \le 1\,\}
\end{array}
$$

we see clearly the relation between the singleton failures semantic definitions and the stable failures definitions of the basic operators.

$$
\begin{aligned}
S\,[\![Stop]\!] \;&=\; slice\,(\,F\,[\![Stop]\!]\,) \\
S\,[\![a \to P]\!] \;&=\; slice\,(\,F\,[\![a \to P]\!]\,) \\
S\,[\![P \,\Box\, Q]\!] \;&=\; slice\,(\,F\,[\![P \,\Box\, Q]\!]\,) \\
S\,[\![P \sqcap Q]\!] \;&=\; slice\,(\,F\,[\![P \sqcap Q]\!]\,)
\end{aligned}
$$

It follows immediately from this observation that the laws for combinations of these operators that hold within the stable failures framework will automatically hold within the singleton failures model.

$$\begin{array}{rcll}
Stop \,\Box\, P & \equiv_S & P & [\,\Box - \mathrm{unit}\,] \\
P \,\Box\, P & \equiv_S & P & [\,\Box - \mathrm{idem}\,] \\
P \sqcap P & \equiv_S & P & [\,\sqcap - \mathrm{idem}\,] \\
P \,\Box\, Q & \equiv_S & Q \,\Box\, P & [\,\Box - \mathrm{sym}\,] \\
P \sqcap P & \equiv_S & Q \,\Box\, P & [\,\sqcap - \mathrm{sym}\,] \\
P \,\Box\, (Q \,\Box\, R) & \equiv_S & (P \,\Box\, Q) \,\Box\, R & [\,\Box - \mathrm{assoc}\,] \\
P \sqcap (Q \sqcap R) & \equiv_S & (P \sqcap Q) \sqcap R & [\,\sqcap - \mathrm{assoc}\,] \\
P \,\Box\, (Q \sqcap R) & \equiv_S & (P \,\Box\, Q) \sqcap (P \,\Box\, R) & [\,\Box - \mathrm{dist}\,] \\
P \sqcap (Q \,\Box\, R) & \equiv_S & (P \sqcap Q) \,\Box\, (P \sqcap R) & [\,\sqcap - \mathrm{dist}\,]
\end{array}$$

Consider for instance Lemma 1 below.

Lemma 1 $[\,\Box - \mathrm{dist}\,]$

The external choice operator is distributive with respect to the internal choice operator within the singleton failures semantic model. More formally, for all processes P, Q and R the following property must be true.

$$P \,\Box\, (Q \sqcap R) \;\; \equiv_S \;\; (P \,\Box\, Q) \sqcap (P \,\Box\, R)$$

Proof

Given processes P, Q and R we reason as follows:

$S\,[\![\, P \,\Box\, (\, Q \sqcap R \,)\,]\!]$

$\equiv_S$ *[property of $S\,[\![P \,\Box\, Q]\!]$]*

$slice\,(\,F\,[\![\, P \,\Box\, (\, Q \sqcap R \,)\,]\!]\,)$

$\equiv_S$ *[distributivity of $\Box$ over $\sqcap$ within F]*

$slice\,(\,F\,[\![\,(\, P \,\Box\, Q \,) \sqcap (\, P \,\Box\, R \,)\,]\!]\,)$

$\equiv_S$ *[definition of $F\,[\![\, P \sqcap Q \,]\!]$]*

$slice\,(\,F\,[\![\,(\, P \,\Box\, Q \,)\,]\!] \cup F\,[\![\,(\, P \,\Box\, R \,)\,]\!]\,)$

$\equiv_S$ *[distributivity]*

$slice\,(\,F\,[\![\,(\, P \,\Box\, Q \,)\,]\!]\,) \cup slice\,(\,F\,[\![\,(\, P \,\Box\, R \,)\,]\!]\,)$

$\equiv_S$ *[property of $S\,[\![\, P \,\Box\, Q \,]\!]$]*

$S\,[\![\,(\, P \,\Box\, Q \,)\,]\!] \cup S\,[\![\,(\, P \,\Box\, R \,)\,]\!]$

$\equiv_S$ *[property of $S\,[\![\, P \sqcap Q \,]\!]$]*

$S\,[\![\,(\, P \,\Box\, Q \,) \sqcap (\, P \,\Box\, R \,)\,]\!]$

In Lemma 1 above we proved that the external choice operator distributes over the internal choice operator within the singleton failures semantics model. This result followed directly from the corresponding property within the stable failures model and the fact that we may obtain the singleton failures semantic definitions for processes $P \sqcap Q$ and $P \;\Box\; Q$ by applying a projection function to the stable failures semantic definitions for these processes. We may apply similar arguments to prove that the other laws involving only the process *Stop*, the prefix operator and the internal and external choice operators are preserved within the singleton failures semantic framework.

3.2.2 Parallel combination

The definition of the parallel combination of two processes within the singleton failures semantics is as we would expect. Given processes P and Q defined respectively over alphabets X and Y, we define the effect of the parallel operator as follows:

$$
\begin{array}{rcl}
S\,[\![P \;{}_X\|_Y\; Q]\!] & = & \{tr : \mathrm{seq}\,\mathit{Event};\; \mathit{ref} : \mathbb{P}\,\mathit{Event} \mid \\
& & \quad \mathrm{ran}\, tr \subseteq X \cup Y \\
& & \quad \wedge \\
& & \quad \exists\, \mathit{ref}_P, \mathit{ref}_Q : \mathbb{P}\,\mathit{Event} \bullet \\
& & \qquad \mathit{ref} \cap (X \cup Y) = (\mathit{ref}_P \cap X) \cup (\mathit{ref}_Q \cap Y) \\
& & \qquad \wedge \\
& & \qquad (tr \upharpoonright X, \mathit{ref}_P) \in S\,[\![P]\!] \;\wedge\; (tr \upharpoonright Y, \mathit{ref}_Q) \in S\,[\![Q]\!]\,\}.
\end{array}
$$

All events in any possible trace must lie either in X, the alphabet of P, or in Y, the alphabet of Q. Furthermore, after any given trace a set of events can only be refused if those elements that are also in the set X could be refused by process P after the same sequence of events restricted to those in X and if those elements that are also in the set Y could be refused by process Q after the same sequence of events restricted to those in Y.

Laws We see that our definition of the singleton failures semantics for the parallel combination of processes employs the same projection function on the stable failures semantic model.

$$S\,[\![P \;{}_X\|_Y\; Q]\!] \;=\; \mathit{slice}\,(F\,[\![P \;{}_X\|_Y\; Q]\!])$$

Hence, once more, we inherit various laws from the stable failures model.

$$
\begin{array}{rcll}
P \;{}_X\|_Y\; Q & \equiv_S & Q \;{}_X\|_Y\; P & [{}_X\|_Y - \mathrm{sym}] \\
(P \;{}_X\|_Y\; Q) \;{}_{X\cup Y}\|_Z\; R & \equiv_S & P \;{}_X\|_{Y\cup Z}\; (Q \;{}_Y\|_Z\; R) & [{}_X\|_Y - \mathrm{assoc}] \\
P \;{}_X\|_Y\; (Q \sqcap R) & \equiv_S & (P \;{}_X\|_Y\; Q) \sqcap (P \;{}_X\|_Y\; R) & [{}_X\|_Y - \mathrm{dist}]
\end{array}
$$

3.2.3 Hiding

For all the operators we have considered thus far we may obtain the singleton failures semantic definition of the effect of the operator by taking a projection of the stable failures semantic definition of the effect of that operator. In this section we show that we are unable to apply this projection function to the definition of the hiding operator within the stable failures model and still retain a congruence.

Recall the definition of hiding within the stable failures semantic model; given process P and set of events X, the set $F\,[\![P \setminus X]\!]$ is equivalent to

$$\{\, tr : \operatorname{seq} Event;\; ref : \mathbb{P}\, Event \mid (tr, X \cup ref) \in F\,[\![P]\!] \bullet (tr \setminus X, ref) \,\}.$$

Now consider the set obtained by applying the projection function *slice* to this set.

$$\begin{aligned} slice\,(\,F\,[\![P \setminus X]\!]\,) \;=\; \{\, & tr : \operatorname{seq} Event;\; ref : \mathbb{P}\, Event \mid \\ & \#ref \leq 1 \wedge (tr, X \cup ref) \in F\,[\![P]\!] \bullet \\ & (tr \setminus X, ref) \,\} \end{aligned}$$

We see that this is not expressible in terms of the singleton failures semantics since the set $X \cup ref$ may have at least two elements and we do not preserve information about the availability of combinations of events. If we were to simply use the projection function that we used for all other operators then we would not have a congruence: given a process P and a set of events X we would not be able to express $S\,[\![P \setminus X]\!]$ in terms of $S\,[\![P]\!]$ and X.

We must therefore propose an alternative definition for hiding within our singleton failures model: this definition will not conform to our projection function but will be expressible in terms of the singleton failures semantics.

We define the singleton failures semantics for the hiding operator as follows:

$$\begin{aligned} S\,[\![\, P \setminus X \,]\!] \;=\; \{\, & tr : \operatorname{seq} Event;\; ref : \mathbb{P}\, Event \mid \\ & \quad \#ref \leq 1 \\ & \quad \wedge \\ & \quad \forall\, Y : \mathbb{P}\,(X \cup ref) \mid \#Y \leq 1 \bullet (tr, Y) \in S\,[\![P]\!] \bullet \\ & (tr \setminus X, ref) \,\} \end{aligned}$$

Given a process P and a set of operations X to be hidden, the pair (tr, ref) lies in the singleton failures semantics of the process $P \setminus X$ provided there is some sequence of events s that is a trace of process P such that $tr = s \setminus X$ and such that every element in the union of X and ref can *individually* be refused by P after s.

Laws Because we have not been able to define our hiding as a projection of the failures semantics of hiding we do not automatically inherit the laws for hiding as we did with the other operators.

It is easily shown that the following laws hold:

$$
\begin{array}{rcll}
(P \setminus X) \setminus Y & \equiv_S & (P \setminus Y) \setminus X & [\,\mathit{hide} - \mathrm{sym}\,] \\
(P \setminus X) \setminus Y & \equiv_S & P \setminus (X \cup Y) & [\,\mathit{hide} - \mathrm{combine}\,] \\
P \setminus \emptyset & \equiv_S & P & [\,\mathrm{null\ hiding}\,]
\end{array}
$$

however it is equally easily shown that laws of distributivity are not preserved under hiding. Recall that although the hiding operator is not fully distributive over F it is finitely distributive over F. That is, it satisfies the following property:

$$(P \sqcap Q) \setminus X \equiv_F (P \setminus X) \sqcap (Q \setminus X)$$

where $\equiv_F$ denotes equivalence under the stable failures model. Recall further that this result implies *monotonicity*.

Unfortunately, not only is it the case that hiding is not fully distributive over S; it is also the case that it is not finitely distributive over S. The weaker law of finite distributivity is not unconditionally satisfied within the singleton failures model.

$$(\, P \sqcap Q \,) \setminus X \quad \not\equiv_S \quad (\, P \setminus X \,) \ \sqcap \ (\, Q \setminus X \,)$$

We prove this in Observation 1 below.

Observation 1 *The singleton failures semantics does not preserve the law of finite distributivity.*

Proof

Let processes P and Q be defined as follows:

$$
\begin{array}{rcl}
P & = & a \rightarrow Q \ \sqcap \ (\, (\, a \rightarrow Q \,) \ \Box \ Q \,) \\
Q & = & e \rightarrow \mathit{Stop}
\end{array}
$$

and consider the processes $(P \setminus a) \sqcap (Q \setminus a)$ and $(P \sqcap Q) \setminus a$. We can establish easily their associated singleton failures.

$$
\begin{array}{rcl}
S\,[\![\, (P \setminus a) \sqcap (Q \setminus a) \,]\!] & = & \{\, (\langle\rangle, \emptyset), (\langle\rangle, \{a\}) \,\} \\
 & & \cup \\
 & & \{\, (\langle e \rangle, \emptyset), (\langle e \rangle, \{a\}), (\langle e \rangle, \{e\}) \,\} \\
 & & \\
S\,[\![\, (P \sqcap Q) \setminus a \,]\!] & = & \{ (\langle\rangle, \emptyset), (\langle\rangle, \{a\}), (\langle\rangle, \{e\}) \,\} \\
 & & \cup \\
 & & \{\, (\langle e \rangle, \emptyset), (\langle e \rangle, \{a\}), (\langle e \rangle, \{e\}) \,\}
\end{array}
$$

We see that

$$S\,[\![\, (P \sqcap Q) \setminus A \,]\!] \ = \ S\,[\![\, (P \setminus A) \sqcap (Q \setminus A) \,]\!] \ - \ \{\, (\langle\rangle, \{e\}) \,\}$$

and conclude that the singleton failures semantics is not finitely distributive.

Healthiness Conditions We saw in Observation 1 above that the set $S\,[\![(P \setminus A) \sqcap (Q \setminus A)]\!]$ was not equal to the set $S\,[\![(P \sqcap Q) \setminus A]\!]$. However, in B we consider further the conditions under which we may hide a set of events whilst preserving finite distributivity and hence monotonicity. We define a parallel operator and a hiding operator on our communicating data types and prove that these operators are consistent with the corresponding CSP operators. Furthermore we identify a simple set of healthiness conditions for hiding sets of operations on our communicating data types. These conditions are expressed in terms of the internal state space and the relations modelling the operations on the data types and are easily verifiable. We prove that these conditions are necessary; if they are satisfied then hiding the corresponding set of events on the corresponding CSP process will preserve monotonicity.

3.3 Automatic verification of refinement

The singleton failures semantics is concerned with the refusal of individual events rather than combinations of events. However we can use the stable failures refinement ordering, and hence the model-checker FDR [For97], to verify singleton failures refinement.

Given two processes *Spec* and *Impl*, *Spec* is refined by *Impl* within the singleton failures semantic model if every trace of *Impl* is also a trace of *Spec* and if whenever *Impl* can refuse an event after a given trace then *Spec* can refuse the same event after the same trace. It does not matter if *Impl* can refuse a *combination* of events after a given trace that cannot be refused after the same trace by *Spec* provided each event can *individually* be refused by *Spec* after the given trace.

We may therefore use the stable failures refinement ordering to determine whether process *Spec* is refined by process *Impl* within the singleton failures semantics if we can construct a process *Spec′* that has the same traces as *Spec* and that, after every possible trace, can refuse *all possible combinations* of events that can be refused individually by *Spec* after the given trace. Given process *Spec*, the process *Spec′* may be characterised as follows:

$$\begin{aligned} F\,[\![\mathit{Spec}']\!] \;=\; \{ & \mathit{tr} : \operatorname{seq} \mathit{Event};\ \mathit{ref} : \mathbb{P}\,\mathit{Event} \mid \\ & \quad \mathit{tr} \in T\,[\![\mathit{Spec}]\!] \;\wedge\; \forall\, e : \mathit{ref} \bullet (\mathit{tr}, \{e\}) \in F\,[\![\mathit{Spec}]\!] \}. \end{aligned}$$

The method of construction of such a process was suggested by Michael Goldsmith and Gavin Lowe: a process with the requisite traces and refusals may be obtained by putting the process *Spec* in parallel with itself sufficiently many times. To be precise the process *Spec′* may be obtained by putting the process *Spec* in parallel with itself n times where n is the least integer such that

$$F\,[\![\underbrace{\mathit{Spec} \parallel \ldots \parallel \mathit{Spec}}_{\text{(n+1) times}}]\!] \;=\; F\,[\![\underbrace{\mathit{Spec} \parallel \ldots \parallel \mathit{Spec}}_{\text{n times}}]\!]$$

In Theorem 1 below we prove that process *Spec* is refined by process *Impl* within the singleton failures semantic model precisely when we there is some strictly positive integer n such that the n-times parallel combination of *Spec* is refined by *Impl* under the stable failures model.

Theorem 1 *Automation of singleton failures refinement checks.*

For any processes Spec and Impl defined over a finite set of events, Event, the following predicate must hold

$$Spec \sqsubseteq_S Impl \quad \Leftrightarrow \quad \exists\, n : \mathbb{Z}^+ \bullet NSpec(n) \sqsubseteq_F Impl,$$

where the process $NSpec(n)$ is defined as follows:

$$\begin{aligned} NSpec(1) &= Spec \\ \forall\, n : \mathbb{Z}^+ \bullet NSpec(n+1) &= Spec \parallel NSpec(n). \end{aligned}$$

Practical observation We conclude this Section with an observation about the practicality of using the model-checker FDR to automatically verify refinement of processes within the singleton failures model.

An upper bound for n, the number of times we must put *Spec* in parallel with itself, is the number of possible events; *at worst* this will be the product of the number of names of operations on the data type and the square of the number of possible input and output values. However, from a practical point of view, if we were to re-define the process $NSpec(n)$ of Theorem 1 as follows:

$$\begin{aligned} NSpec(1) &= Spec \\ \forall\, n : \mathbb{Z}^+ \bullet NSpec(n+1) &= NSpec(n) \parallel NSpec(n). \end{aligned}$$

the actual number of parallel combinations we would need to perform would be *at most* of order $\log_2(\#Name \times \#Value \times \#Value)$, or equivalently $\log_2(\#Name) + 2\log_2(\#Value)$.

4 Communicating data types and their corresponding processes

Data types may be used to give a state-based description of a system. Typically they are modelled as tuples: one component defines the state space of the data type; a second component defines a non-empty set of possible initial states; and a third component identifies a set of operations on the data type and associates each operation with a relation on the state space describing the effect of that operation.

In this paper we are concerned with *communicating data types*, data types that have an explicit notion of input and output. Each operation on such a data type takes an input value and a state from the internal state space and then returns an output value and a state from the internal state space. Given *State*, the type of the internal state space, and *Value*, the set of all possible input and output values, we model the operations on our communicating data types as relations of the type $(State \times Value) \leftrightarrow (State \times Value)$. Then, given *Name*, the set of all possible names of operations, we define our communicating data types generically as follows.

$$
\begin{array}{|l}
\hline
\textit{CDT}[\textit{State}, \textit{Name}, \textit{Value}] \\
\textit{state} : \mathbb{P}\,\textit{State} \\
\textit{init} : \mathbb{P}\,\textit{State} \\
\textit{ops} : \textit{Name} \nrightarrow ((\textit{State} \times \textit{Value}) \leftrightarrow (\textit{State} \times \textit{Value})) \\
\hline
\emptyset \subset \textit{init} \subseteq \textit{state} \\
\textit{ops} \subseteq \textit{Name} \nrightarrow ((\textit{state} \times \textit{Value}) \leftrightarrow (\textit{state} \times \textit{Value})) \\
\hline
\end{array}
$$

We give a formal semantics to our data types, describing them in terms of programs, or sequences of operations, and the visible effects, in terms of inputs and outputs, of those programs. However, rather than move directly from our communicating data types to a semantics that becomes notationally very heavy we present a function *simplify* [WDB00] from communicating data types (CDTs) to simple data types (SDTs) in which all inputs and outputs are *wound* into the state: all inputs are collected at initialisation and all outputs are returned at finalisation.

The wound state space of a simple data type is a triple. The first element of the triple is of type *State* and records the state of the corresponding communicating data type. The second and third elements are sequences of *Value*s and record respectively the inputs that have not yet been consumed and the outputs that have been given thus far.

$$
\begin{array}{|l}
\hline
\textit{SDT}[\textit{State}, \textit{Name}, \textit{Value}] \\
\textit{state} : \mathbb{P}(\textit{State} \times \operatorname{seq} \textit{Value} \times \operatorname{seq} \textit{Value}) \\
\textit{init} : \operatorname{seq} \textit{Value} \leftrightarrow (\textit{State} \times \operatorname{seq} \textit{Value} \times \operatorname{seq} \textit{Value}) \\
\textit{ops} : \textit{Name} \nrightarrow ((\textit{State} \times \operatorname{seq} \textit{Value} \times \operatorname{seq} \textit{Value}) \leftrightarrow \\
\qquad\qquad\qquad\qquad (\textit{State} \times \operatorname{seq} \textit{Value} \times \operatorname{seq} \textit{Value})) \\
\textit{final} : (\textit{State} \times \operatorname{seq} \textit{Value} \times \operatorname{seq} \textit{Value}) \nrightarrow \operatorname{seq} \textit{Value} \\
\hline
\end{array}
$$

The set of operations on the simple data type is the same as the set of operations on the corresponding communicating data type. Moreover, given values *in* and *out* and states a and a' from the state space of a given communicating data type, if the maplet $(a, in) \mapsto (a', out)$ lies in the relation describing an operation n on the given data type, then for any sequences of values *ins* and *outs*, the maplet $(a, \langle in \rangle \frown ins, outs) \mapsto (a', ins, outs \frown \langle out \rangle)$ will lie in the relation describing the operation n on the corresponding simple data type. The first component of the triples has recorded the change in the state space. The second element of the triples has recorded the consuming of input *in* and the last element of the triples has added the output *out* to the sequence of recorded outputs.

At initialisation a sequence of inputs to be consumed is given. This is then recorded as the second element of the wound state; the first element being any state a' from the set of initial states on the corresponding communicating data type and the last element being the empty sequence thereby recording that no outputs have yet been given. The initialisation operation is then a relation from sequences of inputs to the wound state space.

The finalisation operation on a simple data type is a (functional) relation from the wound state space to sequences of values. It returns the third element of the wound state space, the recorded output values.

The function *simplify* takes a communicating data type and returns the corresponding simple data type. It is formally defined as follows:

$$
\begin{array}{l}
[State, Name, Value] \\
simplify : CDT[State, Name, Value] \nrightarrow SDT[State, Name, Value] \\
\hline
simplify = \\
\quad (\lambda\, CDT[State, Name, Value] \bullet \\
\qquad (\mu\, SDT'[State, Name, Value] \mid \\
\qquad\quad state' = state \times \operatorname{seq} Value \times \operatorname{seq} Value \ \wedge \\
\qquad\quad init' = \{ is : \operatorname{seq} Value;\ st' : init \bullet is \mapsto (st', is, \langle\rangle) \} \ \wedge \\
\qquad\quad \forall\, n : \operatorname{dom} ops' \bullet \\
\qquad\qquad ops'\, n = \{ is, os : \operatorname{seq} Value;\ i, o : Value; \\
\qquad\qquad\qquad\qquad\qquad s_1, s_2 : state \mid \\
\qquad\qquad\qquad (s_1, i) \mapsto (s_2, o) \in ops\, n \bullet \\
\qquad\qquad\qquad\quad (s_1, \langle i \rangle \frown is, os) \mapsto \\
\qquad\qquad\qquad\qquad\qquad (s_2, is, os \frown \langle o \rangle) \} \ \wedge \\
\qquad\quad final' = \{ os : \operatorname{seq} Value;\ st : state \bullet \\
\qquad\qquad\qquad (st, \langle\rangle, os) \mapsto os \}))
\end{array}
$$

4.1 What happens when an operation is called outside its domain of definition?

Because operations need not be defined for every state of the data type a question arises: what happens if we reach a state that is outside the domain of the relation describing the next operation? In this paper we adopt a blocking model in which operations are guarded and may not be called outside their preconditions. An alternative approach [WD96] would have been a non-blocking semantic model in which operations may be called outside their pre-conditions but no guarantees are may about the consequences of so doing. These two models are complementary and may be adopted at different stages in the design process.

If an operation is called outside its preconditions we need to record the fact that a problem has arisen. More than that, we need to preserve that information until finalisation. We do this by augmenting the state space with a bottom element $\perp$ and totalising our relations. Letting $State_\perp = State \cup \{\perp\}$ denote the augmented state space, the totalised versions of the relations describing the operations on our data types are then of the type

$$(State_\perp \times \operatorname{seq} Value \times \operatorname{seq} Value) \leftrightarrow (State_\perp \times \operatorname{seq} Value \times \operatorname{seq} Value).$$

We totalise the relations representing operations by consuming the next input but giving no output and by mapping the state component of any element outside the domain

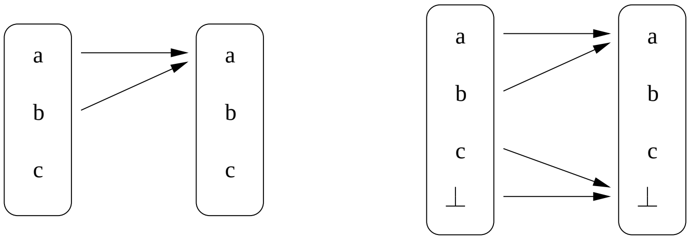

Figure 1: Blocking totalisation: the elements outside the domain are mapped onto the bottom element.

to bottom. Writing $r^{\top}$ to denote the totalised version of relation r, our totalisation of relations corresponding to operations on our simple data types is then defined as follows:

$$
\begin{aligned}
r^{\top} \;=\;\; & r \cup \{\, a : State_{\perp};\ in : Value;\ ins, outs : \mathrm{seq}\, Value \mid \\
& \quad (\, a, \langle in \rangle \frown ins, outs \,) \notin \mathrm{dom}\, r \;\bullet \\
& \quad\quad (\, a, \langle in \rangle \frown ins, outs \,) \mapsto (\, \perp, ins, outs \,) \,\}
\end{aligned}
$$

In Figure 1 we illustrate the effect of the blocking totalisation on the state component of the wound state space.

4.2 Relational semantics

For clarity and brevity we will use the following abbreviations in our next definition. We define *WState* to be the wound state space and overload the operator $_{\perp}$ by defining $WState_{\perp}$ to be the augmented wound state space.

$$
\begin{aligned}
WState \;&==\; State \times \mathrm{seq}\, Value \times \mathrm{seq}\, Value \\
WState_{\perp} \;&==\; State_{\perp} \times \mathrm{seq}\, Value \times \mathrm{seq}\, Value
\end{aligned}
$$

Using these abbreviations we define a function *run* to compose a sequence of named operations. The function *run* takes a program *prog* of operations and a function *ops* describing the effects of each operation. It then returns the relation obtained by composing the *totalised* versions of the relations corresponding to the operations in the program.

$$
\begin{array}{|l}
run : \mathrm{seq}\, Name \rightarrow\!\!\!\!\!\!+ \; (Name \rightarrow\!\!\!\!\!\!+ \; (WState \leftrightarrow WState)) \rightarrow\!\!\!\!\!\!+ \\
\qquad\qquad\qquad\qquad (WState_{\perp} \leftrightarrow WState_{\perp}) \\
\hline
\forall\, n : Name;\ ns : \mathrm{seq}\, Name;\ ops : Name \rightarrow\!\!\!\!\!\!+ \; (WState \leftrightarrow WState) \bullet \\
\quad run\ \langle\rangle\ ops = \mathrm{id}\,[\, WState_{\perp} \,] \\
\quad \wedge \\
\quad run\ (ns \frown \langle n \rangle)\ ops = (run\ ns\ ops) \mathbin{⨾} (ops\ n)^{\top}
\end{array}
$$

To obtain the relational semantics for a simple data type we have only to bracket the result with initialisation and finalisation.

$$
\begin{array}{l}
[State, Name, Value] \\
\hline
R_s : SDT[State, Name, Value] \ \rightarrow \ \mathrm{seq}\, Name \mathrel{+\!\!\!\!\rightarrow} (\mathrm{seq}\, Value \leftrightarrow \mathrm{seq}\, Value) \\
\hline
R_s \ = \ (\, \lambda\, SDT[State, Name, Value] \bullet \\
\qquad\qquad (\, \lambda\, prog : \mathrm{seq}(\mathrm{dom}\, ops) \bullet init \fatsemi run\, prog\, ops \fatsemi final^{\top}\,)\,)
\end{array}
$$

Although *final* is a total function on the wound state space we needed to make it total on the *augmented* wound state space. In the case where the state component is the bottom element it behaves as usual and returns the third component of the state space, the returned outputs.

The relational semantics for communicating data types is then given by the composition of *simplify* and R_s.

$$
\begin{array}{l}
[State, Name, Value] \\
\hline
R : CDT[State, Name, Value] \rightarrow \\
\qquad\quad (\mathrm{seq}\, Name \mathrel{+\!\!\!\!\rightarrow} (\mathrm{seq}\, Value)_{\perp} \times (\mathrm{seq}\, Value)_{\perp}) \\
\hline
R \ = \ simplify \fatsemi R_s
\end{array}
$$

4.3 Data refinement and simulation

We use the relational semantics presented in Section 4.2 to define a refinement ordering on our data types. We say that communicating data type A is data refined by communicating data type C if, for each sequence of named operations, the effect upon C is more deterministic than the effect upon A. For all such sequences p, the relation $R\,C\,p$ must be a subset of the relation $R\,A\,p$. When this is the case, we write $A \sqsubseteq_R C$. Formally,

$$
\begin{array}{l}
[Abs, Con, Name, Value] \\
\hline
\_ \sqsubseteq_R \_ : CDT[Abs, Name, Value] \leftrightarrow CDT[Con, Name, Value] \\
\hline
\forall\, A : CDT[Abs, Name, Value];\ C : CDT[Con, Name, Value] \bullet \\
\qquad A \sqsubseteq_R C \\
\qquad\qquad \Leftrightarrow \\
\qquad\qquad\quad \forall\, prog : \mathrm{seq}\, Name \ \bullet \ R\, C\, prog \ \subseteq \ R\, A\, prog
\end{array}
$$

Alternatively, refinement of data types may be established inductively using simple sets of simulation rules [HHS87, dRE98]; the simulation rules for our communicating data types within the blocking semantic model are presented in [WDB00]. However in Section 3.3 we showed how we may employ the singleton failures semantic model to automatically establish refinement using the model-checker FDR.

4.4 Corresponding processes

The process corresponding to a communicating data type will be parametrised by the current value of the internal state space. Initially it can be any value from the set *init*.

The body of the process will contain an external choice of operation and input value. Provided the external choice was not over the empty set in which case the process would deadlock, this external choice will be followed by an internal choice of after-state and output value. This, in turn, will be followed by the communication of the associated event.

The process will accept any combination of operation and input value provided the combination of the current internal state—recorded as the parameter of the process—and the given input value lie in the domain of the chosen operation. The process will then internally choose any combination of output value and after-state lying in the range of the chosen operation with the current state and chosen input value. It will communicate the event associated with the operation and these input and output values before behaving as the same process but parametrised by the chosen after-state. More concretely, if the current parameter is a then the combination of input value in and operation $name$ may be chosen only if the predicate $(a, in) \in \operatorname{dom} ops\ name$ is true. If this is not the case then that combination of $name$ and in simply cannot be chosen; they are blocked. If it is the case that the chosen combination satisfies the predicate $(a, in) \in \operatorname{dom} ops\ name$ then an output value out and an after-state a' such that the predicate $(a, in) \mapsto (a', out) \in ops\ n$ is true will be chosen internally. After the event $\mathrm{op}.name.in.out$ has been communicated, the process will continue with parameter a'. We see that the body of the process is as described below:

$$
\begin{array}{l}
\Box\, name : \operatorname{dom} ops;\ in : Value \mid (a, in) \in \operatorname{dom} ops\ name \bullet \\
\qquad\qquad \sqcap\, a' : state;\ out : Value \mid (a, in) \mapsto (a', out) \in ops\ name \bullet \\
\qquad\qquad\qquad\qquad \mathrm{op}.name.in.out \rightarrow Proc(a')
\end{array}
$$

Combining these observations, we see that the function $process_{io}$ which takes a communicating data type and returns the corresponding CSP process is as defined below:

$$
\begin{array}{l}
process_{io} \ = \\
\quad (\, \lambda\, CDT[State, Name, Value] \bullet \\
\qquad \text{let} \\
\qquad\quad Proc(a) \ = \ \Box\, name : \operatorname{dom} ops;\ in : Value \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad (a, in) \in \operatorname{dom} ops\ name \bullet \\
\qquad\qquad\qquad\qquad\quad \sqcap\, a' : State;\ out : Value \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad (a, in) \mapsto (a', out) \in ops\ name \bullet \\
\qquad\qquad\qquad\qquad\qquad \mathrm{op}.name.in.out \rightarrow Proc(a') \\
\qquad \text{within} \\
\qquad\quad \sqcap\, a' : init \ \bullet \ Proc(a')\,).
\end{array}
$$

4.5 Equivalence of refinement

In this section we present the key result of this paper: that data refinement of our communicating data types is equivalent to the singleton failures refinement of their corresponding processes. Given two data types, the first is data refined by the second precisely when the process corresponding to the first is refined within the singleton failures semantic model by the process corresponding to the second.

Theorem 2 *Equivalence of refinement.*

Data refinement of communicating data types is equivalent to the singleton failures refinement of their process equivalents. More formally,

$$\forall A : CDT[AState, Name, Value];\ C : CDT[CState, Name, Value] \bullet$$
$$A \sqsubseteq_R C \iff (process_{io}\, A) \sqsubseteq_S (process_{io}\, C).$$

Proof

The proof for this theorem is given in Appendix A.

5 Related work

In this paper we have introduced an alternative semantic model for CSP. Our motivation was that the singleton failures refinement coincides precisely with data refinement of our communicating data types. But combining state-based and behavioural formalisms is by no means a new area of research: these issues have been addressed for well over a decade.

5.1 Linking action systems and state-transition systems with CSP

In [Jos88] Josephs adopts a state-based approach to the refinement and specification of non-divergent CSP processes; Morgan [Mor90] defines a correspondence between action systems [BKS83] and CSP by expressing the traces, failures and divergences of CSP processes as weakest pre-condition formulae [Dij76]; Butler [But92] addresses the issue of unbounded nondeterminism and extends Morgan's correspondence [Mor90] to the infinite traces model; and He [He89], like Josephs, expresses CSP processes as labelled transition systems and presents sets of sound and complete simulation rules. In each case refinement within the state-based formalism is shown to be equivalent either to the stable failures refinement or the failures-divergences refinement of CSP.

In this section we ask why have we not got the same result? What is it that is different about the refinement of data types to that of action systems and state transition systems? Given that it is easy to express data types as action systems or as state transition systems and vice versa, why are their refinement orderings different?

The answer to *why* their refinement orderings are different is because the refinement orderings of action systems and state transition systems are defined in terms of their simulation rules—and these coincide with the failures refinement ordering within CSP—whereas the simulation rules of our data types are *derived* to coincide with refinement ordering induced by the relational semantics. And, as we have shown, this is not equivalent to the failures refinement ordering.

The answer to the question *what* is it that is different about the refinement of data types to the refinement of action systems and state transition systems lies with the simulation rules of the respective models. More specifically, it lies with the simulation rule that deals with the availability of the next operation and is best illustrated by considering the relevant backwards simulation rules. For clarity we will highlight the differences by considering the simpler versions of the simulation rules, those concerning operations that do not have (explicit) input or output values.

In the case of our data types, priming the components of the more concrete data type and given backwards retrieve relation s, the relevant backwards simulation rule is as follows:

$$\forall\, n : \mathrm{dom}\, ops' \;\bullet\; state' \setminus (\, \mathrm{dom}\, ops'\, n\,) \;\subseteq\; \mathrm{dom}\,(\, s \rhd \mathrm{dom}\, ops\, n\,).$$

This states that if a state of the more concrete data type lies outside the domain of a given operation then there must be a corresponding state that lies outside the domain of the given operation on the corresponding data type. Observe that this property is stated in terms of *individual* operations; if a state of the more concrete data type lies outside the domain of two operations it need not be the case that there is a *single* state of the more abstract data type that lies outside the domain of both of these operations. The sets of corresponding abstract states that lie outside the domains of these operations must be non-empty but their intersection may be empty.

We immediately see the difference between the above rule and Josephs' corresponding upwards simulation rule. Josephs' rule considers combinations of events: the set $next_{P_1}(\sigma_1)$ contains all those events that the more abstract state transition system $P1$ is able to engage in from state σ_1. The relevant simulation rule is given by:

$$\forall\, \sigma_2 \in S_2 \;\;.\;\; \exists\, \sigma_1 \in S_1 \;\;.\;\; \sigma_2\, U \sigma_1 \;\wedge\; next_{P_1}(\sigma_1) \;\subseteq\; next_{P_2}(\sigma_2).$$

This rule states that for every state of the more concrete state-transition system there is a corresponding state of the more abstract state-transition system from which the only operations that may be available are those operations that are also available from the given concrete state. Equivalently, every state of the more concrete state transition system must correspond to a *single* state of the more abstract state transition system that lies outside the domain of *every* transition that the more concrete state lies outside the domain of.

Similarly in [WM90], Woodcock and Morgan's corresponding backwards simulation rules for action systems gives information about combinations of operations. For all *sets* of action names *ref*, the relevant backwards simulation rule is as follows.

$$\neg\,(\, gd_C\ ref\,) \;\;\Rightarrow\;\; \exists\, a \;\bullet\; I \wedge \neg\,(gd_A\ ref).$$

This states that any concrete state that lies outside the preconditions of a given set of operations must correspond to a single abstract state that also lies outside the preconditions all the operations in the given set.

5.2 Linking Z or Object-Z and CSP

In Section 5.1 above we discussed the difference between data refinement and the refinement of action systems and state transition systems. But our research into this area—the information content of data refinement and the definition of the induced singleton failures semantic model—has brought to the fore some inconsistencies and anomalies in the literature. In this section we discuss some of these and how they have arisen.

The anomalies we are concerned with have stemmed from the assumption that the simulation rules presented by Woodcock and Davies [WD96] are equivalent in essence, albeit using a different totalisation and different notation, to their predecessors the simulation rules of Josephs [Jos88] and Woodcock and Morgan [WM90]. Josephs and Woodcock and Morgan have each proved that their simulation rules are both sound and jointly complete with respect to the failures model of CSP and the misconception was that this result holds also for the simulation rules of Woodcock and Davies. Indeed, in [BDW99] the authors of this paper with Woodcock—adopting a blocking model—proved that the refinement of (simple non-communicating) data types *with a single operation* is equivalent to the failures refinement of their process equivalents; they then assumed that this result was extensible and concluded that it held for *all* data types. In fact, as we have proved in this paper, data refinement is equivalent not to failures refinement but to singleton failures refinement. Naturally for data types with a single operation the refinement ordering was equivalent to the stable failures refinement, since the singleton failures refinement is a projection of the stable failures refinement considering only those failures in which the refusal set contains at most one element. But this result was *not* extensible.

Other examples of inconsistencies occur in papers linking Object-Z and CSP. Object-Z [DR00, Smi00] is an object-oriented extension of Z that includes notions of class, instance, inheritance and polymorphism. In Section 4.1 we discussed the totalisation of partial relations and the issue of what happens when an operation is called outside its preconditions. In Object-Z they adopt the blocking model that we have been focusing on in this paper. Using this model Smith [Smi97] gives a failures-divergences semantics to Object-Z by identifying the traces, failures and divergences of Object-Z classes. But then in [SD97] he and Derrick present not only the simulation rules of [Jos88] but also a set of incompatible simulation rules of the style of the Woodcock and Davies [WD96] schema versions of the simulation rules. They argue that these two sets of rules are equivalent because "the event corresponding to an Object-Z operation *op* is in $next_P(\sigma)$ iff Pre *op* is true in the state representing σ." Whilst this statement is true it does not acknowledge the loss of information about the availability of combinations of operations in the second set of simulation rules. This loss of information is because the operators *next* and Pre are not equivalent; *next* takes a state and returns a set of operations whereas the Pre takes an operation and returns a set of states—to be precise it returns the set of states from which the given operation is available. The misconception is that these simulation rules are sound and jointly complete with respect to failures divergences refinement; this misconception has been propagated throughout the literature in this area [BD99, Fis00, DB01].

6 Conclusions

We observed at the beginning of the paper that the equivalence of data refinement and singleton failures refinement enables us to automatically verify the refinement of data types. Rather than manually checking that the simulations rules hold, refinement of data types may be established using the model-checker FDR.

But we conclude the paper by asking whether we get more than an additional tool for verifying data refinement. From a software engineering perspective, is the singleton failures semantic model a useful model in its own right? The answer to this is yes. Certainly it will often be the case that the traces model is sufficient for a software engineer's needs. And at other times they may need the full strength of the stable failures model or indeed the failures-divergences model. But the singleton failures model lies between these. It may well be the case—testing springs to mind—that we need more information than the traces model but not as much as the stable failures model.

Consider processes P, Q and R below.

$$\begin{array}{rcl} P & = & a \rightarrow Stop \ \Box \ b \rightarrow Stop \\ Q & = & a \rightarrow Stop \ \sqcap \ b \rightarrow Stop \\ R & = & a \rightarrow Stop \ \sqcap \ b \rightarrow Stop \ \sqcap Stop \end{array}$$

Within the traces model we cannot distinguish between any of these processes. Within the stable failures model we can distinguish between all three. However, within the singleton failures model, we can distinguish P from both Q and R—we know that either of these processes might refuse a and either might refuse b whereas process P can refuse neither a nor b—but we cannot distinguish between processes Q and R. If we simply need to determine whether or not a particular event *might* be refused after a given trace then the singleton failures model exactly meets our needs.

Acknowledgements

We would like to thank Martín Abadi, Michael Goldsmith, Gavin Lowe, Jeff Sanders, and Jim Woodcock for their helpful and insightful comments.

References

[BD99] H. Bowman and J. Derrick. A junction between state based and behavioural specification. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proceedings of Formal methods for open object-based distributed systems (FMOODS '99)*. Kluwer, 1999. Invited Paper.

[BDW99] C. M. Bolton, J. Davies, and J. C. P. Woodcock. On the refinement and simulation of data types and processes. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of Integrated Formal Methods (IFM'99)*. Springer, 1999.

[BKS83] R. J. R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *In 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1983.

[But92] M.J. Butler. *A CSP approach to Action Systems*. PhD thesis, Oxford University Computing Laboratory, Oxford University, 1992.

[DB01] J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer, 2001.

[Dij76] E. W. Dijkstra. *A discipline of Programming*. Prentice Hall International Series in Computer Science, 1976.

[DR00] R. Duke and G. Rose. *Formal object-oriented specification using Object-Z.* Macmillan, 2000.

[dRE98] W.-P. de Roever and K. Engelhardt. *Data refinement: model-oriented proof methods and their comparison.* Cambridge Tracts in Theoretical Computer Science, 1998.

[Fis00] C. Fischer. *Combination and implementation of processes and data: from CSP-OZ to Java.* PhD thesis, University of Oldenburg, 2000.

[For97] Formal Systems (Europe) Limited. *Failures-Divergences Refinement: FDR2*, 1997. FDR2 User Manual.

[He89] J. He. Process refinement. In J. McDermid, editor, *The theory and practice of refinement.* Butterworths, 1989.

[HHS87] C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 1987.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Jos88] M. B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

[Mor90] C.C. Morgan. Of wp and CSP. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra.* Springer-Verlag, 1990.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall Series in Computer Science, 1998.

[SD97] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M. Hinchey and Shaoying Liu, editors, *Proceedings of First IEEE International Conference on Formal Engineering Methods (ICFEM '97).* IEEE Computer Society, 1997.

[Smi97] 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, *Proceedings of FME '97*, volume 1313 of *LNCS.* Springer-Verlag, 1997.

[Smi00] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, 2000.

[Spi92] J. M. Spivey. *The Z notation: a reference manual.* Prentice-Hall International, 1992.

[WD96] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement.* Prentice Hall International Series in Computer Science, 1996.

[WDB00] J. C. P. Woodcock, J. Davies, and C. M. Bolton. Abstract data types and processes. In J. Davies, A. W. Roscoe, and J. C. P. Woodcock, editors, *Millenial perspectives in computer science: proceedings of the 1999 symposium in honour of Sir Tony Hoare.* Palgrave, 2000.

[WM90] J. C. P. Woodcock and C. C. Morgan. Refinement of state-based concurrent systems. In *VDM Symposium 1990*, volume 428 of *LNCS*, pages 340–351. Springer-Verlag, 1990.

A Refinement

In this section we present the proof that refinement of communicating data types is equivalent to the singleton failures refinement of their corresponding processes. We start with a preliminary lemma before presenting the main body of the proof.

Lemma 2

$$
\begin{array}{l}
\forall\, A : CDT[State, Name, Value];\ prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value; \\
\qquad\qquad a_i : init;\ a_f : State_{\perp} \bullet \\
\quad (a_i, ins, \langle\rangle) \mapsto (a_f, \langle\rangle, outs) \\
\qquad\qquad \in \\
\qquad\qquad\quad run\ prog\ \{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value; \\
\qquad\qquad\qquad\qquad\qquad\qquad a, a' : State \mid \\
\qquad\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad\qquad\quad (a, \langle i\rangle \frown is, os) \mapsto \\
\qquad\qquad\qquad\qquad\qquad (a', is, os \frown \langle o\rangle)\}\} \\
\qquad \Leftrightarrow \\
\qquad\quad \exists\, as : \mathrm{seq}\, State_{\perp} \mid \#as = (\, \#ins + 1\,) \bullet \\
\qquad\qquad head\ as = a_i\ \wedge\ last\ as = a_f \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, r : 1 \,..\, \#outs \bullet \\
\qquad\qquad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\ (prog.r) \\
\qquad\qquad \wedge \\
\qquad\qquad \#ins > \#outs \\
\qquad\qquad\quad \Rightarrow \\
\qquad\qquad\qquad (as.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad \mathrm{dom}\ ops\ (prog.(\#outs + 1)) \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, r : (\, \#outs + 2\,) \,..\, \#as\ \bullet\ as.r\ =\ \perp
\end{array}
$$

Proof

We prove this by induction on the length of program prog observing that the sequences prog and ins will always be of equal lengths and that the length of sequence outs will be at most the length of sequence prog.

Given truth set $\mathcal{T}$ our inductive hypothesis is as follows:

$$
\begin{array}{l}
\forall n : \mathbb{N} \bullet n \in \mathcal{T} \\
\quad \Leftrightarrow \\
\quad\quad \forall A : CDT[State, Name, Value];\ prog : \operatorname{seq} Name; \\
\quad\quad\quad ins, outs : \operatorname{seq} Value;\ a_i : init;\ a_f : State_{\perp} \mid \\
\quad\quad\quad\quad \#prog = n \bullet \\
\quad\quad\quad (a_i, ins, \langle\rangle) \mapsto (a_f, \langle\rangle, outs) \\
\quad\quad\quad\quad \in \\
\quad\quad\quad\quad run\ prog\ \{n : \operatorname{dom} ops \bullet \\
\quad\quad\quad\quad\quad n \mapsto \{is, os : \operatorname{seq} Value;\ i, o : Value; \\
\quad\quad\quad\quad\quad\quad a, a' : State \mid \\
\quad\quad\quad\quad\quad\quad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\quad\quad\quad\quad\quad (a, \langle i \rangle \frown is, os) \mapsto \\
\quad\quad\quad\quad\quad\quad (a', is, os \frown \langle o \rangle)\}\} \\
\quad\quad\quad \Leftrightarrow \\
\quad\quad\quad\quad \exists\, as : \operatorname{seq} State_{\perp} \mid \#as = (\#ins + 1) \bullet \\
\quad\quad\quad\quad\quad head\ as = a_i \ \wedge\ last\ as = a_f \\
\quad\quad\quad\quad\quad \wedge \\
\quad\quad\quad\quad\quad \forall r : 1 \mathbin{..} \#outs \bullet \\
\quad\quad\quad\quad\quad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \\
\quad\quad\quad\quad\quad\quad\quad \in ops\,(prog.r) \\
\quad\quad\quad\quad\quad \wedge \\
\quad\quad\quad\quad\quad \#ins > \#outs \\
\quad\quad\quad\quad\quad\quad \Rightarrow \\
\quad\quad\quad\quad\quad\quad\quad (as.(\#outs+1), ins.(\#outs+1)) \\
\quad\quad\quad\quad\quad\quad\quad\quad \notin \\
\quad\quad\quad\quad\quad\quad\quad\quad \operatorname{dom} ops\,(prog.(\#outs+1)) \\
\quad\quad\quad\quad\quad \wedge \\
\quad\quad\quad\quad\quad \forall r : (\#outs + 2) \mathbin{..} \#as \bullet as.r = \perp
\end{array}
$$

Base case: n = 0

First we establish the base case when n is zero; prog is the empty sequence. Since the input sequence is the same length as prog and the output sequence is at most the same length as prog we conclude that both these sequences are also the empty sequence.

Given abstract data type $A : CDT[State, Name, Value]$ and states $a_i : init$ and $a_f : State_{\perp}$ we *reason as follows:*

$$
\begin{array}{l}
(\ (a_i, \langle\rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, \langle\rangle) \\
\qquad \in \\
\qquad\quad run\ \langle\rangle\ \{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad\qquad (a, \langle i\rangle \frown is, os) \mapsto (a', is, os \frown \langle o\rangle)\}\} \\
\qquad \Leftrightarrow \\
\qquad\quad \exists\, as : \mathrm{seq}\ State_\perp \mid \#as = (\ \#\ \langle\rangle\ +1\ ) \bullet \\
\qquad\qquad head\ as = a_i\ \wedge\ last\ as = a_f \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, r : 1 \,..\, \#\ \langle\rangle \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\ (prog.r) \\
\qquad\qquad \wedge \\
\qquad\qquad \#\ \langle\rangle > \#\ \langle\rangle \\
\qquad\qquad\qquad \Rightarrow \\
\qquad\qquad\qquad\quad (as.(\#\ \langle\rangle\ +1), ins.(\#\ \langle\rangle\ +1)) \\
\qquad\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\qquad\quad \mathrm{dom}\ ops\ (prog.(\#\ \langle\rangle\ +1)) \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, r : (\ \#\ \langle\rangle\ +2\ ) \,..\, \#as\ \bullet\ as.r\ =\ \perp\ ) \\
\\
\Leftrightarrow \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\mathit{def}^n\ \mathit{of}\ \mathit{run}] \\
\quad (a_i, \langle\rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, \langle\rangle)\ \in\ \mathrm{id} \\
\qquad \Leftrightarrow \\
\qquad\quad \exists\, as : \mathrm{seq}\ State_\perp \mid \#as = (\ \#\ \langle\rangle\ +1\ ) \bullet \\
\qquad\qquad head\ as = a_i\ \wedge\ last\ as = a_f \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, r : 1 \,..\, \#\ \langle\rangle \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\ (prog.r) \\
\qquad\qquad \wedge \\
\qquad\qquad \#\ \langle\rangle > \#\ \langle\rangle \\
\qquad\qquad\qquad \Rightarrow \\
\qquad\qquad\qquad\quad (as.(\#\ \langle\rangle\ +1), ins.(\#\ \langle\rangle\ +1)) \\
\qquad\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\qquad\quad \mathrm{dom}\ ops\ (prog.(\#\ \langle\rangle\ +1)) \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, r : (\ \#\ \langle\rangle\ +2\ ) \,..\, \#as\ \bullet\ as.r\ =\ \perp
\end{array}
$$

$\Leftrightarrow$ *[property of #]*

$$
\begin{array}{l}
(a_i, \langle\rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, \langle\rangle) \ \in \ \mathrm{id} \\
\quad \Leftrightarrow \\
\quad\quad \exists\, as : \mathrm{seq}\, State_\bot \mid \# as = 1 \bullet \\
\quad\quad\quad head\ as = a_i \ \wedge \ last\ as = a_f \\
\quad\quad\quad \wedge \\
\quad\quad\quad \forall\, r : 1 \mathrel{..} 0 \bullet \\
\quad\quad\quad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\quad\quad\quad \wedge \\
\quad\quad\quad 0 > 0 \\
\quad\quad\quad\quad \Rightarrow \\
\quad\quad\quad\quad\quad (as.(\# \langle\rangle + 1), ins.(\# \langle\rangle + 1)) \\
\quad\quad\quad\quad\quad\quad \notin \\
\quad\quad\quad\quad\quad\quad\quad \mathrm{dom}\ ops\,(prog.(\# \langle\rangle + 1)) \\
\quad\quad\quad \wedge \\
\quad\quad\quad \forall\, r : (\, 0 + 2\,) \mathrel{..} 1 \ \bullet \ as.r \ = \ \bot
\end{array}
$$

$\Leftrightarrow$ *[property of >]*

$$
\begin{array}{l}
(a_i, \langle\rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, \langle\rangle) \ \in \ \mathrm{id} \\
\quad \Leftrightarrow \\
\quad\quad \exists\, as : \mathrm{seq}\, State_\bot \mid \# as = 1 \bullet \\
\quad\quad\quad head\ as = a_i \ \wedge \ last\ as = a_f \\
\quad\quad\quad \wedge \\
\quad\quad\quad \forall\, r : 1 \mathrel{..} 0 \bullet \\
\quad\quad\quad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\quad\quad\quad \wedge \\
\quad\quad\quad \mathrm{false} \\
\quad\quad\quad\quad \Rightarrow \\
\quad\quad\quad\quad\quad (as.(\# \langle\rangle + 1), ins.(\# \langle\rangle + 1)) \\
\quad\quad\quad\quad\quad\quad \notin \\
\quad\quad\quad\quad\quad\quad\quad \mathrm{dom}\ ops\,(prog.(\# \langle\rangle + 1)) \\
\quad\quad\quad \wedge \\
\quad\quad\quad \forall\, r : 2 \mathrel{..} 1 \ \bullet \ as.r \ = \ \bot
\end{array}
$$

$\Leftrightarrow$ *[property of $\Rightarrow$]*

$$
\begin{array}{l}
(a_i, \langle\rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, \langle\rangle) \ \in \ \mathrm{id} \\
\quad \Leftrightarrow \\
\quad\quad \exists\, as : \mathrm{seq}\, State_\bot \mid \# as = 1 \bullet \\
\quad\quad\quad head\ as = a_i \ \wedge \ last\ as = a_f \\
\quad\quad\quad \wedge \\
\quad\quad\quad \forall\, r : 1 \mathrel{..} 0 \bullet \\
\quad\quad\quad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\quad\quad\quad \wedge \\
\quad\quad\quad \mathrm{true} \\
\quad\quad\quad \wedge \\
\quad\quad\quad \forall\, r : 2 \mathrel{..} 1 \ \bullet \ as.r \ = \ \bot
\end{array}
$$

$\Leftrightarrow$ *[property of $\wedge$]*

$$
\begin{array}{l}
(a_i, \langle\rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, \langle\rangle) \ \in \ \mathrm{id} \\
\quad \Leftrightarrow \\
\quad\quad \exists\, as : \mathrm{seq}\, State_\perp \mid \# as = 1 \bullet \\
\quad\quad\quad head\ as = a_i \ \wedge \ last\ as = a_f \\
\quad\quad\quad \wedge \\
\quad\quad\quad \forall\, r : 1 \mathrel{..} 0 \bullet \\
\quad\quad\quad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\quad\quad\quad \wedge \\
\quad\quad\quad \forall\, r : 2 \mathrel{..} 1 \ \bullet \ as.r \ = \ \perp
\end{array}
$$

$\Leftrightarrow$ *[property of $\mathrel{..}$]*

$$
\begin{array}{l}
(a_i, \langle\rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, \langle\rangle) \ \in \ \mathrm{id} \\
\quad \Leftrightarrow \\
\quad\quad \exists\, as : \mathrm{seq}\, State_\perp \mid \# as = 1 \bullet \\
\quad\quad\quad head\ as = a_i \ \wedge \ last\ as = a_f \\
\quad\quad\quad \wedge \\
\quad\quad\quad \forall\, r : \emptyset \bullet (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\quad\quad\quad \wedge \\
\quad\quad\quad \forall\, r : \emptyset \ \bullet \ as.r \ = \ \perp
\end{array}
$$

$\Leftrightarrow$ *[property of $\forall$]*

$$
\begin{array}{l}
(a_i, \langle\rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, \langle\rangle) \ \in \ \mathrm{id} \\
\quad \Leftrightarrow \\
\quad\quad \exists\, as : \mathrm{seq}\, State_\perp \mid \# as = 1 \bullet \\
\quad\quad\quad head\ as = a_i \ \wedge \ last\ as = a_f \\
\quad\quad\quad \wedge \\
\quad\quad\quad \mathrm{true} \\
\quad\quad\quad \wedge \\
\quad\quad\quad \mathrm{true}
\end{array}
$$

$\Leftrightarrow$ *[property of $\wedge$]*

$$
\begin{array}{l}
(a_i, \langle\rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, \langle\rangle) \ \in \ \mathrm{id} \\
\quad \Leftrightarrow \\
\quad\quad \exists\, as : \mathrm{seq}\, State_\perp \mid \# as = 1 \bullet \\
\quad\quad\quad head\ as = a_i \ \wedge \ last\ as = a_f
\end{array}
$$

$\Leftrightarrow$ *[property of sequences]*

$$
\begin{array}{l}
(a_i, \langle\rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, \langle\rangle) \ \in \ \mathrm{id} \\
\quad \Leftrightarrow \\
\quad\quad a_i \ = \ a_f
\end{array}
$$

$\Leftrightarrow$ *[def^n of id]*

$$(a_i \ = \ a_f) \ \Leftrightarrow \ (a_i \ = \ a_f)$$

$\Leftrightarrow$ *[property of $=$]*

true

We have proved that the base case holds; $0 \in \mathcal{T}$

Inductive step

Suppose now that some natural number k lies in the truth set $\mathcal{T}$. Furthermore, let $prog \frown \langle name \rangle$ be a sequence of names of length $k+1$, $ins \frown \langle in \rangle$ be a sequence of values of length $k+1$, $outs$ be a sequence of values of length at most $k+1$ and let a_i and a_f be states respectively of types init and $State_\bot$. We reason as follows:

$$
\begin{array}{l}
(a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, outs) \\
\quad \in \\
\quad\quad run\ (\ prog \frown \langle name \rangle\ ) \\
\quad\quad\quad \{n : \mathrm{dom}\ ops \bullet \\
\quad\quad\quad\quad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\quad\quad\quad\quad\quad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\quad\quad\quad\quad\quad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\}
\end{array}
$$

$$
\begin{array}{ll}
\Leftrightarrow & \textit{[def}^n \textit{ of run]} \\
\quad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, outs) & \\
\quad\quad \in & \\
\quad\quad\quad run\ prog \{n : \mathrm{dom}\ ops \bullet & \\
\quad\quad\quad\quad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid & \\
\quad\quad\quad\quad\quad (a, i) \mapsto (a', o) \in ops\ n \bullet & \\
\quad\quad\quad\quad\quad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} & \\
\quad\quad\quad\quad \fatsemi & \\
\quad\quad\quad\quad (\ \{n : \mathrm{dom}\ ops \bullet & \\
\quad\quad\quad\quad\quad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid & \\
\quad\quad\quad\quad\quad\quad (a, i) \mapsto (a', o) \in ops\ n \bullet & \\
\quad\quad\quad\quad\quad\quad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\} & \\
\quad\quad\quad\quad\quad\quad\quad \}\ name\ )^{\top} &
\end{array}
$$

$$
\begin{array}{ll}
\Leftrightarrow & \textit{[property of functions]} \\
\quad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, outs) & \\
\quad\quad \in & \\
\quad\quad\quad run\ prog \{n : \mathrm{dom}\ ops \bullet & \\
\quad\quad\quad\quad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid & \\
\quad\quad\quad\quad\quad (a, i) \mapsto (a', o) \in ops\ n \bullet & \\
\quad\quad\quad\quad\quad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} & \\
\quad\quad\quad\quad \fatsemi & \\
\quad\quad\quad\quad \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid & \\
\quad\quad\quad\quad\quad (a, i) \mapsto (a', o) \in ops\ name \bullet & \\
\quad\quad\quad\quad\quad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}^{\top} &
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[def}^n\textit{ of }{}^{\top}\textit{]} \\
\quad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, outs) \\
\qquad \in \\
\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\qquad\qquad ⨾ \\
\qquad\qquad \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\quad (a, i) \mapsto (a', o) \in ops\ name \bullet \\
\qquad\qquad\quad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\} \\
\qquad\qquad \cup \\
\qquad\qquad \{ is, os : \mathrm{seq}\ Value;\ i : Value;\ a : State_{\perp} \mid \\
\qquad\qquad\quad (a, i) \notin \mathrm{dom}(ops\ name) \bullet \\
\qquad\qquad\quad (a, \langle in \rangle \frown is, os) \mapsto (\perp, is, os)\} \\
\\
\Leftrightarrow \hfill \textit{[property of relations]} \\
\quad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, outs) \\
\qquad \in \\
\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\qquad\qquad ⨾ \\
\qquad\qquad \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\quad (a, i) \mapsto (a', o) \in ops\ name \bullet \\
\qquad\qquad\quad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\} \\
\quad \vee \\
\quad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, outs) \\
\qquad \in \\
\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\qquad\qquad ⨾ \\
\qquad\qquad \{ is, os : \mathrm{seq}\ Value;\ i : Value;\ a : State_{\perp} \mid \\
\qquad\qquad\quad (a, i) \notin \mathrm{dom}(ops\ name) \bullet \\
\qquad\qquad\quad (a, \langle in \rangle \frown is, os) \mapsto (\perp, is, os)\}
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[property of } \fatsemi\textit{]} \\
\quad \exists\, a : state;\ i : Value;\ is, os : \mathrm{seq}\ Value \bullet \\
\qquad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a, \langle i \rangle \frown is, os) \\
\qquad\quad \in \\
\qquad\qquad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\qquad \wedge \\
\qquad (a, \langle i \rangle \frown is, os) \mapsto (a_f, \langle\rangle, outs) \\
\qquad\quad \in \\
\qquad\qquad \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ name \bullet \\
\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\} \\
\quad \vee \\
\quad \exists\, a : State_{\perp};\ i : Value;\ is, os : \mathrm{seq}\ Value \bullet \\
\qquad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a, \langle i \rangle \frown is, os) \\
\qquad\quad \in \\
\qquad\qquad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\qquad \wedge \\
\qquad (a, \langle i \rangle \frown is, os) \mapsto (a_f, \langle\rangle, outs) \\
\qquad\quad \in \\
\qquad\qquad \{\, is, os : \mathrm{seq}\ Value;\ i : Value;\ a : State_{\perp} \mid \\
\qquad\qquad\qquad (\, a, i) \notin \mathrm{dom}(ops\ name) \ \bullet \\
\qquad\qquad\qquad (\, a, \langle in \rangle \frown is, os\,) \ \mapsto\ (\,\perp, is, os\,)\,\}
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[set theory]} \\
\quad \exists\, a : state;\ i : Value;\ is, os : \mathrm{seq}\ Value \bullet \\
\qquad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a, \langle i \rangle \frown is, os) \\
\qquad\qquad \in \\
\qquad\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad\qquad\quad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\qquad \wedge \\
\qquad (a, i) \mapsto (a_f, last\ outs) \in ops\ name \\
\qquad \wedge \\
\qquad is = \langle\rangle \wedge\ os = first\ outs \\
\quad \vee \\
\quad \exists\, a : State_\perp;\ i : Value;\ is, os : \mathrm{seq}\ Value \bullet \\
\qquad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a, \langle i \rangle \frown is, os) \\
\qquad\qquad \in \\
\qquad\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad\qquad\quad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\qquad \wedge \\
\qquad (a, i) \notin \mathrm{dom}(ops\ name) \\
\qquad \wedge \\
\qquad a_f = \perp\ \wedge\ is = \langle\rangle \wedge\ os = outs \\
\\
\Leftrightarrow \hfill \textit{[one point rule]} \\
\quad \exists\, a : state;\ i : Value \bullet \\
\qquad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a, \langle i \rangle \frown \langle\rangle, first\ outs) \\
\qquad\qquad \in \\
\qquad\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad\qquad\quad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\qquad \wedge \\
\qquad (a, i) \mapsto (a_f, last\ outs) \in ops\ name \\
\quad \vee \\
\quad \exists\, a : State_\perp;\ i : Value \bullet \\
\qquad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a, \langle i \rangle \frown \langle\rangle, outs) \\
\qquad\qquad \in \\
\qquad\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad\qquad\quad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\qquad \wedge \\
\qquad (a, i) \notin \mathrm{dom}(ops\ name) \\
\qquad \wedge \\
\qquad a_f = \perp
\end{array}
$$

$\Leftrightarrow$ *[property of $\frown$]*

$$
\begin{array}{l}
\exists\, a : state;\ i : Value \bullet \\
\quad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a, \langle i \rangle, first\ outs) \\
\qquad \in \\
\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\quad \wedge \\
\quad (a, i) \mapsto (a_f, last\ outs) \in ops\ name \\
\vee \\
\exists\, a : State_\perp;\ i : Value \bullet \\
\quad (a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a, \langle i \rangle, outs) \\
\qquad \in \\
\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\quad \wedge \\
\quad (a, i) \notin \mathrm{dom}(ops\ name) \\
\quad \wedge \\
\quad a_f = \perp
\end{array}
$$

$\Leftrightarrow$ *[property of run]*

$$
\begin{array}{l}
\exists\, a : state;\ i : Value \bullet \\
\quad (a_i, ins, \langle\rangle) \mapsto (a, \langle\rangle, first\ outs) \\
\qquad \in \\
\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\quad \wedge \\
\quad i = in \\
\quad \wedge \\
\quad (a, i) \mapsto (a_f, last\ outs) \in ops\ name \\
\vee \\
\exists\, a : State_\perp;\ i : Value \bullet \\
\quad (a_i, ins, \langle\rangle) \mapsto (a, \langle\rangle, outs) \\
\qquad \in \\
\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\} \\
\quad \wedge \\
\quad i = in \\
\quad \wedge \\
\quad (a, i) \notin \mathrm{dom}(ops\ name) \\
\quad \wedge \\
\quad a_f = \perp
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[one point rule]} \\
\left.\begin{array}{l}
\exists\, a : state \bullet \\
\quad (a_i, ins, \langle\rangle) \mapsto (a, \langle\rangle, first\ outs) \\
\qquad \in \\
\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad (a, \langle i\rangle \frown is, os) \mapsto (a', is, os \frown \langle o\rangle)\}\} \\
\quad \wedge \\
\quad (a, in) \mapsto (a_f, last\ outs) \in ops\ name
\end{array}\right\}(1) \\
\vee \\
\left.\begin{array}{l}
\exists\, a : State_\perp \bullet \\
\quad (a_i, ins, \langle\rangle) \mapsto (a, \langle\rangle, outs) \\
\qquad \in \\
\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad (a, \langle i\rangle \frown is, os) \mapsto (a', is, os \frown \langle o\rangle)\}\} \\
\quad \wedge \\
\quad (a, in) \notin \mathrm{dom}(ops\ name) \\
\quad \wedge \\
\quad a_f = \perp
\end{array}\right\}(2)
\end{array}
$$

We will consider these two clauses separately. Clause (1) *refers to those cases where the sequences* $ins \frown \langle in\rangle$ *and outs are the same length and clause* (2) *refers to those cases where the length of the sequence outs is at most the length of the sequence ins. We consider first the case when* $ins \frown \langle in\rangle$ *and outs are the same length and reason as follows:*

$$
\begin{array}{l}
\exists\, a : state \bullet \\
\quad (a_i, ins, \langle\rangle) \mapsto (a, \langle\rangle, first\ outs) \\
\qquad \in \\
\qquad\quad run\ prog\{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad (a, \langle i\rangle \frown is, os) \mapsto (a', is, os \frown \langle o\rangle)\}\} \\
\quad \wedge \\
\quad (a, in) \mapsto (a_f, last\ outs) \in ops\ name
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[inductive hyposthesis]} \\
\quad \exists\, a : \mathit{state} \bullet \\
\qquad \exists\, \mathit{as} : \mathrm{seq}\, \mathit{State}_\perp \mid \#\mathit{as} = (\, \#\mathit{ins} + 1\,) \bullet \\
\qquad\quad \mathit{head}\ \mathit{as} = a_i \ \wedge\ \mathit{last}\ \mathit{as} = a \\
\qquad\quad \wedge \\
\qquad\quad \forall\, r : 1 \mathinner{\ldotp\ldotp} \#(\mathit{first}\ \mathit{outs}) \bullet \\
\qquad\qquad (\mathit{as}.r, \mathit{ins}.r) \mapsto (\mathit{as}.(r+1), (\mathit{first}\ \mathit{outs}).r) \\
\qquad\qquad\quad \in \mathit{ops}\,(\mathit{prog}.r) \\
\qquad\quad \wedge \\
\qquad\quad \#\mathit{ins} > \#(\mathit{first}\ \mathit{outs}) \\
\qquad\qquad \Rightarrow \\
\qquad\qquad\quad (\mathit{as}.(\#(\mathit{first}\ \mathit{outs}) + 1), \mathit{ins}.(\#(\mathit{first}\ \mathit{outs}) + 1)) \\
\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\quad \mathrm{dom}\, \mathit{ops}\,(\mathit{prog}.(\#(\mathit{first}\ \mathit{outs}) + 1)) \\
\qquad\quad \wedge \\
\qquad\quad \forall\, r : (\, \#(\mathit{first}\ \mathit{outs}) + 2\,) \mathinner{\ldotp\ldotp} \#\mathit{as} \ \bullet\ \mathit{as}.r \ = \ \perp \\
\qquad \wedge \\
\qquad (a, \mathit{in}) \mapsto (a_f, \mathit{last}\ \mathit{outs}) \in \mathit{ops}\ \mathit{name}
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[property of } \exists\textit{]} \\
\quad \exists\, a : \mathit{state};\ \mathit{as} : \mathrm{seq}\, \mathit{State}_\perp \mid \#\mathit{as} = (\, \#\mathit{ins} + 1\,) \bullet \\
\qquad \mathit{head}\ \mathit{as} = a_i \ \wedge\ \mathit{last}\ \mathit{as} = a \\
\qquad \wedge \\
\qquad \forall\, r : 1 \mathinner{\ldotp\ldotp} \#(\mathit{first}\ \mathit{outs}) \bullet \\
\qquad\quad (\mathit{as}.r, \mathit{ins}.r) \mapsto (\mathit{as}.(r+1), (\mathit{first}\ \mathit{outs}).r) \\
\qquad\qquad \in \mathit{ops}\,(\mathit{prog}.r) \\
\qquad \wedge \\
\qquad \#\mathit{ins} > \#(\mathit{first}\ \mathit{outs}) \\
\qquad\quad \Rightarrow \\
\qquad\qquad (\mathit{as}.(\#(\mathit{first}\ \mathit{outs}) + 1), \mathit{ins}.(\#(\mathit{first}\ \mathit{outs}) + 1)) \\
\qquad\qquad\qquad\quad \notin \\
\qquad\qquad\qquad\qquad \mathrm{dom}\, \mathit{ops}\,(\mathit{prog}.(\#(\mathit{first}\ \mathit{outs}) + 1)) \\
\qquad \wedge \\
\qquad \forall\, r : (\, \#(\mathit{first}\ \mathit{outs}) + 2\,) \mathinner{\ldotp\ldotp} \#\mathit{as} \ \bullet\ \mathit{as}.r \ = \ \perp \\
\qquad \wedge \\
\qquad (a, \mathit{in}) \mapsto (a_f, \mathit{last}\ \mathit{outs}) \in \mathit{ops}\ \mathit{name}
\end{array}
$$

$$
\begin{array}{ll}
\Leftrightarrow & \textit{[\#ins = \#(first outs)]} \\
\quad \exists\, a : state;\ \ as : \mathrm{seq}\, State_{\bot} \mid \#as = (\ \#ins + 1\ ) \bullet & \\
\qquad head\ as = a_i\ \wedge\ last\ as = a & \\
\qquad \wedge & \\
\qquad \forall\, r : 1 \mathinner{\ldotp\ldotp} \#(first\ outs) \bullet & \\
\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), (first\ outs).r) & \\
\qquad\qquad\qquad \in ops\ (prog.r) & \\
\qquad \wedge & \\
\qquad \text{false} & \\
\qquad\qquad \Rightarrow & \\
\qquad\qquad\quad (as.(\#(first\ outs) + 1), ins.(\#(first\ outs) + 1)) & \\
\qquad\qquad\qquad\qquad \notin & \\
\qquad\qquad\qquad\qquad\ \ \mathrm{dom}\ ops\ (prog.(\#(first\ outs) + 1)) & \\
\qquad \wedge & \\
\qquad \forall\, r : (\ \#(first\ outs) + 2\ ) \mathinner{\ldotp\ldotp} \#as\ \bullet\ as.r\ =\ \bot & \\
\qquad \wedge & \\
\qquad (a, in) \mapsto (a_f, last\ outs) \in ops\ name & \\
& \\
\Leftrightarrow & \textit{[property of} \Rightarrow\textit{]} \\
\quad \exists\, a : state;\ \ as : \mathrm{seq}\, State_{\bot} \mid \#as = (\ \#ins + 1\ ) \bullet & \\
\qquad head\ as = a_i\ \wedge\ last\ as = a & \\
\qquad \wedge & \\
\qquad \forall\, r : 1 \mathinner{\ldotp\ldotp} \#(first\ outs) \bullet & \\
\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), (first\ outs).r) & \\
\qquad\qquad\qquad \in ops\ (prog.r) & \\
\qquad \wedge & \\
\qquad \text{true} & \\
\qquad \wedge & \\
\qquad \forall\, r : (\ \#(first\ outs) + 2\ ) \mathinner{\ldotp\ldotp} \#as\ \bullet\ as.r\ =\ \bot & \\
\qquad \wedge & \\
\qquad (a, in) \mapsto (a_f, last\ outs) \in ops\ name & \\
& \\
\Leftrightarrow & \textit{[one point rule]} \\
\quad \exists\, a : state;\ \ as, as' : \mathrm{seq}\, State_{\bot} \mid & \\
\qquad\qquad \#as = (\ \#ins + 1\ )\ \wedge\ as' = as \frown \langle a_f \rangle \bullet & \\
\qquad head\ as = a_i\ \wedge\ last\ as = a & \\
\qquad \wedge & \\
\qquad \forall\, r : 1 \mathinner{\ldotp\ldotp} \#(first\ outs) \bullet & \\
\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), (first\ outs).r) & \\
\qquad\qquad\qquad \in ops\ (prog.r) & \\
\qquad \wedge & \\
\qquad \text{true} & \\
\qquad \wedge & \\
\qquad \forall\, r : (\ \#(first\ outs) + 2\ ) \mathinner{\ldotp\ldotp} \#as\ \bullet\ as.r\ =\ \bot & \\
\qquad \wedge & \\
\qquad (a, in) \mapsto (a_f, last\ outs) \in ops\ name &
\end{array}
$$

$$
\begin{array}{ll}
\Leftrightarrow & \textit{[substitution]} \\
\quad \exists\, a : state;\ as, as' : \mathrm{seq}\, State_{\perp} \mid & \\
\qquad\qquad \#as' = (\, \#(ins \frown \langle in \rangle) + 1\,) \ \wedge\ last\ as' = a_f \bullet & \\
\qquad head\ as' = a_i \ \wedge\ as'.(\#as' - 1) = a & \\
\qquad \wedge & \\
\qquad \forall\, r : 1 \mathrel{..} \#(first\ outs) \bullet & \\
\qquad\qquad (as'.r, (ins \frown \langle in \rangle).r) \mapsto (as'.(r+1), outs.r) & \\
\qquad\qquad\qquad \in ops\,((prog \frown \langle name \rangle).r) & \\
\qquad \wedge & \\
\qquad \mathrm{true} & \\
\qquad \wedge & \\
\qquad \forall\, r : (\, \#outs + 1\,) \mathrel{..} (\#as' - 1) \ \bullet\ as'.r \ = \ \perp & \\
\qquad \wedge & \\
\qquad (a, in) \mapsto (a_f, last\ outs) \in ops\ name &
\end{array}
$$

$$
\begin{array}{ll}
\Leftrightarrow & \textit{[property of } \Rightarrow\textit{]} \\
\quad \exists\, a : state;\ as, as' : \mathrm{seq}\, State_{\perp} \mid & \\
\qquad\qquad \#as' = (\, \#(ins \frown \langle in \rangle) + 1\,) \ \wedge\ last\ as' = a_f \bullet & \\
\qquad head\ as' = a_i \ \wedge\ as'.(\#as' - 1) = a & \\
\qquad \wedge & \\
\qquad \forall\, r : 1 \mathrel{..} \#(first\ outs) \bullet & \\
\qquad\qquad (as'.r, (ins \frown \langle in \rangle).r) \mapsto (as'.(r+1), outs.r) & \\
\qquad\qquad\qquad \in ops\,((prog \frown \langle name \rangle).r) & \\
\qquad \wedge & \\
\qquad \mathrm{false} & \\
\qquad\qquad \Rightarrow & \\
\qquad\qquad\quad (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) & \\
\qquad\qquad\qquad\qquad \notin & \\
\qquad\qquad\qquad\qquad\quad \mathrm{dom}\ ops\,((prog \frown \langle name \rangle).(\#outs + 1)) & \\
\qquad \wedge & \\
\qquad \forall\, r : (\, \#outs + 1\,) \mathrel{..} (\#as' - 1) \ \bullet\ as'.r \ = \ \perp & \\
\qquad \wedge & \\
\qquad (a, in) \mapsto (a_f, last\ outs) \in ops\ name &
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill [\#(ins \frown \langle in \rangle) = \#outs] \\
\quad \exists\, a : state;\ as, as' : \operatorname{seq} State_{\perp} \mid \\
\qquad\qquad \#as' = (\, \#(ins \frown \langle in \rangle) + 1\,) \ \wedge\ last\ as' = a_f \bullet \\
\qquad head\ as' = a_i \ \wedge\ as'.(\#as' - 1) = a \\
\qquad \wedge \\
\qquad \forall\, r : 1 \mathrel{..} \#(first\ outs) \bullet \\
\qquad\qquad (as'.r, (ins \frown \langle in \rangle).r) \mapsto (as'.(r+1), outs.r) \\
\qquad\qquad\qquad \in ops\ ((prog \frown \langle name \rangle).r) \\
\qquad \wedge \\
\qquad \#(ins \frown \langle in \rangle) > \#outs \\
\qquad\qquad \Rightarrow \\
\qquad\qquad\quad (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \\
\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\quad \operatorname{dom} ops\ ((prog \frown \langle name \rangle).(\#outs + 1)) \\
\qquad \wedge \\
\qquad \forall\, r : (\, \#outs + 1\,) \mathrel{..} (\#as' - 1) \ \bullet\ as'.r \ = \ \perp \\
\qquad \wedge \\
\qquad (a, in) \mapsto (a_f, last\ outs) \in ops\ name
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill [property\ of\ \forall] \\
\quad \exists\, as' : \operatorname{seq} State_{\perp} \mid \\
\qquad\qquad \#as' = (\, \#(ins \frown \langle in \rangle) + 1\,) \ \wedge\ last\ as' = a_f \bullet \\
\qquad head\ as' = a_i \\
\qquad \wedge \\
\qquad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad (as'.r, (ins \frown \langle in \rangle).r) \mapsto (as'.(r+1), outs.r) \\
\qquad\qquad\qquad \in \\
\qquad\qquad\qquad\quad ops\ ((prog \frown \langle name \rangle).r) \\
\qquad \wedge \\
\qquad \#(ins \frown \langle in \rangle) > \#outs \\
\qquad\qquad \Rightarrow \\
\qquad\qquad\quad (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \\
\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\quad \operatorname{dom} ops\ ((prog \frown \langle name \rangle).(\#outs + 1)) \\
\qquad \wedge \\
\qquad \forall\, r : (\, \#outs + 1\,) \mathrel{..} (\#as' - 1) \ \bullet\ as'.r \ = \ \perp
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[property of } \exists\textit{]} \\
\quad \exists\, as' : \operatorname{seq} State_\perp \mid \#as' = (\, \#(ins \frown \langle in \rangle) + 1 \,) \bullet \\
\qquad head\ as' = a_i \ \wedge \ last\ as' = a_f \\
\qquad \wedge \\
\qquad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\quad (as'.r, (ins \frown \langle in \rangle).r) \mapsto (as'.(r+1), outs.r) \\
\qquad\qquad \in \\
\qquad\qquad\quad ops\ ((prog \frown \langle name \rangle).r) \\
\qquad \wedge \\
\qquad \#(ins \frown \langle in \rangle) > \#outs \\
\qquad\quad \Rightarrow \\
\qquad\qquad (as'.(\#outs+1), (ins \frown \langle in \rangle).(\#outs+1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \operatorname{dom} ops\ ((prog \frown \langle name \rangle).(\#outs+1)) \\
\qquad \wedge \\
\qquad \forall\, r : (\, \#outs + 1 \,) \mathrel{..} (\#as' - 1) \ \bullet \ as'.r \ = \ \perp
\end{array}
$$

Now we consider clause (2)*: those cases where the length of sequence outs is at most the length of sequence ins. We reason as follows:*

$$
\begin{array}{l}
\exists\, a : State_\perp \bullet \\
\quad (a_i, ins, \langle\rangle) \mapsto (a, \langle\rangle, outs) \\
\qquad \in \\
\qquad\quad run\ prog \{ n : \operatorname{dom} ops \bullet \\
\qquad\qquad\qquad n \mapsto \{ is, os : \operatorname{seq} Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad\qquad\quad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle) \} \} \\
\quad \wedge \\
\quad (a, in) \notin \operatorname{dom}(ops\ name) \\
\quad \wedge \\
\quad a_f = \perp
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[inductive hypothesis]} \\
\quad \exists\, a : State_\perp \bullet \\
\qquad \exists\, as : \operatorname{seq} State_\perp \mid \#as = (\, \#ins + 1\,) \bullet \\
\qquad\quad head\ as = a_i \ \wedge\ last\ as = a \\
\qquad\quad \wedge \\
\qquad\quad \forall\, r : 1 \,..\, \#outs \bullet \\
\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \\
\qquad\qquad\quad \in ops\,(prog.r) \\
\qquad\quad \wedge \\
\qquad\quad \#ins > \#outs \\
\qquad\qquad \Rightarrow \\
\qquad\qquad\quad (as.(\#outs+1), ins.(\#outs+1)) \\
\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\quad \operatorname{dom} ops\,(prog.(\#outs+1)) \\
\qquad\quad \wedge \\
\qquad\quad \forall\, r : (\, \#outs + 2\,) \,..\, \#as \ \bullet\ as.r \ =\ \perp \\
\qquad \wedge \\
\qquad (a, in) \notin \operatorname{dom}(ops\ name) \\
\qquad \wedge \\
\qquad a_f = \perp
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[property of } \exists \textit{]} \\
\quad \exists\, a : State_\perp;\ as, as' : \operatorname{seq} State_\perp \mid \\
\qquad\quad \#as = (\, \#ins + 1\,) \ \wedge as' = as \frown \langle a_f \rangle \bullet \\
\qquad head\ as = a_i \ \wedge\ last\ as = a \\
\qquad \wedge \\
\qquad \forall\, r : 1 \,..\, \#outs \bullet \\
\qquad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \\
\qquad\qquad \in ops\,(prog.r) \\
\qquad \wedge \\
\qquad \#ins > \#outs \\
\qquad\quad \Rightarrow \\
\qquad\qquad (as.(\#outs+1), ins.(\#outs+1)) \\
\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\quad \operatorname{dom} ops\,(prog.(\#outs+1)) \\
\qquad \wedge \\
\qquad \forall\, r : (\, \#outs + 2\,) \,..\, \#as \ \bullet\ as.r \ =\ \perp \\
\qquad \wedge \\
\qquad (a, in) \notin \operatorname{dom}(ops\ name) \\
\qquad \wedge \\
\qquad a_f = \perp
\end{array}
$$

$\Leftrightarrow$ *[property of sequences]*

$$
\begin{array}{l}
\exists\, a : State_\perp;\ as' : \operatorname{seq} State_\perp \mid \\
\qquad \#as' = (\,\#ins + 2\,)\ \wedge\ last\ as' = a_f \bullet \\
\quad head\ as' = a_i\ \wedge\ as'.(\#as' - 1) = a \\
\quad \wedge \\
\quad \forall\, r : 1\,..\,\#outs \bullet \\
\qquad (as'.r, (ins \frown \langle in \rangle).r) \mapsto (as'.(r+1), outs.r) \\
\qquad\qquad \in ops\,((prog \frown \langle name \rangle).r) \\
\quad \wedge \\
\quad \#(ins \frown \langle in \rangle) - 1 > \#outs \\
\qquad \Rightarrow \\
\qquad\quad (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \operatorname{dom} ops\,((prog \frown \langle name \rangle).(\#outs + 1)) \\
\quad \wedge \\
\quad \forall\, r : (\,\#outs + 2\,)\,..\,(\#as' - 1)\ \bullet\ as'.r\ =\ \perp \\
\quad \wedge \\
\quad (a, in) \notin \operatorname{dom}(ops\ name) \\
\quad \wedge \\
\quad a_f = \perp
\end{array}
$$

$\Leftrightarrow$ *[property of $\forall$]*

$$
\begin{array}{l}
\exists\, a : State_\perp;\ as' : \operatorname{seq} State_\perp \mid \\
\qquad \#as' = (\,\#ins + 2\,)\ \wedge\ last\ as' = a_f \bullet \\
\quad head\ as' = a_i\ \wedge\ as'.(\#as' - 1) = a \\
\quad \wedge \\
\quad \forall\, r : 1\,..\,\#outs \bullet \\
\qquad (as'.r, (ins \frown \langle in \rangle).r) \mapsto (as'.(r+1), outs.r) \\
\qquad\qquad \in ops\,((prog \frown \langle name \rangle).r) \\
\quad \wedge \\
\quad \#(ins \frown \langle in \rangle) - 1 > \#outs \\
\qquad \Rightarrow \\
\qquad\quad (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \operatorname{dom} ops\,((prog \frown \langle name \rangle).(\#outs + 1)) \\
\quad \wedge \\
\quad \forall\, r : (\,\#outs + 2\,)\,..\,\#as'\ \bullet\ as'.r\ =\ \perp \\
\quad \wedge \\
\quad (a, in) \notin \operatorname{dom}(ops\ name)
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{[one point rule]} \\
\quad \exists\, as' : \mathrm{seq}\, State_{\perp} \mid \#as' = (\, \#ins + 2\,) \bullet \\
\qquad\quad head\ as' = a_i \ \wedge\ last\ as' = a_f \\
\qquad\quad \wedge \\
\qquad\quad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad (as'.r, (ins \frown \langle in \rangle).r) \mapsto (as'.(r+1), outs.r) \\
\qquad\qquad\qquad \in ops\, ((prog \frown \langle name \rangle).r) \\
\qquad\quad \wedge \\
\qquad\quad \#(ins \frown \langle in \rangle) - 1 > \#outs \\
\qquad\qquad \Rightarrow \\
\qquad\qquad\quad (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \\
\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\quad \mathrm{dom}\, ops\, ((prog \frown \langle name \rangle).(\#outs + 1)) \\
\qquad\quad \wedge \\
\qquad\quad \forall\, r : (\, \#outs + 2\,) \mathrel{..} \#as' \ \bullet\ as'.r \ = \ \perp \\
\qquad\quad \wedge \\
\qquad\quad (as'.(\#as' - 1), in) \notin \mathrm{dom}(ops\ name)
\end{array}
$$

Now let's consider the third and fifth clauses:

$$
\begin{array}{l}
\#(ins \frown \langle in \rangle) - 1 > \#outs \\
\quad \Rightarrow \\
\qquad (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \\
\qquad\qquad\quad \notin \\
\qquad\qquad\qquad \mathrm{dom}\, ops\, ((prog \frown \langle name \rangle).(\#outs + 1)) \\
\wedge \\
(as'.(\#as' - 1), in) \notin \mathrm{dom}(ops\ name)
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{[def}^{n}\textit{ if} \Rightarrow\textit{]} \\
\quad (\ \neg\ \#(ins \frown \langle in \rangle) - 1 > \#outs \\
\qquad \vee \\
\qquad (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \mathrm{dom}\, ops\, ((prog \frown \langle name \rangle).(\#outs + 1))\ ) \\
\quad \wedge \\
\quad (as'.(\#as' - 1), in) \notin \mathrm{dom}(ops\ name)
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{[distributivity]} \\
\quad (\ \neg\ \#(ins \frown \langle in \rangle) - 1 > \#outs \\
\qquad \wedge \\
\qquad (as'.(\#as' - 1), in) \notin \mathrm{dom}(ops\ name)\ ) \\
\quad \vee \\
\quad (\ (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \mathrm{dom}\, ops\, ((prog \frown \langle name \rangle).(\#outs + 1))\ )
\end{array}
$$

$\Leftrightarrow$ *[substitution]*

$$(\ (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \notin \operatorname{dom} ops \, ((prog \frown \langle name \rangle).(\#outs + 1)) \ ) \lor (\ (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \notin \operatorname{dom} ops \, ((prog \frown \langle name \rangle).(\#outs + 1)) \ )$$

$\Leftrightarrow$ *[property of $\lor$]*

$$(as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \notin \operatorname{dom} ops \, ((prog \frown \langle name \rangle).(\#outs + 1))$$

We now substitute this result in in place of clauses three and five and continue reasoning as follows:

$\Leftrightarrow$ *[substitution]*

$$\exists \, as' : \operatorname{seq} State_\perp \mid \#as' = (\ \#ins + 2\ ) \bullet \\ head\ as' = a_i \ \land \ last\ as' = a_f \\ \land \\ \forall \, r : 1 \,..\, \#outs \bullet \\ (as'.r, (ins \frown \langle in \rangle).r) \mapsto (as'.(r+1), outs.r) \in ops \, ((prog \frown \langle name \rangle).r) \\ \land \\ (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \notin \operatorname{dom} ops \, ((prog \frown \langle name \rangle).(\#outs + 1)) \\ \land \\ \forall \, r : (\ \#outs + 2\ ) \,..\, \#as' \ \bullet \ as'.r \ = \ \perp$$

$\Leftrightarrow$ *[property of $\Rightarrow$]*

$$\exists \, as' : \operatorname{seq} State_\perp \mid \#as' = (\ \#ins + 2\ ) \bullet \\ head\ as' = a_i \ \land \ last\ as' = a_f \\ \land \\ \forall \, r : 1 \,..\, \#outs \bullet \\ (as'.r, (ins \frown \langle in \rangle).r) \mapsto (as'.(r+1), outs.r) \in ops \, ((prog \frown \langle name \rangle).r) \\ \land \\ \text{true} \Rightarrow (as'.(\#outs + 1), (ins \frown \langle in \rangle).(\#outs + 1)) \notin \operatorname{dom} ops \, ((prog \frown \langle name \rangle).(\#outs + 1)) \\ \land \\ \forall \, r : (\ \#outs + 2\ ) \,..\, \#as' \ \bullet \ as'.r \ = \ \perp$$

$$
\begin{array}{l}
\Leftrightarrow \hfill [\#(ins \frown \langle in \rangle) > \#outs] \\
\quad \exists\, as' : \mathrm{seq}\, State_\perp \mid \#as' = (\, \#ins + 2\,) \bullet \\
\qquad head\ as' = a_i \ \wedge\ last\ as' = a_f \\
\qquad \wedge \\
\qquad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\quad (as'.r, (ins \frown \langle in \rangle).r) \mapsto (as'.(r+1), outs.r) \\
\qquad\qquad \in ops\ ((prog \frown \langle name \rangle).r) \\
\qquad \wedge \\
\qquad \#(ins \frown \langle in \rangle) > \#outs \\
\qquad\quad \Rightarrow \\
\qquad\qquad (as'.(\#outs+1), (ins \frown \langle in \rangle).(\#outs+1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \mathrm{dom}\ ops\ ((prog \frown \langle name \rangle).(\#outs+1)) \\
\qquad \wedge \\
\qquad \forall\, r : (\, \#outs + 2\,) \mathrel{..} \#as' \ \bullet\ as'.r \ = \ \perp \\
\qquad \wedge \\
\qquad (as'.(\#as' - 1), in) \notin \mathrm{dom}(ops\ name)
\end{array}
$$

We have proved that both in the case that the length of outs is equal to the length of $ins \frown \langle in \rangle$ *and when it is less than the length of* $ins \frown \langle in \rangle$*, the following predicate must be true.*

$$
\begin{array}{l}
(a_i, ins \frown \langle in \rangle, \langle\rangle) \mapsto (a_f, \langle\rangle, outs) \\
\quad \in \\
\qquad run\ (\ prog \frown \langle name \rangle\ ) \\
\qquad\quad \{n : \mathrm{dom}\ ops \bullet \\
\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\ Value;\ i, o : Value;\ a, a' : State \mid \\
\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \bullet \\
\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto (a', is, os \frown \langle o \rangle)\}\}
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \\
\quad \exists\, as' : \mathrm{seq}\, State_\perp \mid \#as' = (\, \#ins + 2\,) \bullet \\
\qquad head\ as' = a_i \ \wedge\ last\ as' = a_f \\
\qquad \wedge \\
\qquad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\quad (as'.r, (ins \frown \langle in \rangle).r) \mapsto (as'.(r+1), outs.r) \\
\qquad\qquad \in ops\ ((prog \frown \langle name \rangle).r) \\
\qquad \wedge \\
\qquad \#(ins \frown \langle in \rangle) > \#outs \\
\qquad\quad \Rightarrow \\
\qquad\qquad (as'.(\#outs+1), (ins \frown \langle in \rangle).(\#outs+1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \mathrm{dom}\ ops\ ((prog \frown \langle name \rangle).(\#outs+1)) \\
\qquad \wedge \\
\qquad \forall\, r : (\, \#outs + 2\,) \mathrel{..} \#as' \ \bullet\ as'.r \ = \ \perp \\
\qquad \wedge \\
\qquad (as'.(\#as' - 1), in) \notin \mathrm{dom}(ops\ name)
\end{array}
$$

and hence have completed our inductive step.

We have show that $0 \in \mathcal{T}$ *and that*

$$\forall\, k : \mathbb{N} \bullet k \in \mathcal{T} \Rightarrow (k+1) \in \mathcal{T}$$

and therefore conclude that all natural numbers lie in the truth set $\mathcal{T}$ and hence that our desired result holds.

We are now able to present the proof for the main theorem of this chapter.

Theorem 3

$$\begin{array}{l}\forall A : CDT[AState, Name, Value];\ C : CDT'[CState, Name, Value] \bullet \\ \quad A \sqsubseteq_{relation} C \\ \qquad \Leftrightarrow \\ \qquad\quad (\, process_{io} A\,) \sqsubseteq_{single} (\, process_{io} C\,)\end{array}$$

Proof

Let A and C be communicating data types respectively of types $CDT[AState, Name, Value]$ and $CDT'[CState, Name, Value]$. Considering the left-hand side and expanding out our definitions we reason as follows.

$$A \sqsubseteq_{relation} C$$

$$\Leftrightarrow \qquad \text{[}def^n \text{ of } \sqsubseteq_{relation}\text{]}$$

$$\forall prog : \operatorname{seq} Name \bullet (\, relation\ C\ prog\,) \subseteq (\, relation\ A\ prog\,)$$

$$\Leftrightarrow \qquad \text{[}def^n \text{ of } relation\text{]}$$

$$\begin{array}{l}\forall prog : \operatorname{seq} Name \bullet \\ \quad relation_s\,(\, simplify\ C\,)\ prog \subseteq relation_s\,(\, simplify\ A\,)\ prog\end{array}$$

$$\Leftrightarrow \qquad \text{[}def^n s \text{ of } relation_s \text{ and } simplify \text{ and } final^{\top}\text{]}$$

$$\begin{array}{l}\forall prog : \operatorname{seq} Name \bullet \\ \quad \{is : \operatorname{seq} Value;\ c_i : init' \bullet is \mapsto (c_i, is, \langle\rangle)\} \\ \qquad ⨾ \\ \qquad run\ prog\ \{n : \operatorname{dom} ops' \bullet \\ \qquad\qquad n \mapsto \{is, os : \operatorname{seq} Value;\ i, o : Value; \\ \qquad\qquad\qquad c, c' : CState \mid \\ \qquad\qquad\qquad (c, i) \mapsto (c', o) \in ops'\ n \bullet \\ \qquad\qquad\qquad (c, \langle i\rangle \frown is, os) \mapsto \\ \qquad\qquad\qquad\quad (c', is, os \frown \langle o\rangle)\}\} \\ \qquad\quad ⨾ \\ \qquad\quad \{os : \operatorname{seq} Value;\ c : CState_{\perp} \bullet (c, \langle\rangle, os) \mapsto os\} \\ \quad \subseteq \\ \quad \{is : \operatorname{seq} Value;\ a_i : init \bullet is \mapsto (a_i, is, \langle\rangle)\} \\ \qquad ⨾ \\ \qquad run\ prog\ \{n : \operatorname{dom} ops \bullet \\ \qquad\qquad n \mapsto \{is, os : \operatorname{seq} Value;\ i, o : Value; \\ \qquad\qquad\qquad a, a' : AState \mid \\ \qquad\qquad\qquad (a, i) \mapsto (c', o) \in ops\ n \bullet \\ \qquad\qquad\qquad (a, \langle i\rangle \frown is, os) \mapsto \\ \qquad\qquad\qquad\quad (a', is, os \frown \langle o\rangle)\}\} \\ \qquad\quad ⨾ \\ \qquad\quad \{os : \operatorname{seq} Value;\ a : AState_{\perp} \bullet (a, \langle\rangle, os) \mapsto os\}\end{array}$$

$\Leftrightarrow$ *[set theory]*

$$
\begin{array}{l}
\forall\, prog : \operatorname{seq} Name;\ ins, outs : \operatorname{seq} Value \bullet \\
\quad ins \mapsto outs \\
\qquad \in \\
\qquad \{is : \operatorname{seq} Value;\ c_i : init' \bullet is \mapsto (c_i, is, \langle\rangle)\} \\
\qquad\quad \mathbin{⨾} \\
\qquad\quad run\ prog\ \{n : \operatorname{dom} ops' \bullet \\
\qquad\qquad n \mapsto \{is, os : \operatorname{seq} Value;\ i, o : Value; \\
\qquad\qquad\qquad c, c' : CState \mid \\
\qquad\qquad\qquad (c, i) \mapsto (c', o) \in ops'\ n \bullet \\
\qquad\qquad\qquad (c, \langle i\rangle \frown is, os) \mapsto \\
\qquad\qquad\qquad\quad (c', is, os \frown \langle o\rangle)\}\} \\
\qquad\qquad \mathbin{⨾} \\
\qquad\qquad \{os : \operatorname{seq} Value;\ c : CState_{\perp} \bullet (c, \langle\rangle, os) \mapsto os\} \\
\quad \Rightarrow \\
\quad ins \mapsto outs \\
\qquad \in \\
\qquad \{is : \operatorname{seq} Value;\ a_i : init \bullet is \mapsto (a_i, is, \langle\rangle)\} \\
\qquad\quad \mathbin{⨾} \\
\qquad\quad run\ prog\ \{n : \operatorname{dom} ops \bullet \\
\qquad\qquad n \mapsto \{is, os : \operatorname{seq} Value;\ i, o : Value; \\
\qquad\qquad\qquad a, a' : AState \mid \\
\qquad\qquad\qquad (a, i) \mapsto (c', o) \in ops\ n \bullet \\
\qquad\qquad\qquad (a, \langle i\rangle \frown is, os) \mapsto \\
\qquad\qquad\qquad\quad (a', is, os \frown \langle o\rangle)\}\} \\
\qquad\qquad \mathbin{⨾} \\
\qquad\qquad \{os : \operatorname{seq} Value;\ a : AState_{\perp} \bullet (a, \langle\rangle, os) \mapsto os\}
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill [\mathit{def}^n \ \mathit{of} \ {}_{9}^{\circ}] \\
\quad \forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \bullet \\
\qquad \exists\, c_i : init';\ c_f : CState_{\perp};\ is, os, is', os, : \mathrm{seq}\, Value \bullet \\
\qquad\quad ins \mapsto (c_i, is, os) \ \in \\
\qquad\qquad \{is : \mathrm{seq}\, Value;\ c_i : CState \bullet is \mapsto (c_i, is, \langle\rangle)\} \\
\qquad\quad \wedge \\
\qquad\quad (c_i, is, os) \mapsto (c_f, is', os') \ \in \\
\qquad\qquad run\, prog\, \{n : \mathrm{dom}\, ops' \bullet \\
\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\, Value;\ i, o : Value; \\
\qquad\qquad\qquad\qquad c, c' : CState \mid \\
\qquad\qquad\qquad\qquad (c, i) \mapsto (c', o) \in ops'\, n \bullet \\
\qquad\qquad\qquad\quad (c, \langle i \rangle \frown is, os) \mapsto \\
\qquad\qquad\qquad\qquad (c', is, os \frown \langle o \rangle)\}\} \\
\qquad\quad \wedge \\
\qquad\quad (c_f, is', os') \mapsto outs \ \in \\
\qquad\qquad \{os : \mathrm{seq}\, Value;\ c : CState_{\perp} \bullet (c, \langle\rangle, os) \mapsto os\} \\
\qquad \Rightarrow \\
\qquad \exists\, a_i : init;\ a_f : AState_{\perp};\ is, os, is', os, : \mathrm{seq}\, Value \bullet \\
\qquad\quad ins \mapsto (a_i, is, os) \ \in \\
\qquad\qquad \{is : \mathrm{seq}\, Value;\ a_i : AState \bullet is \mapsto (c_i, is, \langle\rangle)\} \\
\qquad\quad \wedge \\
\qquad\quad (a_i, is, os) \mapsto (a_f, is', os') \ \in \\
\qquad\qquad run\, prog\, \{n : \mathrm{dom}\, ops \bullet \\
\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\, Value;\ i, o : Value; \\
\qquad\qquad\qquad\qquad a, a' : AState \mid \\
\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\, n \bullet \\
\qquad\qquad\qquad\quad (a, \langle i \rangle \frown is, os) \mapsto \\
\qquad\qquad\qquad\qquad (a', is, os \frown \langle o \rangle)\}\} \\
\qquad\quad \wedge \\
\qquad\quad (a_f, is', os') \mapsto outs \ \in \\
\qquad\qquad \{os : \mathrm{seq}\, Value;\ a : AState_{\perp} \bullet (a, \langle\rangle, os) \mapsto os\}
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[set theory]} \\
\quad \forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \bullet \\
\qquad \exists\, c_i : init';\ c_f : CState_\perp;\ is, os, is', os, : \mathrm{seq}\, Value \bullet \\
\qquad\quad is = ins \ \wedge\ os = \langle\rangle \\
\qquad\quad \wedge \\
\qquad\quad (c_i, is, os) \mapsto (c_f, is', os') \ \in \\
\qquad\qquad run\, prog\, \{n : \mathrm{dom}\, ops' \bullet \\
\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\, Value;\ i, o : Value; \\
\qquad\qquad\qquad\qquad c, c' : CState \mid \\
\qquad\qquad\qquad\qquad (c, i) \mapsto (c', o) \in ops'\, n \bullet \\
\qquad\qquad\qquad\qquad (c, \langle i \rangle \frown is, os) \mapsto \\
\qquad\qquad\qquad\qquad\quad (c', is, os \frown \langle o \rangle)\}\} \\
\qquad\quad \wedge \\
\qquad\quad is' = \langle\rangle \wedge\ os' = outs \\
\qquad \Rightarrow \\
\qquad \exists\, a_i : init;\ a_f : AState_\perp;\ is, os, is', os, : \mathrm{seq}\, Value \bullet \\
\qquad\quad is = ins \ \wedge\ os = \langle\rangle \\
\qquad\quad \wedge \\
\qquad\quad (a_i, is, os) \mapsto (a_f, is', os') \ \in \\
\qquad\qquad run\, prog\, \{n : \mathrm{dom}\, ops \bullet \\
\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\, Value;\ i, o : Value; \\
\qquad\qquad\qquad\qquad a, a' : AState \mid \\
\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\, n \bullet \\
\qquad\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto \\
\qquad\qquad\qquad\qquad\quad (a', is, os \frown \langle o \rangle)\}\} \\
\qquad\quad \wedge \\
\qquad\quad is' = \langle\rangle \wedge\ os' = outs
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[one point rule]} \\
\quad \forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \bullet \\
\qquad \exists\, c_i : init';\ c_f : CState_\perp \bullet \\
\qquad\quad (c_i, ins, \langle\rangle) \mapsto (c_f, \langle\rangle, outs) \\
\qquad\qquad \in \\
\qquad\qquad\quad run\, prog\, \{n : \mathrm{dom}\, ops' \bullet \\
\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\, Value;\ i, o : Value; \\
\qquad\qquad\qquad\qquad c, c' : CState \mid \\
\qquad\qquad\qquad\qquad (c, i) \mapsto (c', o) \in ops'\, n \bullet \\
\qquad\qquad\qquad\qquad (c, \langle i \rangle \frown is, os) \mapsto \\
\qquad\qquad\qquad\qquad\quad (c', is, os \frown \langle o \rangle)\}\} \\
\qquad \Rightarrow \\
\qquad \exists\, a_i : init;\ a_f : AState_\perp \bullet \\
\qquad\quad (a_i, ins, \langle\rangle) \mapsto (a_f, \langle\rangle, outs) \\
\qquad\qquad \in \\
\qquad\qquad\quad run\, prog\, \{n : \mathrm{dom}\, ops \bullet \\
\qquad\qquad\qquad n \mapsto \{is, os : \mathrm{seq}\, Value;\ i, o : Value; \\
\qquad\qquad\qquad\qquad a, a' : AState \mid \\
\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\, n \bullet \\
\qquad\qquad\qquad\qquad (a, \langle i \rangle \frown is, os) \mapsto \\
\qquad\qquad\qquad\qquad\quad (a', is, os \frown \langle o \rangle)\}\}
\end{array}
$$

$\Leftrightarrow$ *[Lemma 2]*

$$
\begin{array}{l}
\forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \bullet \\
\quad \exists\, c_i : init';\ c_f : CState_\perp \bullet \\
\qquad \exists\, cs : \mathrm{seq}\, CState_\perp \mid \#cs = (\#ins + 1) \bullet \\
\qquad\quad head\ cs = c_i \ \wedge\ last\ cs = c_f \\
\qquad\quad \wedge \\
\qquad\quad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\quad \wedge \\
\qquad\quad \#ins > \#outs \\
\qquad\qquad \Rightarrow \\
\qquad\qquad\quad (cs.(\#outs+1), ins.(\#outs+1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \mathrm{dom}\, ops\,(prog.(\#outs+1)) \\
\qquad\quad \wedge \\
\qquad\quad \forall\, r : (\#outs+2) \mathrel{..} \#cs \ \bullet\ cs.r = \perp \\
\quad \Rightarrow \\
\qquad \exists\, a_i : init;\ a_f : AState_\perp \bullet \\
\qquad\quad \exists\, as : \mathrm{seq}\, AState_\perp \mid \#as = (\#ins + 1) \bullet \\
\qquad\qquad head\ as = a_i \ \wedge\ last\ as = a_f \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \wedge \\
\qquad\qquad \#ins > \#outs \\
\qquad\qquad\quad \Rightarrow \\
\qquad\qquad\qquad (as.(\#outs+1), ins.(\#outs+1)) \\
\qquad\qquad\qquad\quad \notin \\
\qquad\qquad\qquad\qquad \mathrm{dom}\, ops\,(prog.(\#outs+1)) \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, r : (\#outs+2) \mathrel{..} \#as \ \bullet\ as.r = \perp
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[property of } \exists\textit{]} \\
\quad \forall\, prog : \operatorname{seq} Name;\ ins, outs : \operatorname{seq} Value \bullet \\
\qquad \exists\, c_i : init';\ c_f : CState_{\perp};\ cs : \operatorname{seq} CState_{\perp} \mid \#cs = (\#ins + 1) \bullet \\
\qquad\qquad head\ cs = c_i \ \wedge\ last\ cs = c_f \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, r : 1 \,..\, \#outs \bullet \\
\qquad\qquad\quad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \wedge \\
\qquad\qquad \#ins > \#outs \\
\qquad\qquad\quad \Rightarrow \\
\qquad\qquad\qquad (cs.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\quad \operatorname{dom} ops\,(prog.(\#outs + 1)) \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, r : (\#outs + 2) \,..\, \#cs \ \bullet\ cs.r = \perp \\
\qquad\quad \Rightarrow \\
\qquad\qquad \exists\, a_i : init;\ a_f : AState_{\perp};\ as : \operatorname{seq} AState_{\perp} \mid \\
\qquad\qquad\qquad \#as = (\#ins + 1) \bullet \\
\qquad\qquad\quad head\ as = a_i \ \wedge\ last\ as = a_f \\
\qquad\qquad\quad \wedge \\
\qquad\qquad\quad \forall\, r : 1 \,..\, \#outs \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\quad \wedge \\
\qquad\qquad\quad \#ins > \#outs \\
\qquad\qquad\qquad \Rightarrow \\
\qquad\qquad\qquad\quad (as.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad\qquad\quad \notin \\
\qquad\qquad\qquad\qquad\qquad \operatorname{dom} ops\,(prog.(\#outs + 1)) \\
\qquad\qquad\quad \wedge \\
\qquad\qquad\quad \forall\, r : (\#outs + 2) \,..\, \#as \ \bullet\ as.r = \perp
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[one point rule]} \\
\quad \forall\, prog : \operatorname{seq} Name;\ ins, outs : \operatorname{seq} Value \bullet \\
\qquad \exists\, cs : \operatorname{seq} CState_{\perp} \mid \#cs = (\#ins + 1) \land head\ cs \in init' \bullet \\
\qquad\qquad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad\qquad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \land \\
\qquad\qquad \#ins > \#outs \\
\qquad\qquad\qquad \Rightarrow \\
\qquad\qquad\qquad\quad (cs.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\qquad\quad \operatorname{dom} ops\,(prog.(\#outs + 1)) \\
\qquad\qquad \land \\
\qquad\qquad \forall\, r : (\#outs + 2) \mathrel{..} \#cs \ \bullet\ cs.r = \perp \\
\qquad\quad \Rightarrow \\
\qquad\qquad \exists\, as : \operatorname{seq} AState_{\perp} \mid \#as = (\#ins + 1) \land head\ as \in init \bullet \\
\qquad\qquad\quad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad \#ins > \#outs \\
\qquad\qquad\qquad \Rightarrow \\
\qquad\qquad\qquad\quad (as.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\qquad\quad \operatorname{dom} ops\,(prog.(\#outs + 1)) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad \forall\, r : (\#outs + 2) \mathrel{..} \#as \ \bullet\ as.r = \perp
\end{array}
$$

Observing that it is always the length of outs is at most the length of ins we continue reasoning as follows:

$$
\begin{array}{ll}
\Leftrightarrow & \textit{[property of } \forall \textit{]} \\
\quad \forall\, prog : \operatorname{seq} Name;\ ins, outs : \operatorname{seq} Value \mid \# ins = \# outs \bullet & \\
\qquad \exists\, cs : \operatorname{seq} CState_{\perp} \mid \# cs = (\# ins + 1) \land head\ cs \in init' \bullet & \\
\qquad\qquad \forall\, r : 1 \mathinner{..} \# outs \bullet & \\
\qquad\qquad\quad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) & \\
\qquad\qquad \land & \\
\qquad\qquad \# ins > \# outs & \\
\qquad\qquad\quad \Rightarrow & \\
\qquad\qquad\qquad (cs.(\# outs + 1), ins.(\# outs + 1)) & \\
\qquad\qquad\qquad\qquad \notin & \\
\qquad\qquad\qquad\qquad\quad \operatorname{dom} ops\,(prog.(\# outs + 1)) & \\
\qquad\qquad \land & \\
\qquad\qquad \forall\, r : (\# outs + 2) \mathinner{..} \# cs \ \bullet\ cs.r = \perp & \\
\qquad\quad \Rightarrow & \\
\qquad\qquad \exists\, as : \operatorname{seq} AState_{\perp} \mid \# as = (\# ins + 1) \land head\ as \in init \bullet & \\
\qquad\qquad\quad \forall\, r : 1 \mathinner{..} \# outs \bullet & \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) & \\
\qquad\qquad\quad \land & \\
\qquad\qquad\quad \# ins > \# outs & \\
\qquad\qquad\qquad \Rightarrow & \\
\qquad\qquad\qquad\quad (as.(\# outs + 1), ins.(\# outs + 1)) & \\
\qquad\qquad\qquad\qquad\quad \notin & \\
\qquad\qquad\qquad\qquad\qquad \operatorname{dom} ops\,(prog.(\# outs + 1)) & \\
\qquad\qquad\quad \land & \\
\qquad\qquad\quad \forall\, r : (\# outs + 2) \mathinner{..} \# as \ \bullet\ as.r = \perp & \\
\quad \land & \\
\quad \forall\, prog : \operatorname{seq} Name;\ ins, outs : \operatorname{seq} Value \mid \# outs < \# ins \bullet & \\
\qquad \exists\, cs : \operatorname{seq} CState_{\perp} \mid \# cs = (\# ins + 1) \land head\ cs \in init' \bullet & \\
\qquad\qquad \forall\, r : 1 \mathinner{..} \# outs \bullet & \\
\qquad\qquad\quad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) & \\
\qquad\qquad \land & \\
\qquad\qquad \# ins > \# outs & \\
\qquad\qquad\quad \Rightarrow & \\
\qquad\qquad\qquad (cs.(\# outs + 1), ins.(\# outs + 1)) & \\
\qquad\qquad\qquad\qquad \notin & \\
\qquad\qquad\qquad\qquad\quad \operatorname{dom} ops\,(prog.(\# outs + 1)) & \\
\qquad\qquad \land & \\
\qquad\qquad \forall\, r : (\# outs + 2) \mathinner{..} \# cs \ \bullet\ cs.r = \perp & \\
\qquad\quad \Rightarrow & \\
\qquad\qquad \exists\, as : \operatorname{seq} AState_{\perp} \mid \# as = (\# ins + 1) \land head\ as \in init \bullet & \\
\qquad\qquad\quad \forall\, r : 1 \mathinner{..} \# outs \bullet & \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) & \\
\qquad\qquad\quad \land & \\
\qquad\qquad\quad \# ins > \# outs & \\
\qquad\qquad\qquad \Rightarrow & \\
\qquad\qquad\qquad\quad (as.(\# outs + 1), ins.(\# outs + 1)) & \\
\qquad\qquad\qquad\qquad\quad \notin & \\
\qquad\qquad\qquad\qquad\qquad \operatorname{dom} ops\,(prog.(\# outs + 1)) & \\
\qquad\qquad\quad \land & \\
\qquad\qquad\quad \forall\, r : (\# outs + 2) \mathinner{..} \# as \ \bullet\ as.r = \perp &
\end{array}
$$

$\Leftrightarrow$ *[property of $\forall$]*

$$
\begin{array}{l}
\forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \mid \#ins = \#outs \bullet \\
\quad \exists\, cs : \mathrm{seq}\, CState_{\perp} \mid \#cs = (\#ins + 1) \land head\ cs \in init' \bullet \\
\quad\quad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\quad\quad\quad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\quad\quad \land \\
\quad\quad \mathrm{false} \\
\quad\quad\quad \Rightarrow \\
\quad\quad\quad\quad (cs.(\#outs + 1), ins.(\#outs + 1)) \\
\quad\quad\quad\quad\quad \notin \\
\quad\quad\quad\quad\quad \mathrm{dom}\, ops\,(prog.(\#outs + 1)) \\
\quad\quad \land \\
\quad\quad \forall\, r : (\#outs + 2) \mathrel{..} \#cs \;\bullet\; cs.r = \perp \\
\quad \Rightarrow \\
\quad\quad \exists\, as : \mathrm{seq}\, AState_{\perp} \mid \#as = (\#ins + 1) \land head\ as \in init \bullet \\
\quad\quad\quad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\quad\quad\quad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\quad\quad\quad \land \\
\quad\quad\quad \mathrm{false} \\
\quad\quad\quad\quad \Rightarrow \\
\quad\quad\quad\quad\quad (as.(\#outs + 1), ins.(\#outs + 1)) \\
\quad\quad\quad\quad\quad\quad \notin \\
\quad\quad\quad\quad\quad\quad \mathrm{dom}\, ops\,(prog.(\#outs + 1)) \\
\quad\quad\quad \land \\
\quad\quad\quad \forall\, r : (\#outs + 2) \mathrel{..} \#as \;\bullet\; as.r = \perp \\
\land \\
\forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \mid \#outs < \#ins \bullet \\
\quad \exists\, cs : \mathrm{seq}\, CState_{\perp} \mid \#cs = (\#ins + 1) \land head\ cs \in init' \bullet \\
\quad\quad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\quad\quad\quad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\quad\quad \land \\
\quad\quad \mathrm{true} \\
\quad\quad\quad \Rightarrow \\
\quad\quad\quad\quad (cs.(\#outs + 1), ins.(\#outs + 1)) \\
\quad\quad\quad\quad\quad \notin \\
\quad\quad\quad\quad\quad \mathrm{dom}\, ops\,(prog.(\#outs + 1)) \\
\quad\quad \land \\
\quad\quad \forall\, r : (\#outs + 2) \mathrel{..} \#cs \;\bullet\; cs.r = \perp \\
\quad \Rightarrow \\
\quad\quad \exists\, as : \mathrm{seq}\, AState_{\perp} \mid \#as = (\#ins + 1) \land head\ as \in init \bullet \\
\quad\quad\quad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\quad\quad\quad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\quad\quad\quad \land \\
\quad\quad\quad \mathrm{true} \\
\quad\quad\quad\quad \Rightarrow \\
\quad\quad\quad\quad\quad (as.(\#outs + 1), ins.(\#outs + 1)) \\
\quad\quad\quad\quad\quad\quad \notin \\
\quad\quad\quad\quad\quad\quad \mathrm{dom}\, ops\,(prog.(\#outs + 1)) \\
\quad\quad\quad \land \\
\quad\quad\quad \forall\, r : (\#outs + 2) \mathrel{..} \#as \;\bullet\; as.r = \perp
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[property of } \Rightarrow\textit{]} \\
\quad \forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \mid \#ins = \#outs \bullet \\
\qquad \exists\, cs : \mathrm{seq}\, CState_{\perp} \mid \#cs = (\#ins + 1) \land head\ cs \in init' \bullet \\
\qquad\qquad \forall\, r : 1 \mathinner{..} \#outs \bullet \\
\qquad\qquad\qquad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \land \\
\qquad\qquad \mathrm{true} \\
\qquad\qquad \land \\
\qquad\qquad \forall\, r : (\#outs + 2) \mathinner{..} \#cs \ \bullet\ cs.r = \perp \\
\qquad\quad \Rightarrow \\
\qquad\qquad \exists\, as : \mathrm{seq}\, AState_{\perp} \mid \#as = (\#ins + 1) \land head\ as \in init \bullet \\
\qquad\qquad\quad \forall\, r : 1 \mathinner{..} \#outs \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad \mathrm{true} \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad \forall\, r : (\#outs + 2) \mathinner{..} \#as \ \bullet\ as.r = \perp \\
\quad \land \\
\quad \forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \mid \#outs < \#ins \bullet \\
\qquad \exists\, cs : \mathrm{seq}\, CState_{\perp} \mid \#cs = (\#ins + 1) \land head\ cs \in init' \bullet \\
\qquad\qquad \forall\, r : 1 \mathinner{..} \#outs \bullet \\
\qquad\qquad\qquad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \land \\
\qquad\qquad (cs.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \mathrm{dom}\ ops\,(prog.(\#outs + 1)) \\
\qquad\qquad \land \\
\qquad\qquad \forall\, r : (\#outs + 2) \mathinner{..} \#cs \ \bullet\ cs.r = \perp \\
\qquad\quad \Rightarrow \\
\qquad\qquad \exists\, as : \mathrm{seq}\, AState_{\perp} \mid \#as = (\#ins + 1) \land head\ as \in init \bullet \\
\qquad\qquad\quad \forall\, r : 1 \mathinner{..} \#outs \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad (as.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad\quad \notin \\
\qquad\qquad\qquad\qquad \mathrm{dom}\ ops\,(prog.(\#outs + 1)) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad \forall\, r : (\#outs + 2) \mathinner{..} \#as \ \bullet\ as.r = \perp
\end{array}
$$

$\Leftrightarrow$ *[property of $\wedge$]*

$$
\begin{array}{l}
\forall\, prog : \operatorname{seq} Name;\ ins, outs : \operatorname{seq} Value \mid \#ins = \#outs \bullet \\
\quad \exists\, cs : \operatorname{seq} CState_{\perp} \mid \#cs = (\#ins + 1) \wedge head\ cs \in init' \bullet \\
\qquad\qquad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad\quad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, r : (\#outs + 2) \mathrel{..} \#cs \bullet cs.r = \perp \\
\qquad \Rightarrow \\
\qquad\quad \exists\, as : \operatorname{seq} AState_{\perp} \mid \#as = (\#ins + 1) \wedge head\ as \in init \bullet \\
\qquad\qquad\quad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\quad \wedge \\
\qquad\qquad\quad \forall\, r : (\#outs + 2) \mathrel{..} \#as \bullet as.r = \perp \\
\wedge \\
\forall\, prog : \operatorname{seq} Name;\ ins, outs : \operatorname{seq} Value \mid \#outs < \#ins \bullet \\
\quad \exists\, cs : \operatorname{seq} CState_{\perp} \mid \#cs = (\#ins + 1) \wedge head\ cs \in init' \bullet \\
\qquad\qquad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad\quad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \wedge \\
\qquad\qquad (cs.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \operatorname{dom} ops\,(prog.(\#outs + 1)) \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, r : (\#outs + 2) \mathrel{..} \#cs \bullet cs.r = \perp \\
\qquad \Rightarrow \\
\qquad\quad \exists\, as : \operatorname{seq} AState_{\perp} \mid \#as = (\#ins + 1) \wedge head\ as \in init \bullet \\
\qquad\qquad\quad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\quad \wedge \\
\qquad\qquad\quad (as.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad\quad \notin \\
\qquad\qquad\qquad\qquad \operatorname{dom} ops\,(prog.(\#outs + 1)) \\
\qquad\qquad\quad \wedge \\
\qquad\qquad\quad \forall\, r : (\#outs + 2) \mathrel{..} \#as \bullet as.r = \perp
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[substitution]} \\
\quad \forall\, prog : \operatorname{seq} Name;\ ins, outs : \operatorname{seq} Value \mid \# ins = \# outs \bullet \\
\qquad \exists\, cs : \operatorname{seq} CState_{\bot} \mid \# cs = (\# ins + 1) \land head\ cs \in init' \bullet \\
\qquad\qquad \forall\, r : 1 \mathrel{..} (\# cs - 1) \bullet \\
\qquad\qquad\qquad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \land \\
\qquad\qquad \forall\, r : (\# cs + 1) \mathrel{..} \# cs \bullet cs.r = \bot \\
\qquad\quad \Rightarrow \\
\qquad\qquad \exists\, as : \operatorname{seq} AState_{\bot} \mid \# as = (\# ins + 1) \land head\ as \in init \bullet \\
\qquad\qquad\quad \forall\, r : 1 \mathrel{..} (\# cs - 1) \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad \forall\, r : (\# as + 1) \mathrel{..} \# as \bullet as.r = \bot \\
\quad \land \\
\quad \forall\, prog : \operatorname{seq} Name;\ ins, outs : \operatorname{seq} Value \mid \# outs < \# ins \bullet \\
\qquad \exists\, cs : \operatorname{seq} CState_{\bot} \mid \# cs = (\# ins + 1) \land head\ cs \in init' \bullet \\
\qquad\qquad \forall\, r : 1 \mathrel{..} \# outs \bullet \\
\qquad\qquad\qquad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \land \\
\qquad\qquad (cs.(\# outs + 1), ins.(\# outs + 1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \operatorname{dom} ops\,(prog.(\# outs + 1)) \\
\qquad\qquad \land \\
\qquad\qquad \forall\, r : (\# outs + 2) \mathrel{..} \# cs \bullet cs.r = \bot \\
\qquad\quad \Rightarrow \\
\qquad\qquad \exists\, as : \operatorname{seq} AState_{\bot} \mid \# as = (\# ins + 1) \land head\ as \in init \bullet \\
\qquad\qquad\quad \forall\, r : 1 \mathrel{..} \# outs \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad (as.(\# outs + 1), ins.(\# outs + 1)) \\
\qquad\qquad\qquad\quad \notin \\
\qquad\qquad\qquad\qquad \operatorname{dom} ops\,(prog.(\# outs + 1)) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad \forall\, r : (\# outs + 2) \mathrel{..} \# as \bullet as.r = \bot
\end{array}
$$

$\Leftrightarrow$ *[property of ..]*

$$
\begin{array}{l}
\forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \mid \#ins = \#outs \bullet \\
\quad \exists\, cs : \mathrm{seq}\, CState_{\perp} \mid \#cs = (\#ins + 1) \land head\ cs \in init' \bullet \\
\qquad\quad \forall\, r : 1 \mathrel{..} (\#cs - 1) \bullet \\
\qquad\qquad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\quad \land \\
\qquad\quad \forall\, r : \emptyset \bullet cs.r = \perp \\
\qquad \Rightarrow \\
\qquad\quad \exists\, as : \mathrm{seq}\, AState_{\perp} \mid \#as = (\#ins + 1) \land head\ as \in init \bullet \\
\qquad\qquad \forall\, r : 1 \mathrel{..} (\#as - 1) \bullet \\
\qquad\qquad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \land \\
\qquad\qquad \forall\, r : \emptyset \bullet as.r = \perp \\
\land \\
\forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \mid \#outs < \#ins \bullet \\
\quad \exists\, cs : \mathrm{seq}\, CState_{\perp} \mid \#cs = (\#ins + 1) \land head\ cs \in init' \bullet \\
\qquad\quad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\quad \land \\
\qquad\quad (cs.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\quad \notin \\
\qquad\qquad\qquad \mathrm{dom}\, ops\,(prog.(\#outs + 1)) \\
\qquad\quad \land \\
\qquad\quad \forall\, r : (\#outs + 2) \mathrel{..} \#cs \bullet cs.r = \perp \\
\qquad \Rightarrow \\
\qquad\quad \exists\, as : \mathrm{seq}\, AState_{\perp} \mid \#as = (\#ins + 1) \land head\ as \in init \bullet \\
\qquad\qquad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \land \\
\qquad\qquad (as.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \mathrm{dom}\, ops\,(prog.(\#outs + 1)) \\
\qquad\qquad \land \\
\qquad\qquad \forall\, r : (\#outs + 2) \mathrel{..} \#as \bullet as.r = \perp
\end{array}
$$

$\Leftrightarrow$ *[property of $\forall$]*

$$
\begin{array}{l}
\forall\, prog : \mathrm{seq}\, Name;\ \ ins, outs : \mathrm{seq}\ Value \mid \#ins = \#outs \bullet \\
\quad \exists\, cs : \mathrm{seq}\, state' \mid \#cs = (\#ins + 1) \land head\ cs \in init' \bullet \\
\qquad\qquad \forall\, r : 1 \mathrel{..} (\#cs - 1) \bullet \\
\qquad\qquad\quad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \land \\
\qquad\qquad \mathrm{true} \\
\qquad \Rightarrow \\
\qquad\quad \exists\, as : \mathrm{seq}\, state \mid \#as = (\#ins + 1) \land head\ as \in init \bullet \\
\qquad\qquad\quad \forall\, r : 1 \mathrel{..} (\#as - 1) \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad \mathrm{true} \\
\land \\
\forall\, prog : \mathrm{seq}\, Name;\ \ ins, outs : \mathrm{seq}\ Value \mid \#outs < \#ins \bullet \\
\quad \exists\, cs : \mathrm{seq}\, CState_{\perp} \mid \#cs = (\#ins + 1) \land head\ cs \in init' \bullet \\
\qquad\qquad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad\quad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \land \\
\qquad\qquad (cs.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \mathrm{dom}\ ops\,(prog.(\#outs + 1)) \\
\qquad\qquad \land \\
\qquad\qquad \forall\, r : (\#outs + 2) \mathrel{..} \#cs \ \bullet\ cs.r = \perp \\
\qquad \Rightarrow \\
\qquad\quad \exists\, as : \mathrm{seq}\, AState_{\perp} \mid \#as = (\#ins + 1) \land head\ as \in init \bullet \\
\qquad\qquad\quad \forall\, r : 1 \mathrel{..} \#outs \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad (as.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad\quad \notin \\
\qquad\qquad\qquad\qquad \mathrm{dom}\ ops\,(prog.(\#outs + 1)) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad \forall\, r : (\#outs + 2) \mathrel{..} \#as \ \bullet\ as.r = \perp
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[property of } \land \textit{]} \\
\quad \forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \mid \#ins = \#outs \bullet \\
\qquad \exists\, cs : \mathrm{seq}\, state' \mid \#cs = (\#ins + 1) \land head\ cs \in init' \bullet \\
\qquad\qquad \forall\, r : 1 \mathinner{\ldotp\ldotp} (\#cs - 1) \bullet \\
\qquad\qquad\qquad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\quad \Rightarrow \\
\qquad\qquad \exists\, as : \mathrm{seq}\, state \mid \#as = (\#ins + 1) \land head\ as \in init \bullet \\
\qquad\qquad\quad \forall\, r : 1 \mathinner{\ldotp\ldotp} (\#as - 1) \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\quad \land \\
\quad \forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \mid \#outs < \#ins \bullet \\
\qquad \exists\, cs : \mathrm{seq}\, CState_{\perp} \mid \#cs = (\#ins + 1) \land head\ cs \in init' \bullet \\
\qquad\qquad \forall\, r : 1 \mathinner{\ldotp\ldotp} \#outs \bullet \\
\qquad\qquad\qquad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad \land \\
\qquad\qquad (cs.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\quad \mathrm{dom}\, ops\,(prog.(\#outs + 1)) \\
\qquad\qquad \land \\
\qquad\qquad \forall\, r : (\#outs + 2) \mathinner{\ldotp\ldotp} \#cs \ \bullet\ cs.r = \perp \\
\qquad\quad \Rightarrow \\
\qquad\qquad \exists\, as : \mathrm{seq}\, AState_{\perp} \mid \#as = (\#ins + 1) \land head\ as \in init \bullet \\
\qquad\qquad\quad \forall\, r : 1 \mathinner{\ldotp\ldotp} \#outs \bullet \\
\qquad\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad (as.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad\quad \notin \\
\qquad\qquad\qquad\qquad \mathrm{dom}\, ops\,(prog.(\#outs + 1)) \\
\qquad\qquad\quad \land \\
\qquad\qquad\quad \forall\, r : (\#outs + 2) \mathinner{\ldotp\ldotp} \#as \ \bullet\ as.r = \perp
\end{array}
$$

Observing that, by construction of the function $process_{io}$, given communicating data type $A : CDT[State, Name, Value]$, and sequence of names prog and sequences of values ins and outs such that prog and ins and outs are all the same length the following is true:

$$
\begin{array}{l}
\exists\, as : \mathrm{seq}\, state \mid \#as = (\#ins + 1) \land head\ as \in init \bullet \\
\qquad \forall\, r : 1 \mathinner{\ldotp\ldotp} (\#as - 1) \bullet \\
\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\quad \Leftrightarrow \\
\quad \langle\, r : 1 \mathinner{\ldotp\ldotp} \#ins \ \bullet\ \mathrm{op}.(prog.r).(ins.r).(outs.r) \,\rangle \\
\qquad\quad \in \\
\qquad\qquad traces[\, process_{io}\ A \,]
\end{array}
$$

we continue reasoning as follows.

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[above reasoning]} \\
\quad \forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \mid \#ins = \#outs \bullet \\
\qquad \langle\, r : 1 \mathinner{\ldotp\ldotp} \#ins \ \bullet\ \mathrm{op}.(prog.r).(ins.r).(outs.r) \,\rangle \\
\qquad\qquad\qquad \in \\
\qquad\qquad\qquad\quad traces[\, process_{io}\ C\,] \\
\qquad\qquad \Rightarrow \\
\qquad\qquad\quad \langle\, r : 1 \mathinner{\ldotp\ldotp} \#ins \ \bullet\ \mathrm{op}.(prog.r).(ins.r).(outs.r) \,\rangle \\
\qquad\qquad\qquad\qquad \in \\
\qquad\qquad\qquad\qquad\quad traces[\, process_{io}\ A\,] \\
\quad \wedge \\
\quad \forall\, prog : \mathrm{seq}\, Name;\ ins, outs : \mathrm{seq}\, Value \mid \#outs < \#ins \bullet \\
\qquad \exists\, cs : \mathrm{seq}\, CState_{\perp} \mid \#cs = (\#ins + 1) \wedge head\ cs \in init' \bullet \\
\qquad\qquad\qquad \forall\, r : 1 \mathinner{\ldotp\ldotp} \#outs \bullet \\
\qquad\qquad\qquad\qquad (cs.r, ins.r) \mapsto (cs.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad (cs.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad\qquad\quad \notin \\
\qquad\qquad\qquad\qquad\qquad \mathrm{dom}\ ops\,(prog.(\#outs + 1)) \\
\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad \forall\, r : (\#outs + 2) \mathinner{\ldotp\ldotp} \#cs \ \bullet\ cs.r = \perp \\
\qquad\qquad \Rightarrow \\
\qquad\qquad\quad \exists\, as : \mathrm{seq}\, AState_{\perp} \mid \#as = (\#ins + 1) \wedge head\ as \in init \bullet \\
\qquad\qquad\qquad\quad \forall\, r : 1 \mathinner{\ldotp\ldotp} \#outs \bullet \\
\qquad\qquad\qquad\qquad\quad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad\qquad\qquad\quad \wedge \\
\qquad\qquad\qquad\quad (as.(\#outs + 1), ins.(\#outs + 1)) \\
\qquad\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\qquad\quad \mathrm{dom}\ ops\,(prog.(\#outs + 1)) \\
\qquad\qquad\qquad\quad \wedge \\
\qquad\qquad\qquad\quad \forall\, r : (\#outs + 2) \mathinner{\ldotp\ldotp} \#as \ \bullet\ as.r = \perp
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[set theory]} \\
\quad \mathit{traces}[\,\mathit{process}_{io}\ C\,] \ \subseteq\ \mathit{traces}[\,\mathit{process}_{io}\ A\,] \\
\quad \land \\
\quad \forall\, \mathit{prog} : \operatorname{seq} \mathit{Name};\ \mathit{ins}, \mathit{outs} : \operatorname{seq} \mathit{Value} \mid \#\mathit{outs} < \#\mathit{ins} \bullet \\
\qquad\quad \exists\, \mathit{cs} : \operatorname{seq} \mathit{CState}_{\perp} \mid \#\mathit{cs} = (\#\mathit{ins} + 1) \land \mathit{head}\ \mathit{cs} \in \mathit{init}' \bullet \\
\qquad\qquad\qquad\quad \forall\, r : 1 \mathrel{..} \#\mathit{outs} \bullet \\
\qquad\qquad\qquad\qquad\quad (\mathit{cs}.r, \mathit{ins}.r) \mapsto (\mathit{cs}.(r+1), \mathit{outs}.r) \in \mathit{ops}\,(\mathit{prog}.r) \\
\qquad\qquad\qquad\quad \land \\
\qquad\qquad\qquad\quad (\mathit{cs}.(\#\mathit{outs} + 1), \mathit{ins}.(\#\mathit{outs} + 1)) \\
\qquad\qquad\qquad\qquad\qquad\quad \notin \\
\qquad\qquad\qquad\qquad\qquad\qquad \operatorname{dom} \mathit{ops}\,(\mathit{prog}.(\#\mathit{outs} + 1)) \\
\qquad\qquad\qquad\quad \land \\
\qquad\qquad\qquad\quad \forall\, r : (\#\mathit{outs} + 2) \mathrel{..} \#\mathit{cs} \ \bullet\ \mathit{cs}.r = \perp \\
\qquad\qquad\quad \Rightarrow \\
\qquad\qquad\qquad \exists\, \mathit{as} : \operatorname{seq} \mathit{AState}_{\perp} \mid \#\mathit{as} = (\#\mathit{ins} + 1) \land \mathit{head}\ \mathit{as} \in \mathit{init} \bullet \\
\qquad\qquad\qquad\qquad \forall\, r : 1 \mathrel{..} \#\mathit{outs} \bullet \\
\qquad\qquad\qquad\qquad\qquad (\mathit{as}.r, \mathit{ins}.r) \mapsto (\mathit{as}.(r+1), \mathit{outs}.r) \in \mathit{ops}\,(\mathit{prog}.r) \\
\qquad\qquad\qquad\qquad \land \\
\qquad\qquad\qquad\qquad (\mathit{as}.(\#\mathit{outs} + 1), \mathit{ins}.(\#\mathit{outs} + 1)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \operatorname{dom} \mathit{ops}\,(\mathit{prog}.(\#\mathit{outs} + 1)) \\
\qquad\qquad\qquad\qquad \land \\
\qquad\qquad\qquad\qquad \forall\, r : (\#\mathit{outs} + 2) \mathrel{..} \#\mathit{as} \ \bullet\ \mathit{as}.r = \perp
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[property of single]} \\
\quad \mathit{single}[\,\mathit{process}_{io}\ C\,] \ \cap\ \{\mathit{tr} : \operatorname{seq} \mathit{Event} \bullet (\mathit{tr}, \emptyset)\} \\
\qquad\quad \subseteq \\
\qquad\qquad \mathit{single}[\,\mathit{process}_{io}\ A\,] \ \cap\ \{\mathit{tr} : \operatorname{seq} \mathit{Event} \bullet (\mathit{tr}, \emptyset)\} \\
\quad \land \\
\quad \forall\, \mathit{prog} : \operatorname{seq} \mathit{Name};\ \mathit{ins}, \mathit{outs} : \operatorname{seq} \mathit{Value} \mid \#\mathit{outs} < \#\mathit{ins} \bullet \\
\qquad\quad \exists\, \mathit{cs} : \operatorname{seq} \mathit{CState}_{\perp} \mid \#\mathit{cs} = (\#\mathit{ins} + 1) \land \mathit{head}\ \mathit{cs} \in \mathit{init}' \bullet \\
\qquad\qquad\qquad\quad \forall\, r : 1 \mathrel{..} \#\mathit{outs} \bullet \\
\qquad\qquad\qquad\qquad\quad (\mathit{cs}.r, \mathit{ins}.r) \mapsto (\mathit{cs}.(r+1), \mathit{outs}.r) \in \mathit{ops}\,(\mathit{prog}.r) \\
\qquad\qquad\qquad\quad \land \\
\qquad\qquad\qquad\quad (\mathit{cs}.(\#\mathit{outs} + 1), \mathit{ins}.(\#\mathit{outs} + 1)) \\
\qquad\qquad\qquad\qquad\qquad\quad \notin \\
\qquad\qquad\qquad\qquad\qquad\qquad \operatorname{dom} \mathit{ops}\,(\mathit{prog}.(\#\mathit{outs} + 1)) \\
\qquad\qquad\qquad\quad \land \\
\qquad\qquad\qquad\quad \forall\, r : (\#\mathit{outs} + 2) \mathrel{..} \#\mathit{cs} \ \bullet\ \mathit{cs}.r = \perp \\
\qquad\qquad\quad \Rightarrow \\
\qquad\qquad\qquad \exists\, \mathit{as} : \operatorname{seq} \mathit{AState}_{\perp} \mid \#\mathit{as} = (\#\mathit{ins} + 1) \land \mathit{head}\ \mathit{as} \in \mathit{init} \bullet \\
\qquad\qquad\qquad\qquad \forall\, r : 1 \mathrel{..} \#\mathit{outs} \bullet \\
\qquad\qquad\qquad\qquad\qquad (\mathit{as}.r, \mathit{ins}.r) \mapsto (\mathit{as}.(r+1), \mathit{outs}.r) \in \mathit{ops}\,(\mathit{prog}.r) \\
\qquad\qquad\qquad\qquad \land \\
\qquad\qquad\qquad\qquad (\mathit{as}.(\#\mathit{outs} + 1), \mathit{ins}.(\#\mathit{outs} + 1)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \notin \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \operatorname{dom} \mathit{ops}\,(\mathit{prog}.(\#\mathit{outs} + 1)) \\
\qquad\qquad\qquad\qquad \land \\
\qquad\qquad\qquad\qquad \forall\, r : (\#\mathit{outs} + 2) \mathrel{..} \#\mathit{as} \ \bullet\ \mathit{as}.r = \perp
\end{array}
$$

Next observing that, by construction of the function $process_{io}$, given communicating data type $A : CDT[State, Name, Value]$, and sequence of names prog and sequences of values ins and outs such that prog and ins are all the same length and outs is shorter than them both the following is true:

$$
\begin{array}{l}
\exists\, as : \operatorname{seq} State_{\perp} \mid \# as = (\# ins + 1) \land head\ as \in init \bullet \\
\qquad \forall\, r : 1 \mathinner{\ldotp\ldotp} \# outs \bullet \\
\qquad\qquad (as.r, ins.r) \mapsto (as.(r+1), outs.r) \in ops\,(prog.r) \\
\qquad \land \\
\qquad (as.(\# outs + 1), ins.(\# outs + 1)) \\
\qquad\qquad\quad \notin \\
\qquad\qquad\qquad \operatorname{dom} ops\,(prog.(\# outs + 1)) \\
\qquad \land \\
\qquad \forall\, r : (\# outs + 2) \mathinner{\ldotp\ldotp} \# as \;\bullet\; as.r = \perp \\
\quad \Leftrightarrow \\
\quad\; \langle r : 1 \mathinner{\ldotp\ldotp} \# outs \bullet \mathrm{op}.(prog.r).(ins.r).(outs.r) \rangle \\
\qquad \in traces[\, process_{io}\, A \,] \\
\quad\; \land \\
\quad\; \{\, \mathrm{op}.(prog.(\# outs + 1)).(ins.(\# outs + 1)).(outs.(\# outs + 1)) \,\} \\
\qquad \in \\
\qquad\quad refusals\,[\, (\, process_{io}\, A\,) \;\; / \\
\qquad\qquad\qquad\qquad \langle r : 1 \mathinner{\ldotp\ldotp} \# outs \bullet \mathrm{op}.(prog.r).(ins.r).(outs.r) \rangle \,]
\end{array}
$$

We continue reasoning as follows:

$$
\begin{array}{ll}
\Leftrightarrow & \textit{[above reasoning]} \\
\quad single[\, process_{io}\, C \,] \;\cap\; \{ tr : \operatorname{seq} Event \bullet (tr, \emptyset) \} & \\
\qquad \subseteq & \\
\qquad\quad single[\, process_{io}\, A \,] \;\cap\; \{ tr : \operatorname{seq} Event \bullet (tr, \emptyset) \} & \\
\quad \land & \\
\quad \forall\, prog : \operatorname{seq} Name;\; ins, outs : \operatorname{seq} Value \mid \# outs < \# ins \bullet & \\
\qquad \langle r : 1 \mathinner{\ldotp\ldotp} \# outs \bullet \mathrm{op}.(prog.r).(ins.r).(outs.r) \rangle & \\
\qquad\quad \in traces[\, process_{io}\, C \,] & \\
\qquad \land & \\
\qquad \{\, \mathrm{op}.(prog.(\# outs + 1)).(ins.(\# outs + 1)).(outs.(\# outs + 1)) \,\} & \\
\qquad\quad \in & \\
\qquad\qquad refusals\,[\, (\, process_{io}\, C\,) \;\; / & \\
\qquad\qquad\qquad\qquad \langle r : 1 \mathinner{\ldotp\ldotp} \# outs \bullet \mathrm{op}.(prog.r).(ins.r).(outs.r) \rangle \,] & \\
\qquad\quad \Rightarrow & \\
\qquad\qquad \langle r : 1 \mathinner{\ldotp\ldotp} \# outs \bullet \mathrm{op}.(prog.r).(ins.r).(outs.r) \rangle & \\
\qquad\qquad\quad \in traces[\, process_{io}\, A \,] & \\
\qquad\qquad \land & \\
\qquad\qquad \{\, \mathrm{op}.(prog.(\# outs + 1)).(ins.(\# outs + 1)).(outs.(\# outs + 1)) \,\} & \\
\qquad\qquad\quad \in & \\
\qquad\qquad\qquad refusals\,[\, (\, process_{io}\, A\,) \;\; / & \\
\qquad\qquad\qquad\qquad\qquad \langle r : 1 \mathinner{\ldotp\ldotp} \# outs \bullet \mathrm{op}.(prog.r).(ins.r).(outs.r) \rangle \,] &
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[def}^n \textit{ of single]} \\
\quad single[\, process_{io}\; C\,] \;\cap\; \{ tr : \mathrm{seq}\, Event \bullet (tr, \emptyset) \} \\
\qquad \subseteq \\
\qquad\quad single[\, process_{io}\; A\,] \;\cap\; \{ tr : \mathrm{seq}\, Event \bullet (tr, \emptyset) \} \\
\quad \wedge \\
\quad \forall\, prog : \mathrm{seq}\, Name;\; ins, outs : \mathrm{seq}\; Value \mid \#outs < \#ins \bullet \\
\qquad (\, \langle r : 1 \,..\, \#outs \bullet \mathrm{op}.(prog.r).(ins.r).(outs.r) \rangle, \\
\qquad\qquad \{\, \mathrm{op}.(prog.(\#outs + 1)).(ins.(\#outs + 1)).(outs.(\#outs + 1)) \,\}) \\
\qquad\qquad\quad \in \\
\qquad\qquad\qquad single[\, process_{io}\; C\,] \\
\qquad\quad \Rightarrow \\
\qquad\qquad (\, \langle r : 1 \,..\, \#outs \bullet \mathrm{op}.(prog.r).(ins.r).(outs.r) \rangle, \\
\qquad\qquad\quad \{\, \mathrm{op}.(prog.(\#outs + 1)).(ins.(\#outs + 1)).(outs.(\#outs + 1)) \,\}) \\
\qquad\qquad\qquad \in \\
\qquad\qquad\qquad\quad single[\, process_{io}\; A\,]
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[set theory]} \\
\quad single[\, process_{io}\; C\,] \;\cap\; \{ tr : \mathrm{seq}\, Event \bullet (tr, \emptyset) \} \\
\qquad \subseteq \\
\qquad\quad single[\, process_{io}\; A\,] \;\cap\; \{ tr : \mathrm{seq}\, Event \bullet (tr, \emptyset) \} \\
\quad \wedge \\
\quad single[\, process_{io}\; C\,] \;\cap\; \{ tr : \mathrm{seq}\, Event;\; e : Event \bullet (tr, \{e\}) \} \\
\qquad \subseteq \\
\qquad\quad single[\, process_{io}\; A\,] \;\cap\; \{ tr : \mathrm{seq}\, Event;\; e : Event \bullet (tr, \{e\}) \}
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \hfill \textit{[def}^n \textit{ of single]} \\
\quad single[\, process_{io}\; C\,] \\
\qquad \subseteq \\
\qquad\quad single[\, process_{io}\; A\,]
\end{array}
$$

B Combining data types

In the specification and development of large systems it will often be desirable to reason at various levels of abstraction. For instance the initial description of a system may involve just a few components and their interactions, but as we approach an implementation-level description of the system it may be convenient to break these components down into subcomponents and indeed to break the subcomponents down into subsubcomponents, as illustrated in Figure 2 below.

Having broken down components into their subcomponents it is vital that we are able to verify that these descriptions are consistent: that the combined behaviour of a collection of subcomponents satisfies the description of the behaviour of the component that they combine to form.

In this section we give a meaning to hiding a set of operations on communicating data types. We prove that our definition is consistent with hiding a corresponding set of events on the process equivalent. In addition we identify a set of healthiness conditions that a data type and a set of operations must satisfy in order to ensure that we have a congruence.

We begin the section by introducing the necessary notation in Section B.1 before going on to give our formal definition of hiding in Section B.2. Our definition of hiding may be applied only to those data types and sets of operations that satisfy a predicate *hideable* and in Section B.3 we derive and formally define this predicate. In Section B.4 we give an outline of the proof

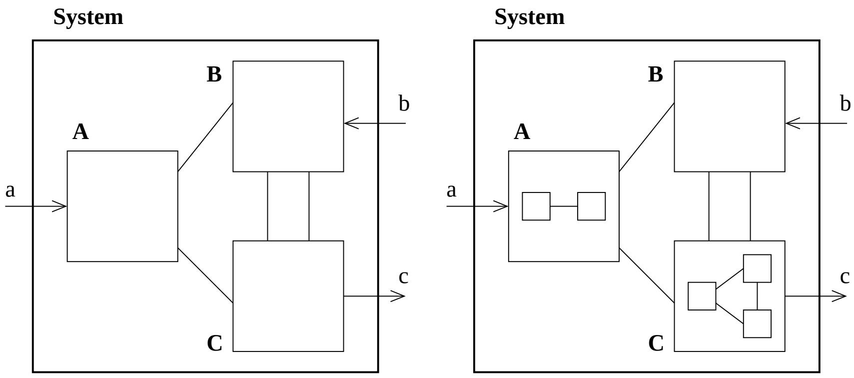

Figure 2: It may sometimes be convenient to break components within a system down into their subcomponents.

that *provided the predicate hideable is true* then hiding on abstract data types is semantically equivalent to hiding on their corresponding processes. We conclude the section with the definition of a parallel operator on our data types. We present a simple example that illustrates how the hiding operator may be used in conjunction with the parallel operator to verify that a combination of subcomponents within a system satisfy the specification of their parent component.

B.1 Notation

Our definition of hiding, and in particular our proof that the notion of hiding on data types is consistent with the notion of hiding on processes requires a not insignificant amount of additional notation and definitions. We introduce these in this section.

Stability and reachability

First we introduce the notions of stable, unstable and reachable states.

We partition the internal state space *state* of our data types into stable and unstable states. A *stable* state is one from which it is not possible to perform any of the operations to be hidden. We introduce the function *stable* which takes an abstract data type and a set of named operations on the data type which are to be hidden as its arguments and returns the set of all stable states.

$$
\begin{array}{l}
[State, Name, Value] \\
\hline
stable : CDT[State, Name, Value] \rightarrow \mathbb{P}\, Name \rightarrow \mathbb{P}\, State \\
\hline
\forall\, D : CDT[State, Name, Value];\ X : \mathbb{P}\, Name \bullet \\
\quad stable\, D\, X\ = \\
\quad\quad \{ a : state\ | \\
\quad\quad\quad \neg\, (\exists\, n : (X \cap \mathrm{dom}\, ops);\ i : Value \bullet \\
\quad\quad\quad\quad (a, i) \in \mathrm{dom}\, ops\, n)\ \}
\end{array}
$$

An *unstable* state is any state within the internal state space of the data type which is not stable. We introduce the function *unStable* which takes as its arguments an abstract data type and a set of names of operations on the data type to be hidden, and returns the set of all unstable states.

$$
\begin{tikzcd}
 & \mathrm{D},\ \mathrm{X} \arrow[dl, "hide"'] \arrow[dr, "process_{io}"] & \\
hide\,(D, X) \arrow[d, "process_{io}"'] & & process_{io}\,D,\ X \arrow[d, "\setminus"] \\
process_{io}\,(hide\,(D, X)) \arrow[rr, leftrightarrow, "single[\,]"] & & (process_{io}\,D) \setminus X
\end{tikzcd}
$$

Figure 3: The process corresponding to the data type obtained by hiding the set of operations X on data type D is semantically equivalent to the process obtained by hiding the set of events corresponding to the operations in set X on the process which corresponds to data type D.

$$
\begin{array}{l}
[State, Name, Value] \\
\hline
unStable : CDT[State, Name, Value] \rightarrow \mathbb{P}\,Name \rightarrow \mathbb{P}\,State \\
\hline
\forall\, D : CDT[State, Name, Value];\ X : \mathbb{P}\,Name \bullet \\
\qquad unStable\,D\,X = state \setminus (stable\,D\,X)
\end{array}
$$

We partition the state space further:

	unStable	**stable**
In the domain of a visible operation	transient	menu
Not in the domain of any visible operations	internal	blocked

A *transient* state is an unstable state that lies not only in the domain of one of the operations to be hidden but also in the domain of one of the remaining visible operations; whereas an *internal* state is an unstable state that does not lie in the domains of any of the visible operations.

A *menu* state is a stable state that lies in the domain of at least one of the visible operations; whereas a *blocked* state does not lie in the domain of any of the operations—hidden or visible—on the data type.

We give formal definitions for each of these types of states below:

$$
\begin{array}{l}
[State, Name, Value] \\
\hline
transient, internal, menu, blocked : \\
\quad CDT[State, Name, Value] \rightarrow \mathbb{P}\, Name \rightarrow \mathbb{P}\, State \\
\hline
\forall D : CDT[State, Name, Value];\ X : \mathbb{P}\, Name \bullet \\
\quad transient\ D\ X\ = \\
\qquad \{ a : unStable\ D\ X\ | \\
\qquad\quad \exists\, n : (\mathrm{dom}\ ops \setminus X);\ i : Value \bullet (a, i) \in \mathrm{dom}\ n \} \\
\quad internal\ D\ X\ = \\
\qquad \{ a : unStable\ D\ X\ | \\
\qquad\quad \neg\ \exists\, n : (\mathrm{dom}\ ops \setminus X);\ i : Value \bullet (a, i) \in \mathrm{dom}\ n \} \\
\quad menu\ D\ X\ = \\
\qquad \{ a : stable\ D\ X\ | \\
\qquad\quad \exists\, n : (\mathrm{dom}\ ops \setminus X);\ i : Value \bullet (a, i) \in \mathrm{dom}\ n \} \\
\quad blocked\ D\ X\ = \\
\qquad \{ a : stable\ D\ X\ | \\
\qquad\quad \neg\ \exists\, n : (\mathrm{dom}\ ops \setminus X);\ i : Value \bullet (a, i) \in \mathrm{dom}\ n \}.
\end{array}
$$

A *reachable* state is one that lies either in the set *init* or in the range of an operation from another reachable state. We define the function *reachable* which takes an abstract data type and returns the set of all reachable states.

$$
\begin{array}{l}
[State, Name, Value] \\
\hline
reachable : CDT[State, Name, Value] \rightarrow \mathbb{P}\, State \\
\hline
reachable\ = \\
\quad (\ \lambda\, D : CDT[State, Name, Value] \bullet \\
\qquad init \\
\qquad \cup \\
\qquad \{ a' : state\ | \\
\qquad\quad \exists\, a : state;\ n : \mathrm{dom}\ ops;\ i, o : Value \bullet \\
\qquad\qquad a \in reachable \\
\qquad\qquad \wedge \\
\qquad\qquad (a, i) \mapsto (a', o) \in ops\ n \}\ )
\end{array}
$$

Divergence

A *divergent* state is one from which it is possible to perform infinitely many hidden operations without a visible operation occurring. A *non-divergent* state is a state which is not divergent. The set of all non-divergent states is the union of the following sets:

- the set of all stable states;
- the set of unstable states from which it is not possible for infinitely many hidden operations to occur without a visible operation occurring.

The abstract data type obtained by hiding a set of operations on a given abstract data type is *divergence-free* if every reachable state is a non-divergent state.

We introduce the function *kNonDiv* which takes as its arguments an integer k, an abstract data type and a set of named operations. It then returns the set of unstable states from which *at* *most* k hidden operations can occur before we end up in a stable state. Note that this is not the same as saying that at most k hidden operations can occur before a visible operation must occur: it could be the case that we end up in a blocked state.

$$
\begin{array}{l}
[State, Name, Value] \\
kNonDiv : \mathbb{N} \rightarrow CDT[State, Name, Value] \rightarrow \mathbb{P}\, Name \rightarrow \mathbb{P}\, State \\
\hline
\forall\, D : CDT[State, Name, Value];\ X : \mathbb{P}\, Name \bullet \\
\quad kNonDiv\, 0\, D\, X \ = \ stable\, D\, X \\
\quad \wedge \\
\quad \forall\, k : \mathbb{N} \setminus \{0\} \bullet \\
\qquad kNonDiv\, k\, D\, X \ = \\
\qquad\quad \{ a : unStable\, D\, X \mid \\
\qquad\qquad \exists\, a' : State;\ n : (X \cap \mathrm{dom}\, ops);\ i, o : Value \bullet \\
\qquad\qquad\quad (a, i) \mapsto (a', o) \in ops\, n \\
\qquad\qquad\quad \wedge \\
\qquad\qquad\quad a' \in kNonDiv\, (k-1)\, D\, X \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, a' : State \mid (\, \exists\, n : (X \cap \mathrm{dom}\, ops);\ i, o : Value \bullet \\
\qquad\qquad\qquad\qquad (a, i) \mapsto (a', o) \in ops\, n\,) \bullet \\
\qquad\qquad\quad (\, \exists\, j : \mathbb{N} \mid 1 \leq j \leq (k-1) \bullet \\
\qquad\qquad\qquad a' \in kNonDiv\, j\, D\, X\,)\, \}
\end{array}
$$

Finally we define a function *divFree* that takes an abstract data type and a set of names of operations on the data type and returns the Boolean true precisely when hiding this set of operations will not cause the system to diverge.

$$
\begin{array}{l}
[State, Name, Value] \\
divFree : CDT[State, Name, Value] \rightarrow \mathbb{P}\, Name \rightarrow Bool \\
\hline
\forall\, D : CDT[State, Name, Value];\ X : \mathbb{P}\, Name \bullet \\
divFree\, D\, X \ = \\
\quad \forall\, a : reachable\, D \ \bullet \ (\, \exists\, k : \mathbb{N} \bullet a \in kNonDiv\, k\, D\, X\,)
\end{array}
$$

Closure

Intuitively we feel that if we hide an operation (or set of operations) on an abstract data type the resulting relations describing the remaining operations will be some composition of the original relations describing those operations and the reflexive transitive closure of the relations describing the operations to be hidden.

In this section we define the diamond operator which takes a communicating data type and a set of named operations on the data type and returns a relation on the state space. A maplet (a, a') lies in this relation either if a' is equal to a or if state a' can be reached from state a after a finite sequence of operations from the given set of names.

First we introduce the box operator which takes a relation from pairs of states and values to pairs of states and values and returns a relation from states to states. Then for all states a and a', the maplet $a \mapsto a'$ lies in the resulting relation if there are some input and output values *in* and *out* such that the maplet $(a, in) \mapsto (a', out)$ lies in the given relation.

$$
\begin{array}{|l}
[X, Y] \\
\hline
\_^{\Box} : ((X \times Y) \leftrightarrow (X \times Y)) \nrightarrow (X \leftrightarrow X) \\
\hline
\forall R : (X \times Y) \leftrightarrow (X \times Y) \bullet \\
\qquad R^{\Box} \ = \ \{x_1, x_2 : X;\ y_1, y_2 : Y \mid \\
\qquad\qquad\qquad (x_1, y_1) \mapsto (x_2, y_2) \in R \bullet (x_1, x_2)\} \\
\hline
\end{array}
$$

We are now able to formally define our diamond operator:

$$
\begin{array}{|l}
[State, Name, Value] \\
\hline
\_^{\diamond} : (CDT[State, Name, Value] \times \mathbb{P}\, Name) \nrightarrow (State \leftrightarrow State) \\
\hline
\forall D : CDT[State, Name, Value];\ X : \mathbb{P}\, Name \mid \\
\qquad divFree\, D\, X \ \wedge \ X \subset \operatorname{dom} ops \bullet \\
\qquad\qquad (D, X)^{\diamond} \ = \ \operatorname{id}[\, state\,] \\
\qquad\qquad\qquad\qquad \cup \\
\qquad\qquad\qquad\qquad (\bigcup\{\, n : X \bullet (ops\, n)^{\Box}\,\})^{*} \\
\hline
\end{array}
$$

where * is the reflexive transitive closure operator. Given any communicating data type D and set of operations X to be hidden, such that hiding X on D will not cause the system to diverge, and given any states a and a', the maplet $a \mapsto a'$ will lie in the relation $(D, X)^{\diamond}$ precisely when a' is equal to a or a' is reachable from a by applying some finite sequence of the operations to be hidden.

B.2 Hiding

Having set up the necessary notation we are now able to give our formal definition of hiding on communicating data types. The function *hide* takes a communicating data type and a set of operations to be hidden as its arguments. It then returns the communicating data type obtained by hiding the set of named operations on the given data type.

We define the state space of the resulting data type to be all those non-internal states of the original data type, the set $state \setminus (internal\, D\, X)$. The set of initial states on the resulting data type is then the set of all non-internal states that can be reached from the set of initial states on the given data type after applying a finite, possibly empty, sequence of the operations to be hidden. This is the set $((D, X)^{\diamond}(\!|\ init\ |\!)) \setminus (internal\, D\, X)$.

The set of operations on the resulting data type contains all those operations on the original data type apart from those that are to be hidden, in other words the set $(\operatorname{dom} ops) \setminus X$.

Given operation n that will remain visible, stable state a and non-internal state a' and values in and out, the maplet $(a, in) \mapsto (a', out)$ will lie in the relation describing n on the resulting data type if there is a state a'' from which a' is reachable after applying a sequence of the operations to be hidden and such that $(a, in) \mapsto (a'', out)$ lies in the relation describing the operation on the given data type. This set of maplets can be described formally as follows:

$$
\begin{array}{l}
\{a, a' : state \setminus internal\, D\, X;\ in, out : Value \mid \\
\qquad \exists\, a'' : state \bullet (a, in) \mapsto (a'', out) \wedge (a'', a') \in (D, X)^{\diamond} \ \bullet \\
\qquad\qquad (a, in) \mapsto (a', out)\}.
\end{array}
$$

There is an additional set of maplets that we must include in the relation describing each operation on the resulting data type: maplets from transient states. Consider the process $P \mathrel{\widehat{=}} a \rightarrow Stop \ \Box \ t \rightarrow b \rightarrow Stop$ illustrated in Figure 4 below. The process $P \setminus t$, obtained by

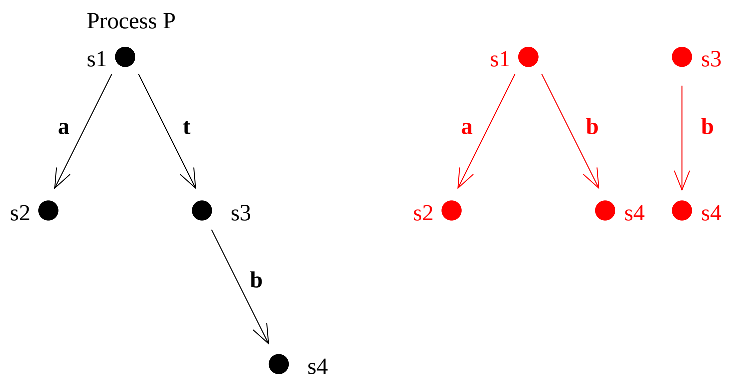

Figure 4: The effect on transient states of hiding

hiding t, is equivalent to the process $(a \rightarrow Stop \;\Box\; b \rightarrow Stop) \sqcap b \rightarrow Stop$. We achieve the same effect on our data types if firstly whenever we might end up in state s_1, we could non-deterministically end up in state s_3 and secondly event b is available from state s_1 as well as from state s_3. In the general case the first condition is achieved by the post-composition of operations with the diamond operator as described above. We achieve the second condition by pre-composing operation b with the diamond operator domain restricted to transient states. More formally we include the following set of maplets:

$$
\begin{array}{l}
\{a : transient\ D\ X;\ a' : state \setminus internal\ D\ X;\ in, out : Value \mid \\
\quad \exists\, b, b' : state \bullet (a, b) \in (D, X)^{\diamond} \\
\qquad\qquad\qquad\quad \wedge \\
\qquad\qquad\qquad\quad (b, in) \mapsto (b', out) \in ops\ n \\
\qquad\qquad\qquad\quad \wedge \\
\qquad\qquad\qquad\quad (b', a') \in (D, X)^{\diamond} \;\bullet \\
\qquad\quad (a, in) \mapsto (a', out)\}.
\end{array}
$$

The union of the two sets of maplets given above is then the relation describing the operation n on the resulting data type. This union may be expressed simply as follows:

$$
\begin{array}{l}
\{a, a' : state \setminus internal;\ in, out : Value \mid \\
\quad \exists\, b, b' : state \bullet \{(a, b), (b', a')\} \subseteq (D, X)^{\diamond} \\
\qquad\qquad\qquad\quad \wedge \\
\qquad\qquad\qquad\quad (b, in) \mapsto (b', out) \in ops\ n \;\bullet \\
\qquad\quad (a, in) \mapsto (a', out)\}.
\end{array}
$$

The function *hide* may then be defined formally as follows:

$$
\begin{array}{l}
[State, Name, Value] \\
hide : CDT[State, Name, Value] \pfun \mathbb{P}\, Name \pfun \\
\qquad CDT[State, Name, Value] \\
\hline
\forall D : CDT[State, Name, Value];\ X : \mathbb{P}\, Name \mid (divFree\ D\ X) \bullet \\
\quad hide\ D\ X\ = \\
\qquad (\mu\ CDT'[State, Name, Value] \mid \\
\qquad\quad state' = state \setminus (internal\ D\ X)\ \wedge \\
\qquad\quad init' = (D, X)^{\diamond} (\!|\ init\ |\!) \setminus (internal\ D\ X)\ \wedge \\
\qquad\quad \mathrm{dom}\, ops' = \mathrm{dom}\, ops \setminus X\ \wedge \\
\qquad\quad \forall n : \mathrm{dom}\, ops';\ a, a' : state';\ i, o : Value\ \bullet \\
\qquad\qquad (a, i) \mapsto (a', o) \in ops'\ n \\
\qquad\qquad\quad \Leftrightarrow \\
\qquad\qquad\qquad \exists\, b, b' : state\ \bullet \\
\qquad\qquad\qquad\quad \{(a, b), (b', a')\} \subseteq (D, X)^{\diamond} \\
\qquad\qquad\qquad\quad \wedge \\
\qquad\qquad\qquad\quad (b, in) \mapsto (b', out) \in ops\ n\ ).
\end{array}
$$

B.3 Healthiness conditions for hiding

We saw in Observation 1 of Section 3.2.3 that given processes P and Q and set A as defined below:

$$
\begin{array}{lll}
P\ =\ a \to e \to Stop & \qquad Q\ =\ e \to Stop, & \qquad A\ =\ \{\, a \,\}, \\
\qquad \sqcap & & \\
\qquad (\ a \to e \to Stop & & \\
\qquad\quad \Box & & \\
\qquad\quad e \to Stop\ ), & &
\end{array}
$$

the set $single[\,(P\ \setminus\ A)\ \sqcap\ (Q\ \setminus\ A)\,]$ is not equal to the set $single[\,(P\ \sqcap\ Q)\ \setminus\ A\,]$. This observation illustrated the fact that the singleton failures semantics is not a congruence. In this section we explore the conditions under which congruence fails and derive a set of healthiness conditions for hiding on our data types: if, for any given data type and set of operations to be hidden, the healthiness conditions are satisfied then hiding the corresponding set of events on the corresponding process will not break the congruence.

The definition of hiding within the singleton failures semantics that we presented in Chapter 3.2.3 was

$$
\begin{array}{lll}
single[\,P \setminus X\,] & = & \{\ tr : \mathrm{seq}\, Event;\ ref : \mathbb{P}\, Event \mid \\
& & \quad \#ref \leq 1 \\
& & \quad \wedge \\
& & \quad \forall\, Y : \mathbb{P}(X \cup ref) \mid \#Y \leq 1\ \bullet\ (tr, Y) \in single[P]\ \bullet \\
& & \qquad (tr \setminus X, ref)\ \}.
\end{array}
$$

By considering separately the cases when ref is the empty set, when it contains a single event from the set X and when it contains a single event not from the set X, we see that the set

$single[\, P \setminus X\,]$ may equivalently be expressed as the following disjoint set union:

$$
\begin{array}{rcl}
single[\, P \setminus X\,] & = & \{\, tr : \operatorname{seq} Event \mid \\
& & \qquad \forall\, x : X \bullet (tr, \{x\}) \in single[P] \;\bullet\; (tr \setminus X, \langle\rangle)\,\} \\
& & \;\cup \\
& & \{\, tr : \operatorname{seq} Event;\; e : X \mid \\
& & \qquad \forall\, x : X \bullet (tr, \{x\}) \in single[P] \;\bullet\; (tr \setminus X, \{e\})\,\} \\
& & \;\cup \\
& & \{\, tr : \operatorname{seq} Event;\; e : Event \setminus X \mid \\
& & \qquad (tr, \{e\}) \in single[P] \\
& & \qquad \wedge \\
& & \qquad \forall\, x : X \bullet (tr, \{x\}) \in single[P] \;\bullet\; (tr \setminus X, \{e\})\,\}.
\end{array}
$$

By expanding the definitions of $single[\,(P \setminus X) \sqcap (Q \setminus X)\,]$ and $single[\,(P \sqcap Q) \setminus X\,]$ as follows:

$$
\begin{array}{l}
single[\,(P \setminus X) \sqcap (Q \setminus X)\,] \\
\quad = \\
\qquad \{\, tr : \operatorname{seq} Event \mid (\,\forall\, x : X \bullet (tr, \{x\}) \in single[P] \\
\qquad\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad\qquad \forall\, x : X \bullet (tr, \{x\}) \in single[Q]\,) \;\bullet\; (tr \setminus X, \langle\rangle)\,\} \\
\qquad \cup \\
\qquad \{\, tr : \operatorname{seq} Event;\; e : X \mid (\,\forall\, x : X \bullet (tr, \{x\}) \in single[P] \\
\qquad\qquad\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad\qquad\qquad \forall\, x : X \bullet (tr, \{x\}) \in single[Q]\,) \;\bullet\; (tr \setminus X, \{e\})\,\} \\
\qquad \cup \\
\qquad \{\, tr : \operatorname{seq} Event;\; e : Event \setminus X \mid \\
\qquad\qquad (\,(tr, \{e\}) \in single[P] \wedge \forall\, x : X \bullet (tr, \{x\}) \in single[P] \\
\qquad\qquad \vee \\
\qquad\qquad (tr, \{e\}) \in single[Q] \wedge \forall\, x : X \bullet (tr, \{x\}) \in single[Q]\,) \;\bullet\; (tr \setminus X, \{e\})\,\}
\end{array}
$$

$$
\begin{array}{l}
single[\,(P \sqcap Q) \setminus X\,] \\
\quad = \\
\qquad \{\, tr : \operatorname{seq} Event \mid \\
\qquad\qquad \forall\, x : X \bullet (tr, \{x\}) \in single[P] \cup single[Q] \bullet\; (tr \setminus X, \langle\rangle)\,\} \\
\qquad \cup \\
\qquad \{\, tr : \operatorname{seq} Event;\; e : X \mid \\
\qquad\qquad \forall\, x : X \bullet (tr, \{x\}) \in single[P] \cup single[Q] \;\bullet\; (tr \setminus X, \{e\})\,\} \\
\qquad \cup \\
\qquad \{\, tr : \operatorname{seq} Event;\; e : Event \setminus X \mid \\
\qquad\qquad (\,(tr, \{e\}) \in single[P] \cup single[Q] \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, x : X \bullet (tr, \{x\}) \in single[P] \cup single[Q]\,) \;\bullet\; (tr \setminus X, \{e\})\,\}
\end{array}
$$

we see that, for *any* processes P and Q and any set of events X, the sets $single[\,(P \setminus X) \sqcap (Q \setminus X)\,]$ and $single[\,(P \sqcap Q) \setminus X\,]$ will not be equivalent either if hiding X on P breaks the congruence or if hiding X on Q breaks the congruence or if any of the following three predicates is true:

•

$$
\begin{array}{l}
\{\, tr : \operatorname{seq} \mathit{Event} \mid (\forall\, x : X \bullet (tr, \{x\}) \in \mathit{single}[P] \\
\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad \forall\, x : X \bullet (tr, \{x\}) \in \mathit{single}[Q]\,) \;\bullet\; (tr \setminus X, \langle\rangle)\,\} \\
\quad \neq \\
\qquad \{\, tr : \operatorname{seq} \mathit{Event} \mid \\
\qquad\quad \forall\, x : X \bullet (tr, \{x\}) \in \mathit{single}[P] \cup \mathit{single}[Q] \;\bullet\; (tr \setminus X, \langle\rangle)\,\}
\end{array}
$$

- $$
\begin{array}{l}
\{\, tr : \operatorname{seq} \mathit{Event};\; e : X \mid (\forall\, x : X \bullet (tr, \{x\}) \in \mathit{single}[P] \\
\qquad\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad\qquad \forall\, x : X \bullet (tr, \{x\}) \in \mathit{single}[Q]\,) \;\bullet\; (tr \setminus X, \{e\})\,\} \\
\quad \neq \\
\qquad \{\, tr : \operatorname{seq} \mathit{Event};\; e : X \mid \\
\qquad\quad \forall\, x : X \bullet (tr, \{x\}) \in \mathit{single}[P] \cup \mathit{single}[Q] \;\bullet\; (tr \setminus X, \{e\})\,\}
\end{array}
$$

- $$
\begin{array}{l}
\{\, tr : \operatorname{seq} \mathit{Event};\; e : \mathit{Event} \setminus X \mid \\
\qquad (\,(tr, \{e\}) \in \mathit{single}[P] \;\wedge\; \forall\, x : X \bullet (tr, \{x\}) \in \mathit{single}[P] \\
\qquad \vee \\
\qquad (tr, \{e\}) \in \mathit{single}[Q] \;\wedge\; \forall\, x : X \bullet (tr, \{x\}) \in \mathit{single}[Q]\,) \\
\qquad\qquad \bullet\; (tr \setminus X, \{e\})\,\} \\
\quad \neq \\
\qquad \{\, tr : \operatorname{seq} \mathit{Event};\; e : \mathit{Event} \setminus X \mid \\
\qquad\quad (\,(tr, \{e\}) \in \mathit{single}[P] \cup \mathit{single}[Q] \\
\qquad\quad \wedge \\
\qquad\quad \forall\, x : X \bullet (tr, \{x\}) \in \mathit{single}[P] \cup \mathit{single}[Q]\,) \;\bullet\; (tr \setminus X, \{e\})\,\}.
\end{array}
$$

By applying a little set theory, we see that the first two predicates are equivalent and hence that the three predicates can be reduced to the two predicates below; we deduce that, for *any* processes P and Q and any set of events X, the sets $\mathit{single}[\,(P \setminus X) \sqcap (Q \setminus X)\,]$ and $\mathit{single}[\,(P \sqcap Q) \setminus X\,]$ will not be equivalent either if hiding X on P breaks the congruence or if hiding X on Q breaks the congruence or if either of the following predicates is true:

- $$
\begin{array}{l}
\exists\, s : \operatorname{seq}(\mathit{Event} \setminus X) \;\bullet \\
\quad \exists\, s' : \operatorname{seq} \mathit{Event} \;\mid\; s' \upharpoonright (\mathit{Event} \setminus X) = s \;\bullet \\
\qquad \forall\, x : X \bullet (s', \{x\}) \in \mathit{single}[P] \cup \mathit{single}[Q] \\
\quad \wedge \\
\quad \forall\, s'' : \operatorname{seq} \mathit{Event} \;\mid\; s'' \upharpoonright (\mathit{Event} \setminus X) = s \;\bullet \\
\qquad \exists\, x_1, x_2 : X \;\bullet\; (s'', \{x_1\}) \notin \mathit{single}[P] \wedge (s'', \{x_2\}) \notin \mathit{single}[Q]
\end{array}
$$

- $$
\begin{array}{l}
\exists\, s : \operatorname{seq}(\mathit{Event} \setminus X);\; v : \mathit{Event} \setminus X \;\bullet \\
\quad \exists\, s' : \operatorname{seq} \mathit{Event} \;\mid\; s' \upharpoonright (\mathit{Event} \setminus X) = s \;\bullet \\
\qquad \forall\, e : X \cup \{v\} \bullet (s', \{e\}) \in \mathit{single}[P] \cup \mathit{single}[Q] \\
\quad \wedge \\
\quad \forall\, s'' : \operatorname{seq} \mathit{Event} \;\mid\; s'' \upharpoonright (\mathit{Event} \setminus A) = s \;\bullet \\
\qquad \exists\, e_1, e_2 : X \cup \{v\} \bullet (s'', \{e_1\}) \notin \mathit{single}[P] \wedge (s'', \{e_2\}) \notin \mathit{single}[Q].
\end{array}
$$

Furthermore, by applying de Morgan's law and then considering the negations of the above predicates we can obtain conditions that ensure that the congruence is *not* broken: provided that hiding X on P does not break the congruence and provided that hiding X on Q does not break the congruence, the sets $\mathit{single}[\,(P \setminus X) \sqcap (Q \setminus X)\,]$ and $\mathit{single}[\,(P \sqcap Q) \setminus X\,]$ will be equivalent when both the predicates below are true.

- $$\begin{array}{l} \forall\, s : \mathrm{seq}(\mathit{Event} \setminus X) \bullet \\ \quad (\,\exists\, s' : \mathrm{seq}\,\mathit{Event} \mid s' \restriction (\mathit{Event} \setminus X) = s \bullet \\ \qquad\qquad \forall\, x : X \bullet (s', \{x\}) \in \mathit{single}[P] \cup \mathit{single}[Q]\,) \\ \qquad \Rightarrow \\ \qquad\quad (\,\exists\, s'' : \mathrm{seq}\,\mathit{Event} \mid s'' \restriction (\mathit{Event} \setminus X) = s \bullet \\ \qquad\qquad\quad \forall\, x : X \bullet (s'', \{x\}) \in \mathit{single}[P] \\ \qquad\qquad \vee \\ \qquad\qquad \exists\, s'' : \mathrm{seq}\,\mathit{Event} \mid s'' \restriction (\mathit{Event} \setminus X) = s \bullet \\ \qquad\qquad\quad \forall\, x : X \bullet (s'', \{x\}) \in \mathit{single}[Q]\,) \end{array}$$

- $$\begin{array}{l} \forall\, s : \mathrm{seq}(\mathit{Event} \setminus X);\ v : \mathit{Event} \setminus X \bullet \\ \quad (\,\exists\, s' : \mathrm{seq}\,\mathit{Event} \mid s' \restriction (\mathit{Event} \setminus X) = s \bullet \\ \qquad\qquad \forall\, e : X \cup \{v\} \bullet (s', \{e\}) \in \mathit{single}[P] \cup \mathit{single}[Q]\,) \\ \qquad \Rightarrow \\ \qquad\quad (\,\exists\, s'' : \mathrm{seq}\,\mathit{Event} \mid s'' \restriction (\mathit{Event} \setminus X) = s \bullet \\ \qquad\qquad\quad \forall\, e : X \cup \{v\} \bullet (s'', \{e\}) \in \mathit{single}[P] \\ \qquad\qquad \vee \\ \qquad\qquad \exists\, s'' : \mathrm{seq}\,\mathit{Event} \mid s'' \restriction (\mathit{Event} \setminus X) = s \bullet \\ \qquad\qquad\quad \forall\, e : X \cup \{v\} \bullet (s'', \{e\}) \in \mathit{single}[Q]\,). \end{array}$$

The first condition states that if each event to be hidden can be refused after a given trace by process P or process Q then there must be a bisimilar[1] trace after which either every event to be hidden can be refused by P, or after which every event to be hidden can be refused by Q.

The second condition is similar. It states that if a visible event v and each event to be hidden can be refused after a given trace by process P or process Q then there must be a bisimilar trace after which either v and every event to be hidden can be refused by P or after which v and every event to be hidden can be refused by Q.

We may obtain a similar result for hiding a set of events over the internal choice of an *indexed* set of processes. Given processes $P(i)$ indexed over a set I and a set of events X—provided that, for each i in I, hiding the set of events X on $P(i)$ does not break the congruence—the sets $\mathit{single}[\sqcap\, i : I \bullet (P(i) \setminus X)\,]$ and $\mathit{single}[\,(\sqcap\, i : I \bullet P(i)) \setminus X\,]$ will be equivalent if the following predicates hold:

- $$\begin{array}{l} \forall\, s : \mathrm{seq}(\mathit{Event} \setminus X) \bullet \\ \quad (\,\exists\, s' : \mathrm{seq}\,\mathit{Event} \mid s' \restriction (\mathit{Event} \setminus X) = s \bullet \\ \qquad\qquad \forall\, x : X \bullet (s', \{x\}) \in \bigcup_{i:I}\ \mathit{single}[P(i)]\,) \\ \qquad \Rightarrow \\ \qquad\quad (\,\exists\, i : I;\ s'' : \mathrm{seq}\,\mathit{Event} \mid s'' \restriction (\mathit{Event} \setminus X) = s \bullet \\ \qquad\qquad\quad \forall\, x : X \bullet (s'', \{x\}) \in \mathit{single}[P(i)]\,) \end{array}$$

- $$\begin{array}{l} \forall\, s : \mathrm{seq}(\mathit{Event} \setminus X);\ v : \mathit{Event} \setminus X \bullet \\ \quad (\,\exists\, s' : \mathrm{seq}\,\mathit{Event} \mid s' \restriction (\mathit{Event} \setminus X) = s \bullet \\ \qquad\qquad \forall\, e : X \cup \{v\} \bullet (s', \{e\}) \in \bigcup_{i:I}\ \mathit{single}[P(i)]\,) \\ \qquad \Rightarrow \\ \qquad\quad (\,\exists\, i : I;\ s'' : \mathrm{seq}\,\mathit{Event} \mid s'' \restriction (\mathit{Event} \setminus X) = s \bullet \\ \qquad\qquad\quad \forall\, e : X \cup \{v\} \bullet (s'', \{e\}) \in \mathit{single}[P(i)]\,). \end{array}$$

[1] We say that two processes are bisimilar if their observable behaviour is the same.

The first constraint states that if each event to be hidden can be refused after a given trace by at least one of the processes then there must be a particular process and a bisimilar trace such that each event to be hidden can be refused by that process after that trace. The second constraint states that if a remaining visible event v and each event to be hidden can be refused after a given trace by at least one of the processes then there must be a particular process and a bisimilar trace such that v and each event to be hidden can be refused by that process after that trace.

We now consider what these predicates mean in terms of the processes corresponding to our data types. The first constraint will trivially be true if we insist that hiding operations does not permit the possibility of introducing divergence: if, for communicating data type D and set of named operations X, the predicate $\mathit{divFree}\ D\ X$ is true then from any reachable state it will always be possible to reach a stable state—from which all hidden operations will be refused—by performing a finite number of operations to be hidden.

The second constraint is a little more subtle. We *could* construct a necessary and sufficient constraint that our communicating data types must satisfy in order to ensure that the second predicate is true. In practice, however, this would be unworkable; every possible sequence of interaction would have to be considered. Instead we present a simple necessary condition that would be easy to verify. Informally we require that if a visible event can be refused from an unstable state then there must be a stable state, reachable from that unstable state after a finite number of hidden events, from which the visible event can also be refused. In fact, because we do not permit systems that diverge we can equivalently relax this to the following even simpler constraint: whenever a visible operation can be refused from a transient state there must be another, distinct, non-internal state reachable from the first state via a sequence of hidden operations, such that the visible operation may also be refused from the second state.

Momentarily—for clarity whilst translating from a process-based perspective to a state-based perspective—mixing our notations we require that the following constraint is true:

$$\begin{array}{l}
\forall\, a : \mathit{transient}\ D\ X;\ \mathit{name} : (\mathrm{dom}\ \mathit{ops}) \setminus X;\ \mathit{in}, \mathit{out} : \mathit{Value}\ \bullet \\
\quad \mathrm{op}.\mathit{name}.\mathit{in}.\mathit{out} \in \mathrm{refusals}\,(\mathit{Proc}(a)) \\
\qquad \Rightarrow \\
\qquad\quad \exists\, b : \mathit{state} \setminus (\mathit{internal}\ D\ X \cup \{a\}) \mid b \in (D, X)^{\diamond}(\!|\ \{a\}\ |\!)\ \bullet \\
\qquad\qquad \mathrm{op}.\mathit{name}.\mathit{in}.\mathit{out} \in \mathrm{refusals}\,(\mathit{Proc}(b))
\end{array}$$

Recalling that an event op.$\mathit{name}.\mathit{in}.\mathit{out}$ can be refused from a state a for three reasons: either the pair (a, in) does not lie in the domain of the relation describing operation *name*; or the pair (a, in) does lie in the domain of the relation describing operation *name* but the value *out* is not one of the possible outputs from state a; or the pair (a, in) does lie in the domain of the relation describing operation *name* and *out* is one of the possible outputs from state a but it is not the only possible output value from state a. The first two reasons can be captured by the predicate

$$\neg\ \exists\, a' : \mathit{state} \bullet (a, \mathit{in}) \mapsto (a', \mathit{out}) \in \mathit{ops}\ \mathit{name}.$$

The third reason can be captured by the predicate

$$\exists\, a' : \mathit{state};\ \mathit{out}' : \mathit{Value} \setminus \{\mathit{out}\} \bullet (a, \mathit{in}) \mapsto (a', \mathit{out}') \in \mathit{ops}\ \mathit{name}.$$

So, given abstract data type D and set of operations to be hidden X, and recalling that we insist that the possibility of divergence is not introduced, our healthiness conditions for hiding on data types are as follows:

$$\mathit{divFree}\ D\ X \qquad (1)$$

$$\begin{array}{l}
\forall a : transient\ D\ X;\ name : \mathrm{dom}\ ops \setminus X;\ in, out : Value \bullet \\
\quad (\neg\ \exists a' : state \bullet (a, in) \mapsto (a', out) \in ops\ name \\
\quad \lor \\
\quad \exists a' : state;\ out' : Value \setminus \{out\} \bullet (a, in) \mapsto (a', out') \in ops\ name\ ) \\
\qquad \Rightarrow \\
\qquad\quad (\ \exists b : state \setminus ((internal\ D\ X) \cup \{a\}) \mid b \in (D, X)^{\diamond} (\!|\ \{a\}\ |\!) \bullet \\
\qquad\qquad \neg\ \exists b' : state \bullet (b, in) \mapsto (b', out) \in ops\ name \\
\qquad\qquad \lor \\
\qquad\qquad \exists b' : state;\ out' : Value \setminus \{out\} \bullet \\
\qquad\qquad\quad (b, in) \mapsto (b', out') \in ops\ name\ ).
\end{array} \tag{2}$$

B.4 Consistency of hiding

In this section we prove that, provided the healthiness conditions are satisfied, hiding a set of operations on a data type is equivalent to hiding the corresponding set of events on the corresponding data types.

First we introduce the predicate *hideable* which, for any communicating data type and any set of operations to be hidden, is true precisely when the healthiness conditions are satisfied.

$$\begin{array}{l}
[State, Name, Value] \\
hideable : CDT[State, Name, Value] \nrightarrow \mathbb{P}\ Name \nrightarrow Bool \\
\hline
\forall D : CDT[State, Name, Value];\ X : \mathbb{P}\ Name \bullet \\
\quad hideable\ D\ X\ = \\
\qquad divFree\ D\ X \\
\qquad \land \\
\qquad \forall a : transient\ D\ X;\ name : \mathrm{dom}\ ops \setminus X;\ in, out : Value \bullet \\
\qquad\quad (\neg\ \exists a' : state \bullet (a, in) \mapsto (a', out) \in ops\ name \\
\qquad\quad \lor \\
\qquad\quad \exists a' : state;\ out' : Value \setminus \{out\} \bullet \\
\qquad\qquad (a, in) \mapsto (a', out') \in ops\ name\ ) \\
\qquad\qquad \Rightarrow \\
\qquad\qquad (\ \exists b : state \setminus ((internal\ D\ X) \cup \{a\}) \mid \\
\qquad\qquad\qquad b \in (D, X)^{\diamond} (\!|\ \{a\}\ |\!) \bullet \\
\qquad\qquad\quad \neg\ \exists b' : state \bullet \\
\qquad\qquad\qquad (b, in) \mapsto (b', out) \in ops\ name \\
\qquad\qquad\quad \lor \\
\qquad\qquad\quad \exists b' : state;\ out' : Value \setminus \{out\} \bullet \\
\qquad\qquad\qquad (b, in) \mapsto (b', out') \in ops\ name\ )
\end{array}$$

We introduce *Event*, the set of all Events,

$$Event == \{name : Name;\ in, out : Value\ \bullet\ \mathrm{op}.name.in.out\}$$

and finally define the functions *eventH* and *eventV* that take the names of the operations to be hidden and respectively return the corresponding set of events to be hidden and those events remaining visible.

$$
\begin{array}{l}
[State, Name, Value] \\
eventH, eventV : \mathbb{P}\, Name \rightarrow\!\!\!\!\!+ \mathbb{P}\, Event \\
\hline
\forall X : \mathbb{P}\, Name \bullet \\
\quad eventH\ X \;=\; \{\, n : Name;\ i, o : Value \mid n \in X \bullet \mathrm{op}.n.i.o \,\} \\
\quad \wedge \\
\quad eventV\ X \;=\; \{\, n : Name;\ i, o : Value \mid n \notin X \bullet \mathrm{op}.n.i.o \,\}
\end{array}
$$

The theorem that our two definitions of hiding are consistent can then be stated formally as follows:

Theorem 4

$$
\begin{array}{l}
\forall D : CDT[State, Name, Value];\ X : \mathbb{P}\, Name \mid hideable\ D\ X \bullet \\
\quad single[\, process_{io}\, (hide\ D\ X)\,] \\
\qquad = \\
\qquad\quad single[\, (\, process_{io}\ D\,) \setminus (\, eventH\ X\,)\,]
\end{array}
$$

Proof

Let D be a communicating data type and X be a set of names of operations such that the predicate $hideable\, D\, X$ is true. We will consider separately the cases when the intersection between the set of operations to be hidden and the set of operations on the data type is the empty set and when it is non-empty.

Case 1 *The intersection between the set of operations to be hidden and the set of operations on the data type is the empty set.*

In this case the result is trivially true: both

$$single[\, process_{io}\, (hide\ D\ X)\,] \qquad \text{and} \qquad single[\, (\, process_{io}\ D\,) \setminus (\, eventH\ X\,)\,]$$

are equal to $single[\, process_{io}\, D\,]$. Since the intersection between the set of operations to be hidden and the set of operations on D is the empty set, the process

$$(\, process_{io}\ D\,) \setminus (\, eventH\ X\,)$$

is semantically equivalent to the process $process_{io}\, D \setminus \emptyset$ and hence to $process_{io}\, D$. Similarly data type $hide\, D\, X$ is equivalent to data type $hide\, D\, \emptyset$ to hence to D.

Case 2 *The intersection between the set of operations to be hidden and the set of operations on the data type is non-empty.*

Recall the definition of hiding within the singleton failures semantic model. The set of singleton failures can be expressed as the disjoint union of three sets: the first contains trace-refusal pairs in which the refusal set is empty; the second contains trace-refusal pairs in which the refusal set contains a single event from the set of events to be hidden; and the third set contains trace-refusal *pairs in which the refusal set contains a single event from the set of remaining visible events.*

$$\begin{aligned} single[\,P \setminus A\,] \;=\; & \{\, s : \operatorname{seq} Event \mid \\ & \quad \forall\, a : A \bullet (s, \{a\}) \in single[P] \;\bullet\; (\, s \setminus A, \emptyset\,)\,\} \\ & \cup \\ & \{\, s : \operatorname{seq} Event;\; e : A \mid \\ & \quad \forall\, a : A \bullet (s, \{a\}) \in single[P] \;\bullet\; (\, s \setminus A, \{e\}\,)\,\} \\ & \cup \\ & \{\, s : \operatorname{seq} Event;\; e : Event \setminus X \mid \\ & \quad (s, \{e\}) \in single[P] \\ & \quad \wedge \\ & \quad \forall\, a : A \bullet (s, \{a\}) \in single[P] \;\bullet\; (s \setminus A, \{e\}\,)\,\} \end{aligned}$$

Towards the end of our proof we will consider these three sub-cases separately but, noting that $single[P \setminus A]$ is expressed in terms of $single[P]$, we see that first we must expand our definitions of $process_{io}\,(hide\,D\,X)$ and $process_{io}\,D$ and express $single[\,process_{io}\,(hide\,D\,X)\,]$ and $single[\,(\,process_{io}\,D\,)]$ as set comprehensions.

Step 1 *Expanding $single[\,process_{io}\,(hide\,D\,X)\,]$ and $single[\,(\,process_{io}\,D\,)]$.*

First we consider $single[\,process_{io}\,(hide\,D\,X)\,]$. We reason as follows:

$$single[\,process_{io}\,(hide\,D\,X\,)\,]$$

$$\begin{aligned} \equiv \quad & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [def^n \text{ of } hide] \\ & single[\,process_{io}\,(\,(\,\mu\, CDT'[State, Name, Value] \mid \\ & \qquad state' = state \setminus (internal\,D\,X) \;\wedge \\ & \qquad init' = (D, X)^{\diamond}\,(\!|\; init \;|\!)\setminus (internal\,D\,X) \;\wedge \\ & \qquad \operatorname{dom} ops' = \operatorname{dom} ops \setminus X \;\wedge \\ & \qquad \forall\, n : \operatorname{dom} ops';\; a, a' : state';\; i, o : Value \;\bullet \\ & \qquad\quad (a, i) \mapsto (a', o) \in ops'\, n \\ & \qquad\qquad \Leftrightarrow \\ & \qquad\qquad\quad \exists\, b, b' : state \bullet \\ & \qquad\qquad\qquad \{(a, b), (b', a')\} \in (D, X)^{\diamond} \\ & \qquad\qquad\qquad \wedge \\ & \qquad\qquad\qquad (b, i) \mapsto (b', o) \in ops\, n\,)\,)\,] \end{aligned}$$

$$
\begin{array}{ll}
\equiv & \hfill \textit{[def}^n\textit{ of process}_{io}\textit{]} \\
& single[\,let \\
& \qquad Proc(a) \;= \\
& \qquad\quad \Box\, n : \mathrm{dom}\, ops';\ i : Value \mid (a,i) \in \mathrm{dom}\, ops'\, n \bullet \\
& \qquad\qquad \sqcap\, a' : State;\ o : Value \mid \\
& \qquad\qquad\qquad (a,i) \mapsto (a',o) \in ops'\, n \bullet \\
& \qquad\qquad\quad \mathrm{op}.n.i.o \to Proc(a') \\
& \qquad state' = state \setminus (internal\, D\, X) \\
& \qquad init' = (D,X)^{\diamond} (\!|\, init \,|\!) \setminus (internal\, D\, X) \\
& \qquad \mathrm{dom}\, ops' = \mathrm{dom}\, ops \setminus X \\
& \qquad \forall\, n : \mathrm{dom}\, ops';\ a, a' : state';\ i, o : Value \;\bullet \\
& \qquad\quad (a,i) \mapsto (a',o) \in ops'\, n \\
& \qquad\qquad \Leftrightarrow \\
& \qquad\qquad\quad \exists\, b, b' : state \bullet \{(a,b),(b',a')\} \subseteq (D,X)^{\diamond} \\
& \qquad\qquad\qquad\qquad \wedge \\
& \qquad\qquad\qquad\qquad (b,i) \mapsto (b',o) \in ops\, n \\
& \quad within \\
& \qquad \sqcap\, a' : init' \;\bullet\; Proc(a')\,] \\
\equiv & \hfill \textit{[substitution and re-naming]} \\
& single[\,let \\
& \qquad ProcLHS(a) \;= \\
& \qquad\quad \Box\, n : (\mathrm{dom}\, ops \setminus X);\ i : Value \mid \\
& \qquad\qquad\quad \exists\, b : state \bullet (a,b) \in (D,X)^{\diamond} \\
& \qquad\qquad\qquad\qquad \wedge \\
& \qquad\qquad\qquad\qquad (b,i) \in \mathrm{dom}\, ops\, n \;\bullet \\
& \qquad\qquad \sqcap\, a' : State;\ o : Value \mid \\
& \qquad\qquad\quad \exists\, b, b' : state \bullet \{(a,b),(b',a')\} \subseteq (D,X)^{\diamond} \\
& \qquad\qquad\qquad\qquad \wedge \\
& \qquad\qquad\qquad\qquad (b,i) \mapsto (b',o) \in ops\, n \\
& \qquad\qquad\qquad \bullet\; \mathrm{op}.n.i.o \to ProcLHS(a') \\
& \quad within \\
& \qquad \sqcap\, a' : (D,X)^{\diamond} (\!|\, init \,|\!) \setminus (internal\, D\, X) \;\bullet\; ProcLHS(a')\,]
\end{array}
$$

Recalling the definition of $process_{io}\, D$ we can express $single[\,(\, process_{io}\, D\,)]$ in a similar form to our expanded version of $single[\, process_{io}\,(hide\, D\, X)\,]$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{single[\,(\, process_{io}\, D\,)]} \\
= & \hfill \textit{[def}^n\textit{ of process}_{io}\textit{]} \\
& single[\,let \\
& \qquad Proc(a) \;= \\
& \qquad\quad \Box\, n : \mathrm{dom}\, ops;\ i : Value \mid (a,i) \in \mathrm{dom}\, ops\, n \bullet \\
& \qquad\qquad \sqcap\, a' : State;\ o : Value \mid (a,i) \mapsto (a',o) \in ops\, n \bullet \\
& \qquad\qquad\quad \mathrm{op}.n.i.o \to Proc(a') \\
& \quad within \\
& \qquad \sqcap\, a' : init \;\bullet\; Proc(a')\,)\,]
\end{array}
$$

Step 2 *Expanding* $single[ProcLHS(a)]$ *and* $single[Proc(a)]$.

Our expanded definitions of $single[\,process_{io}\,(hide\,D\,X)\,]$ *and* $single[\,(\,process_{io}\,D\,)]$ *are very similar in structure. They are defined recursively in terms of the processes ProcLHS and Proc respectively and we need to consider these processes further. Given any stable state* $a : stable\,D\,X$ *we express* $single[ProcLHS(a)]$ *and* $single[Proc(a)]$ *as the disjoint union of sets as follows:*

$$
\begin{array}{l}
single[\,let \\
\qquad ProcLHS(a)\ = \\
\qquad\quad \Box\, n : \mathrm{dom}\, ops \setminus X;\ i : Value \mid \\
\qquad\qquad\quad \exists\, b : state\ \bullet\ (a,b) \in (D,X)^{\diamond} \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge \\
\qquad\qquad\qquad\qquad\qquad\quad (b,i) \in \mathrm{dom}\, ops\, n\ \bullet \\
\qquad\qquad \sqcap\, a' : State;\ o : Value \mid \\
\qquad\qquad\qquad \exists\, b, b' : state\ \bullet\ \{(a,b),(b',a')\} \subseteq (D,X)^{\diamond} \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad (b,i) \mapsto (b',o) \in ops\, n \\
\qquad\qquad\quad \mathrm{op}.n.i.o \rightarrow ProcLHS(a') \\
\quad within \\
\qquad ProcLHS(a)\,]
\end{array}
$$

$$
\begin{array}{l}
= \\
\quad \bigcap \{\, n : \mathrm{dom}\, ops \setminus X;\ i : Value \mid \\
\qquad\qquad \exists\, b : state\ \bullet\ (a,b) \in (D,X)^{\diamond} \\
\qquad\qquad\qquad\qquad\quad \wedge \\
\qquad\qquad\qquad\qquad\quad (b,i) \in \mathrm{dom}\, ops\, n\ \bullet \\
\qquad\quad \bigcup \{\, a' : State;\ o : Value \mid \\
\qquad\qquad\qquad \exists\, b, b' : state\ \bullet\ \{(a,b),(b',a')\} \subseteq (D,X)^{\diamond} \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge \\
\qquad\qquad\qquad\qquad\qquad\quad (b,i) \mapsto (b',o) \in ops\, n \\
\qquad\qquad \{\, (\langle\rangle, \emptyset)\,\} \\
\qquad\qquad \cup \\
\qquad\qquad \{\, e : Event \mid e \neq \mathrm{op}.n.i.o \bullet (\langle\rangle, \{e\})\,\} \\
\qquad\quad \} \\
\quad \} \\
\quad \cup \\
\quad \bigcup \{\, n : \mathrm{dom}\, ops \setminus X;\ i : Value \mid \\
\qquad\qquad \exists\, b : state\ \bullet\ (a,b) \in (D,X)^{\diamond} \\
\qquad\qquad\qquad\qquad\quad \wedge \\
\qquad\qquad\qquad\qquad\quad (b,i) \in \mathrm{dom}\, ops\, n \bullet \\
\qquad\quad \bigcup \{\, a' : State;\ o : Value \mid \\
\qquad\qquad\qquad \exists\, b, b' : state\ \bullet\ \{(a,b),(b',a')\} \subseteq (D,X)^{\diamond} \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge \\
\qquad\qquad\qquad\qquad\qquad\quad (b,i) \mapsto (b',o) \in ops\, n \\
\qquad\qquad \{ s : \mathrm{seq}\, Event;\ X : \mathbb{P}\, Event \mid \\
\qquad\qquad\quad (s, X) \in single[\, ProcLHS(a')\,] \bullet \\
\qquad\qquad\qquad (\langle \mathrm{op}.n.i.o \rangle \frown s, X)\} \\
\qquad\quad \} \\
\quad \}
\end{array}
$$

Similarly,

$$
\begin{array}{l}
\mathit{single}[\,\mathit{let} \\
\qquad\qquad \mathit{Proc}(a) \;=\; \\
\qquad\qquad\qquad \Box\, n : \mathrm{dom}\, \mathit{ops};\; i : \mathit{Value} \mid (a, i) \in \mathrm{dom}\, \mathit{ops}\, n \bullet \\
\qquad\qquad\qquad\qquad \sqcap\, a' : \mathit{State};\; o : \mathit{Value} \mid (a, i) \mapsto (a', o) \in \mathit{ops}\, n \bullet \\
\qquad\qquad\qquad\qquad\qquad \mathrm{op}.n.i.o \rightarrow \mathit{Proc}(a') \\
\qquad\quad \mathit{within} \\
\qquad\qquad \mathit{Proc}(a)\,] \\
\qquad = \\
\qquad\quad \bigcap\,\{\, n : \mathrm{dom}\, \mathit{ops};\; i : \mathit{Value} \mid (a, i) \in \mathrm{dom}\, \mathit{ops}\, n \bullet \\
\qquad\qquad\qquad \bigcup\,\{\, a' : \mathit{State};\; o : \mathit{Value} \mid (a, i) \mapsto (a', o) \in \mathit{ops}\, n \bullet \\
\qquad\qquad\qquad\qquad \{\, (\langle\rangle, \emptyset)\,\} \\
\qquad\qquad\qquad\qquad \cup \\
\qquad\qquad\qquad\qquad \{\, e : \mathit{Event} \mid e \neq \mathrm{op}.n.i.o \bullet (\langle\rangle, \{e\})\,\} \\
\qquad\qquad\qquad \} \\
\qquad\qquad \} \\
\qquad\quad \cup \\
\qquad\quad \bigcup\,\{\, n : \mathrm{dom}\, \mathit{ops};\; i : \mathit{Value} \mid (a, i) \in \mathrm{dom}\, \mathit{ops}\, n \bullet \\
\qquad\qquad\qquad \bigcup\,\{\, a' : \mathit{State};\; o : \mathit{Value} \mid (a, i) \mapsto (a', o) \in \mathit{ops}\, n \bullet \\
\qquad\qquad\qquad\qquad \{ s : \mathrm{seq}\, \mathit{Event};\; X : \mathbb{P}\, \mathit{Event} \mid \\
\qquad\qquad\qquad\qquad\qquad (s, X) \in \mathit{single}[\, \mathit{Proc}(a')\,] \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad (\langle \mathrm{op}.n.i.o \rangle \frown s, X) \} \\
\qquad\qquad\qquad \} \\
\qquad\qquad \}
\end{array}
$$

The outermost structure of these two set comprehensions

$\bigcap\,\{$
Trace-refusal pairs in which the trace is the empty sequence
$\}$

$\cup$

$\bigcup\,\{$
Trace-refusal pairs in which the trace is non-empty
$\}$

is due to the outermost external choice in the definitions of ProcLHS and Proc. The middle structure

$\bigcup\,\{$
Trace-refusal pairs
$\}$

is due to the internal choice operator in the definitions of ProcLHS and Proc. These arise from the semantic definitions for internal and external choice that we gave in Section 3.

$$
\begin{array}{lcl}
\mathit{single}[P \,\Box\, Q] & = & (\,\mathit{single}[P] \;\cap\; \mathit{single}[Q]\,) \;\cap\; \{ s : \mathrm{seq}\, \mathit{Event};\; \mathit{ref} : \mathbb{P}\, \mathit{Event} \;\mid\; s = \langle\rangle\,\} \\
& & \cup \\
& & (\,\mathit{single}[P] \;\cup\; \mathit{single}[Q]\,) \;\cap\; \{ s : \mathrm{seq}\, \mathit{Event};\; \mathit{ref} : \mathbb{P}\, \mathit{Event} \;\mid\; s \neq \langle\rangle\,\} \\
\\
\mathit{single}[P \sqcap Q] & = & \mathit{single}[P] \cup \mathit{single}[Q]
\end{array}
$$

Step 3 *Recursive definitions for ProcLHS - setting up our inductive step.*

Our next step is to use the set unions derived in Step 2 to find the necessary and sufficient constraints for trace-refusal pairs to lie in the singleton failures semantics of $ProcLHS$, $single[ProcLHS(a)]$. *The constraints for trace-refusal pairs in which the trace is not the empty sequence will be recursive in the sense that they are expressed in terms of the set* $single[ProcLHS(a')]$. *The constraints for trace-refusal pairs in which the trace is the empty sequence will be non-recursive. They will be used as base cases in the inductive proof in the next step in which we establish the corresponding non-recursive constraints.*

To ease the transition from reasoning from an event-based perspective to reasoning from a state-based perspective we will introduce a function progRel that, for a given data type and sequence of operations—where an operation is a triple recording the name of an operation and its associated input and output values—

$$Op : Name \times Value \times Value$$

returns a relation on the state space. Given data type D, *sequence of operations prog and states* a *and* a', *the maplet* $a \mapsto a'$ *lies in the relation* $progRel\,D\,prog$ *precisely when state* a' *may be reached from state* a *by applying program* $prog$.

$$
\begin{array}{l}
[State, Name, Value] \\
progRel : CDT[State, Name, Value] \nrightarrow \mathrm{seq}\, Op \nrightarrow (State \leftrightarrow State) \\
\hline
\forall\, D : CDT[State, Name, Value]; \\
\qquad\qquad n : Name;\ i, o : Value;\ prog : \mathrm{seq}\, Op \bullet \\
\qquad progRel\, D\, \langle\rangle = \mathrm{id}[State] \\
\qquad \wedge \\
\qquad progRel\, D\, (prog \frown \langle (n, i, o) \rangle) = \\
\qquad\qquad \{\, a, a' : State \mid \\
\qquad\qquad\qquad \exists\, a'' : State \bullet a \mapsto a'' \in progRel\, D\, prog \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad (a'', i) \mapsto (a', o) \in ops\, n \,\}
\end{array}
$$

Given the above definition, and observing that the predicates

$$
\begin{array}{l}
\forall\, n : \mathrm{dom}\, ops \setminus X;\ i : Value \mid \\
\qquad \exists\, b : state \ \bullet\ (a, b) \in (D, X)^{\diamond} \wedge (b, i) \in \mathrm{dom}\, ops\, n \ \bullet \\
\qquad\qquad n \neq name \ \vee\ i \neq in \\
\qquad\qquad \vee \\
\qquad\qquad \exists\, o : Value \mid a \in \mathrm{dom}\, progRel\, D \langle (n, i, o) \rangle \bullet o \neq out
\end{array}
$$

and $\{\mathrm{op}.name.in.out\} \in \mathrm{refusals}\, ProcLHS(a)$ *are equivalent, we can then prove that the following constraints are necessary and sufficient for trace-refusal pairs to lie in the set* $single[ProcLHS(a)]$.

- *The trace-refusal pair is which the trace is the empty sequence and the refusal set is the* *empty trace will* always *lie in the singleton failures semantics of ProcLHS.*

$$\begin{array}{l}
(\langle\rangle, \emptyset) \ \in \ single[\ let \\
\qquad\qquad ProcLHS(a) \ = \\
\qquad\qquad\quad \Box\, n : \mathrm{dom}\, ops \setminus X;\ i : Value \mid \\
\qquad\qquad\qquad \exists\, b : state \ \bullet \ (a,b) \in (D,X)^{\diamond} \\
\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad (b,i) \in \mathrm{dom}\, ops\, n \ \bullet \\
\qquad\qquad\quad \sqcap\, a' : State;\ o : Value \mid \\
\qquad\qquad\qquad \exists\, b, b' : state \ \bullet \\
\qquad\qquad\qquad\qquad \{(a,b),(b',a')\} \subseteq (D,X)^{\diamond} \\
\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad (b,i) \mapsto (b',o) \in ops\, n \ \bullet \\
\qquad\qquad\qquad \mathrm{op}.n.i.o \rightarrow ProcLHS(a') \\
\qquad within \\
\qquad\qquad ProcLHS(a)\,] \\
\Leftrightarrow \\
\quad \mathrm{true}
\end{array}$$

- *Trace-refusal pairs in which the trace is the empty sequence and the refusal set contains a single event* op.*name*.*in*.*out* *for some operation name and values in and out will lie in the singleton failures semantics of ProcLHS provided the event* op.*name*.*in*.*out* *is in the set of initial refusals of process* $ProcLHS(a)$.

$$\begin{array}{l}
(\langle\rangle, \{\mathrm{op}.name.in.out\}) \\
\quad \in \\
\quad single[\ let \\
\qquad\qquad ProcLHS(a) \ = \\
\qquad\qquad\quad \Box\, n : \mathrm{dom}\, ops \setminus X;\ i : Value \mid \\
\qquad\qquad\qquad \exists\, b : state \ \bullet \ (a,b) \in (D,X)^{\diamond} \\
\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad (b,i) \in \mathrm{dom}\, ops\, n \ \bullet \\
\qquad\qquad\quad \sqcap\, a' : State;\ o : Value \mid \\
\qquad\qquad\qquad \exists\, b, b' : state \ \bullet \\
\qquad\qquad\qquad\qquad \{(a,b),(b',a')\} \subseteq (D,X)^{\diamond} \\
\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad (b,i) \mapsto (b',o) \in ops\, n \ \bullet \\
\qquad\qquad\qquad \mathrm{op}.n.i.o \rightarrow ProcLHS(a') \\
\qquad within \\
\qquad\qquad ProcLHS(a)\,] \\
\Leftrightarrow \\
\quad \forall\, n : \mathrm{dom}\, ops \setminus X;\ i : Value \mid \\
\qquad \exists\, b : state \ \bullet (a,b) \in (D,X)^{\diamond} \wedge (b,i) \in \mathrm{dom}\, ops\, n \ \bullet \\
\qquad\quad n \neq name \ \vee \ i \neq in \\
\qquad\quad \vee \\
\qquad\quad \exists\, o : Value \mid a \in \mathrm{dom}\, progRel\, D\langle (n,i,o)\rangle \bullet o \neq out
\end{array}$$

- *Trace-refusal pairs of the form* $(\langle \mathrm{op}.name.in.out\rangle \frown tr, ref)$ *for some trace tr, refusal set ref, named operation name and values in and out—i.e. those trace-refusal pairs in which the trace is non-empty—will lie in the set* $single[ProcLHS(a)]$ *precisely when the singleton sequence* $\langle \mathrm{op}.name.in.out\rangle$ *is a possible trace of* $ProcLHS(a)$ *and there is some state*

a', reachable from a by applying operation $(name, in, out)$ such that the trace-refusal pair (tr, ref) lies in the set $single[ProcLHS(a')]$. More formally,

$$
\begin{array}{l}
(\langle \mathrm{op}.name.in.out\rangle \frown tr, ref) \\
\quad \in \\
\quad\quad single[\ let \\
\qquad\qquad\qquad ProcLHS(a) \ = \\
\qquad\qquad\qquad\quad \Box\, n : \mathrm{dom}\, ops \setminus X;\ i : Value \mid \\
\qquad\qquad\qquad\qquad\quad \exists\, b : state \ \bullet \ (a, b) \in (D, X)^{\diamond} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (b, i) \in \mathrm{dom}\, ops\, n \bullet \\
\qquad\qquad\qquad\qquad \sqcap\, a' : State;\ o : Value \mid \\
\qquad\qquad\qquad\qquad\qquad \exists\, b, b' : state \ \bullet \\
\qquad\qquad\qquad\qquad\qquad\quad \{(a, b), (b', a')\} \subseteq (D, X)^{\diamond} \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge \\
\qquad\qquad\qquad\qquad\qquad\quad (b, i) \mapsto (b', o) \in ops\, n \bullet \\
\qquad\qquad\qquad\qquad\quad \mathrm{op}.n.i.o \rightarrow ProcLHS(a') \\
\qquad\qquad within \\
\qquad\qquad\qquad ProcLHS(a)\,] \\
\quad \Leftrightarrow \\
\quad\quad name \in \mathrm{dom}\, ops \setminus X \\
\quad\quad \wedge \\
\quad\quad \exists\, a' : state \setminus (internal\ D\ X);\ prog : \mathrm{seq}\, Op \mid \\
\qquad\quad prog \upharpoonright (opV\ X) = \langle (name, in, out)\rangle \bullet \\
\qquad\qquad a \mapsto a' \in progRel\ D\ prog \\
\qquad\qquad \wedge \\
\qquad\qquad (tr, ref) \in single[\, ProcLHS(a')\,]
\end{array}
$$

where the set $opV\ X$ contain all those operations that will remain visible.

$$
\begin{array}{|l}
\hline
[State, Name, Value] \\
opV, opH : \mathbb{P}\, Name \rightarrow \mathbb{P}\, Op \\
\hline
\forall\, X : \mathbb{P}\, Name \bullet \\
\qquad opV\ X \ = \ \{\, n : Name;\ i, o : Value \mid n \notin X \,\} \\
\qquad opH\ X \ = \ \{\, n : Name;\ i, o : Value \mid n \in X \,\} \\
\hline
\end{array}
$$

Step 4 *Non-recursive definitions for ProcLHS - inductive proof.*

Having established in Step 3 the recursive conditions for a general trace-refusal pair to lie in the singleton failures semantics of process procLHS we now do two inductive proofs to establish the non-recursive constraints for trace-refusal pairs to lie in the singleton failures semantics of process procLHS. The first proof is for the case when the refusal set is empty and the second proof is for the case when the refusal set is non-empty. We do our induction on the length of the trace using the instances—also established in Step 3—in which the trace is the empty trace as our base cases.

First we introduce the function seqEventToSeqOp which converts sequences of events to sequences of operations.

$$
\begin{array}{|l}
seqEventToSeqOp : \mathrm{seq}\, Event \rightarrow \mathrm{seq}\, Op \\
\hline
seqEventToSeqOp\ \langle\rangle = \langle\rangle \\
\wedge \\
\forall\, n : Name;\ i, o : Value;\ tr : \mathrm{seq}\, Event \bullet \\
\qquad seqEventToSeqOp\ \langle \mathrm{op}.n.i.o \rangle \frown tr\ = \\
\qquad\qquad\qquad \langle (n, i, o) \rangle \frown (seqEventToSeqOp\ tr)
\end{array}
$$

- *A general trace-refusal pair* $(tr, \emptyset)$*, in which the refusal set is empty, lies in the singleton failures semantics of procLHS precisely when the trace contains only visible events and there is some program of events that may be performed which, when restricted to visible events, gives the desired trace. More formally,*

$$
\begin{array}{l}
\mathrm{ran}\, tr\ \subseteq\ \{ n : \mathrm{dom}\, ops \setminus X;\ i, o : Value \bullet \mathrm{op}.n.i.o \} \\
\wedge \\
\exists\, a' : state \setminus (internal\ D\ X);\ prog : \mathrm{seq}\, Op \mid \\
\qquad seqEventToSeqOp\ tr\ =\ prog \upharpoonright (opV\ X) \bullet \\
\qquad\qquad a \mapsto a' \in progRel\ D\ prog
\end{array}
$$

- *A general trace-refusal pair* $(tr, \{\mathrm{op}.name.in.out\})$*, in which the refusal set is non-empty, lies in the singleton failures semantics of procLHS precisely when the trace contains only visible events and there is some program of events that may be performed after which the event in the refusal set may be refused and which, when restricted to visible events, gives the desired trace. More formally,*

$$
\begin{array}{l}
\mathrm{ran}\, tr\ \subseteq\ \{ n : \mathrm{dom}\, ops \setminus X;\ i, o : Value \bullet \mathrm{op}.n.i.o \} \\
\wedge \\
\exists\, a' : state \setminus (internal\ D\ X);\ prog : \mathrm{seq}\, Op \mid \\
\qquad seqEventToSeqOp\ tr\ =\ prog \upharpoonright (opV\ X) \bullet \\
\qquad\qquad a \mapsto a' \in progRel\ D\ prog \\
\qquad\qquad \wedge \\
\qquad\qquad \forall\, n : \mathrm{dom}\, ops \setminus X;\ i : Value \mid \\
\qquad\qquad\qquad\qquad \exists\, b : state \bullet (a', b) \in (D, X)^{\diamond} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (b, i) \in \mathrm{dom}\, ops\ n \bullet \\
\qquad\qquad\qquad n \neq name\ \vee\ i \neq in \\
\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad \exists\, o : Value \mid a' \in \mathrm{dom}\, progRel\ D \langle (n, i, o) \rangle \bullet \\
\qquad\qquad\qquad\qquad o \neq out
\end{array}
$$

Step 5 *Constraints for single*$[\, process_{io}\ (hide\ D\ X)\,]$.

Having established the constraints necessary for trace-refusal pair to lie in the singleton failures semantics of the process $ProcLHS(a)$ *we are now able to determine the constraints necessary for a trace-refusal pair to lie in the singleton failures semantics of process* $process_{io}\ (hide\ D\ X)$.

- *A general trace-refusal pair* $(tr, \emptyset)$*, in which the refusal set is empty, lies in the singleton failures semantics of* $process_{io}\ (hide\ D\ X)$ *precisely when the trace contains only visible events and there is some program of events that may be performed from some initial state* *which, when restricted to visible events, gives the desired trace. More formally,*

$$\begin{array}{l}
\operatorname{ran} tr \subseteq \{n : \operatorname{dom} ops \setminus X;\ i, o : Value \bullet \mathrm{op}.n.i.o\} \\
\wedge \\
\exists\, a_i : init;\ a' : state \setminus (internal\ D\ X);\ prog : \operatorname{seq} Op \mid \\
\quad seqEventToSeqOp\ tr\ =\ prog \upharpoonright (opV\ X) \bullet \\
\quad\quad a_i \mapsto a' \in progRel\ D\ prog
\end{array}$$

- *A general trace-refusal pair* $(tr, \{\mathrm{op}.name.in.out\})$*, in which the refusal set is non-empty, lies in the singleton failures semantics of* $process_{io}\ (hide\ D\ X)$ *precisely when the trace contains only visible events and there is some program of events that may be performed from some initial state after which the event in the refusal set may be refused and such that the program restricted to visible events gives the desired trace. More formally,*

$$\begin{array}{l}
\operatorname{ran} tr \subseteq \{n : \operatorname{dom} ops \setminus X;\ i, o : Value \bullet \mathrm{op}.n.i.o\} \\
\wedge \\
\exists\, a_i : init;\ a' : state \setminus (internal\ D\ X);\ prog : \operatorname{seq} Op \mid \\
\quad seqEventToSeqOp\ tr\ =\ prog \upharpoonright (opV\ X) \bullet \\
\quad\quad a_i \mapsto a' \in progRel\ D\ prog \\
\quad\quad \wedge \\
\quad\quad \forall\, n : \operatorname{dom} ops \setminus X;\ i : Value \mid \\
\quad\quad\quad\quad \exists\, b : state \bullet (a', b) \in (D, X)^{\diamond} \\
\quad\quad\quad\quad\quad\quad \wedge \\
\quad\quad\quad\quad\quad\quad (b, i) \in \operatorname{dom} ops\ n \bullet \\
\quad\quad\quad n \neq name\ \vee\ i \neq in \\
\quad\quad\quad \vee \\
\quad\quad\quad \exists\, o : Value \mid a' \in \operatorname{dom} progRel\ D\langle (n, i, o)\rangle \bullet \\
\quad\quad\quad\quad o \neq out
\end{array}$$

Steps 6, 7 and 8 *Constraints for Proc and single*$[\,process_{io}\ D]$*.*

Steps 6 and 7 are similar to Steps 3 and 4 but for the simpler process Proc. We determine—first recursively and then non-recursively—constraints that trace-refusal pairs must satisfy in order to lie in the singleton failures semantics of process Proc. Once more we use the recursive constraints established in Step 6 in our proof of the non-recursive constraints in Step 7. Note that in these steps we need consider only those trace-refusal pairs in which the refusal set is non-empty since we are concerned only with events that might be refused in single$[\,(\,process_{io}\ D\,) \setminus (\,eventH\ X\,)\,]$*.*

Step 8, taking initialisation into consideration, is similar to Step 5 but for the simpler process $process_{io}\ D$*. We use the non-recursive constraints established in Step 7 to determine the constraints that trace-refusal pairs in which the refusal set is non-empty must satisfy if they are to lie in the singleton failures semantics of single*$[\,process_{io}\ D]$*. We give these constraints below.*

- *A general trace-refusal pair* $(tr, \{\mathrm{op}.name.in.out\})$*, in which the refusal set is non-empty, lies in the singleton failures semantics of* $process_{io}\ D$ *precisely when the there is some program of events corresponding to the trace that may be performed from some initial state* *after which the event in the refusal set may be refused. More formally,*

$$
\begin{array}{l}
\operatorname{ran} tr \;\subseteq\; \{n : \operatorname{dom} ops;\; i, o : Value \bullet \mathrm{op}.n.i.o\} \\
\wedge \\
\exists\, a_i : init;\; a' : State \mid \\
\qquad\qquad a_i \mapsto a' \in progRel\, D\,(\, seqEventToSeqOp\; tr\,) \bullet \\
\qquad \forall\, n : \operatorname{dom} ops;\; i : Value \mid (a', i) \in \operatorname{dom} ops\; n \bullet \\
\qquad\qquad n \neq name \;\vee\; i \neq in \\
\qquad\qquad \vee \\
\qquad\qquad \exists\, o : Value \mid \\
\qquad\qquad\qquad a' \in \operatorname{dom} progRel\, D\langle (n, i, o)\rangle \bullet o \neq out
\end{array}
$$

Step 9 *Interesting step.*

Step 9 is the interesting step! Having established in the first eight steps the conditions necessary for trace-refusal pairs to lie in the sets $single[process_{io}\,(hide\, D\, X)]$ *and* $single[process_{io}\, D]$, *in this step we determine the conditions necessary for a trace refusal pair to lie in the in the set* $single[(process_{io}\, D) \setminus X]$. *Then, subdividing the problem further and considering the following three cases separately*

Case 2a *Trace-refusal pairs in which the refusal set is empty;*

Case 2b *Trace-refusal pairs in which the singleton event in the refusal set is one of the events to be hidden;*

Case 2c *Trace-refusal pairs in which the singleton event in the refusal set is one of the remaining visible events.*

we prove that in each case the trace-refusal pair under consideration lies in the singleton failures semantics of the process $process_{io}\,(hide\, D\, X)$ *precisely when it lies in the singleton failures semantics of* $(\, process_{io}\, D\,) \setminus (\, eventH\; X\,)$. *In each case we establish the desired result by sliding back and forth between notions and concepts that lie naturally either within the data type world or within the process world.*

Step 10 *Combining our results.*

In Step 10 we pull together the results from Case 1 and from Cases 2a, 2b and 2c in Step 9 to get our desired result. We conclude that our theorem holds: for all data types and sets of names of operations such that the predicate hideable is true, hiding on the data type is equivalent to hiding on its corresponding process.

B.5 Combining data types using parallel composition and hiding

In this section we define the parallel operator on our data types and prove that our definition of the parallel composition of data types is consistent with the definition of the parallel composition of their corresponding processes.

We then go on to illustrate how the parallel operator may be used in conjunction with the hiding operator to combine data types and reason about their combined behaviour. We will present a simple example of a service provider that may be broken down into separate client and server components. We conclude the chapter with a discussion of what it *means* to put two communicating data types in parallel.

$$
\begin{tikzcd}
 & \mathrm{A,\ B} \arrow[dl, "parallel"'] \arrow[dr, "process_{io}"] & \\
parallel\,(A, B) \arrow[d, "process_{io}"'] & & process_{io}\, A, process_{io}\, B \arrow[d, "\parallel"] \\
process_{io}\,(parallel\,(A, B)) \arrow[rr, leftrightarrow, "single[\,]"] & & process_{io}\, A \parallel process_{io}\, B
\end{tikzcd}
$$

Figure 5: The process corresponding to the parallel composition of data types A and B is semantically equivalent to the parallel composition of the process corresponding to A and the process corresponding to B.

B.5.1 Parallel composition of data types

We define the parallel composition of communicating data types in the natural way: the state space of the resulting data type is the cross product of the state spaces of the two given data types; the set of possible initial states is the cross product of the sets of possible initial states of each of the data types; and the set of operations on the resulting data type is the union of the set of operations on the two given data types.

Given communicating data types A and B respectively of types $CDT[AState, Name, Value]$ and $CDT'[BState, Name, Value]$—we prime the second data type in order to distinguish between them—we write $parallel\,(A, B)$ to denote the communicating data type obtained from their parallel combination. The state space of $parallel\,(A, B)$ is then $state \times state'$, the set of initial states is $init \times init'$ and the set of operations on $parallel\,(A, B)$ is $\mathrm{dom}\ ops \cup \mathrm{dom}\ ops'$.

The definition of the relation describing each operation on the resulting data type depends on whether that operation is defined on precisely one or on both of the data types to be put in parallel. In the case when the operation is defined on precisely one of the data types, the component of the state space corresponding to that data type is changed whereas the component of the state space corresponding to the other data type is left unaltered.

Suppose, for instance, that n is the name of an operation on data type A but not on data type B. Then, for any states (a, b) and (a', b') in the state space of the data type $parallel\,(A, B)$ and any input and output values in and out, the maplet $((a, b), in) \mapsto ((a', b'), out)$ lies in the relation describing the operation n on data type $parallel\,(A, B)$ precisely when b' is equal to b and the maplet $(a, in) \mapsto (a', out)$ lies in the relation describing the operation n on A.

Similarly, if n is an operation on B but not on A, then $((a, b), in) \mapsto ((a', b'), out)$ lies in the relation describing operation n on $parallel\,(A, B)$ precisely when a' is equal to a and the maplet $(b, in) \mapsto (b', out)$ lies in the relation describing the operation n on B.

Operations defined on both data types will be available only when both data types are willing to communicate the operation. If n is the name of an operation on both data types A and B then, for any states (a, b) and (a', b') in the state space of the data type $parallel\,(A, B)$ and any input and output values in and out, then the maplet $((a, b), in) \mapsto ((a', b'), out)$ lies in the relation describing the operation n on data type $parallel\,(A, B)$ precisely when $(a, in) \mapsto (a', out)$ lies in the relation describing the operation n on A and $(b, in) \mapsto (b', out)$ lies in the relation describing the operation n on B.

Given these constraints we formally define the parallel operator on data types in the following manner:

$$
\begin{array}{l}
[AState, BState, Name, Value] \\
\hline
parallel : (\ CDT[AState, Name, Value] \times \\
\qquad\qquad CDT[BState, Name, Value]\ ) \pfun \\
\qquad\qquad\qquad CDT[AState \times BState, Name, Value] \\
\hline
\forall A : CDT[AState, Name, Value];\ B : CDT'[BState, Name, Value] \bullet \\
\quad parallel\,(A, B)\ = \\
\qquad (\ \mu\, CDT''[State, Name, Value] \mid \\
\qquad\quad state''\ =\ state \times state'\ \wedge \\
\qquad\quad init''\ =\ init \times init'\ \wedge \\
\qquad\quad \mathrm{dom}\, ops'' = \mathrm{dom}\, ops\ \cup\ \mathrm{dom}\, ops'\ \wedge \\
\qquad\quad \forall n : \mathrm{dom}\, ops \setminus \mathrm{dom}\, ops' \bullet \\
\qquad\qquad ops''\, n\ = \\
\qquad\qquad\quad \{a, a' : state;\ b : state';\ in, out : Value \mid \\
\qquad\qquad\qquad (a, in) \mapsto (a', out) \in ops\, n \bullet \\
\qquad\qquad\quad ((a, b), in) \mapsto ((a', b), out)\}\ \wedge \\
\qquad\quad \forall n : \mathrm{dom}\, ops' \setminus \mathrm{dom}\, ops \bullet \\
\qquad\qquad ops''\, n\ = \\
\qquad\qquad\quad \{a : state;\ b, b' : state';\ in, out : Value \mid \\
\qquad\qquad\qquad (b, in) \mapsto (b', out) \in ops'\, n \bullet \\
\qquad\qquad\quad ((a, b), in) \mapsto ((a, b'), out)\}\ \wedge \\
\qquad\quad \forall n : \mathrm{dom}\, ops \cap \mathrm{dom}\, ops' \bullet \\
\qquad\qquad ops''\, n\ = \\
\qquad\qquad\quad \{a, a' : state;\ b, b' : state';\ in, out : Value \mid \\
\qquad\qquad\qquad (a, in) \mapsto (a', out) \in ops\, n \\
\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad (b, in) \mapsto (b', out) \in ops'\, n \bullet \\
\qquad\qquad\quad ((a, b), in) \mapsto ((a', b'), out)\}\ ).
\end{array}
$$

B.5.2 Consistency of the parallel operators

In this section we present a healthiness condition—and the associated predicate *parallelable*—that ensures that the parallel combination of two data types is consistent with their parallel combination of their corresponding processes.

No problems arise when we consider operations that are defined only on one data type. Furthermore, for operations on both data types, no problems arise if we are in a state from which the operation and given input and output values will always be blocked by the parallel combination or if we are in a state from which the operation and given input and output values will never be blocked by the parallel combination. We must, however, be careful if we are in a state from which the chosen operation with specified input and output may be performed or may be blocked.

From a CSP perspective, this may occur when *either* of the processes is able to perform the same operation with the same input value but with a different output value. Whereas, from the state-based perspective, the operation may be blocked only if there is a different output value which could be communicated by *both* data types given the same operation and input value.

Given data types A and B of types $CDT[AState, Name, Value]$ and $CDT'[BState, Name, Value]$ respectively, our healthiness condition for parallel composition is therefore as follows:

$$\begin{array}{l}
\forall\, a : state;\ b : state';\ n : (\mathrm{dom}\, ops \cap \mathrm{dom}\, ops');\ in, out : Value \mid \\
\qquad \exists\, a' : state;\ b' : state' \bullet (a, in) \mapsto (a', out) \in ops\, n \\
\qquad\qquad \wedge \\
\qquad\qquad (b, in) \mapsto (b', out) \in ops'\, n \;\bullet \\
\quad \exists\, o : Value \setminus \{out\} \bullet \exists\, a'' : state \bullet (a, in) \mapsto (a'', o) \in ops\, n \\
\qquad\qquad \vee \\
\qquad\qquad \exists\, b'' : state' \bullet (b, in) \mapsto (b'', o) \in ops'\, n \\
\qquad \Rightarrow \\
\qquad\quad \exists\, o : Value \setminus \{out\} \bullet \exists\, a'' : state \bullet (a, in) \mapsto (a'', o) \in ops\, n \\
\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad \exists\, b'' : state' \bullet (b, in) \mapsto (b'', o) \in ops'\, n.
\end{array} \tag{3}$$

Given the above healthiness condition for the parallel composition of data types we can define the associated predicate *parallelable* that must be true if we wish to put two data types in parallel.

$$\begin{array}{l}
[AState, BState, Name, Value] \\
parallelable : \\
\qquad (CDT[AState, Name, Value] \times CDT[BState, Name, Value]) \leftrightarrow Bool \\
\hline
\forall\, A : CDT[AState, Name, Value];\ B : CDT'[BState, Name, Value] \bullet \\
\qquad parallelable\,(A, B) \;= \\
\qquad\quad \forall\, a : state;\ b : state';\ n : (\mathrm{dom}\, ops \cap \mathrm{dom}\, ops');\ in, out : Value \mid \\
\qquad\qquad \exists\, a' : state;\ b' : state' \bullet (a, in) \mapsto (a', out) \in ops\, n \\
\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad (b, in) \mapsto (b', out) \in ops'\, n \;\bullet \\
\qquad\quad \exists\, o : Value \setminus \{out\} \bullet \exists\, a'' : state \bullet (a, in) \mapsto (a'', o) \in ops\, n \\
\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad \exists\, b'' : state' \bullet (b, in) \mapsto (b'', o) \in ops'\, n \\
\qquad\qquad \Rightarrow \\
\qquad\qquad\quad \exists\, o : Value \setminus \{out\} \bullet \exists\, a'' : state \bullet (a, in) \mapsto (a'', o) \in ops\, n \\
\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad \exists\, b'' : state' \bullet (b, in) \mapsto (b'', o) \in ops'\, n
\end{array}$$

The theorem that the parallel composition of our communicating data types is consistent with the parallel composition of their corresponding processes can then be stated formally as follows:

Theorem 5

$$\begin{array}{l}
\forall\, A : CDT[AState, Name, Value];\ B : CDT[BState, Name, Value] \mid \\
\qquad parallelable\,(A, B) \bullet \\
\quad single[\, process_{io}\,(parallel\,(A, B))\,] \\
\qquad = \\
\qquad\quad single[\,(\, process_{io}\, A\,) \parallel (\, process_{io}\, B\,)\,].
\end{array}$$

The proof of this theorem is straight-forward and is therefore omitted.

In Example 1 below we give a simple example of two components, a client and a server, that communicate with one another to determine whether or not a service that has been requested should be offered.

Example 1 *The parallel composition of two data types.*

A client, upon receiving a request to supply a particular service, checks with its server as to whether or not it may provide the requested service. If it is given permission then the server updates its records and the client provides the service. Otherwise the server does not update the system and no service is offered. Such a system may be described by the state-transition diagrams shown in Figure 6 below.

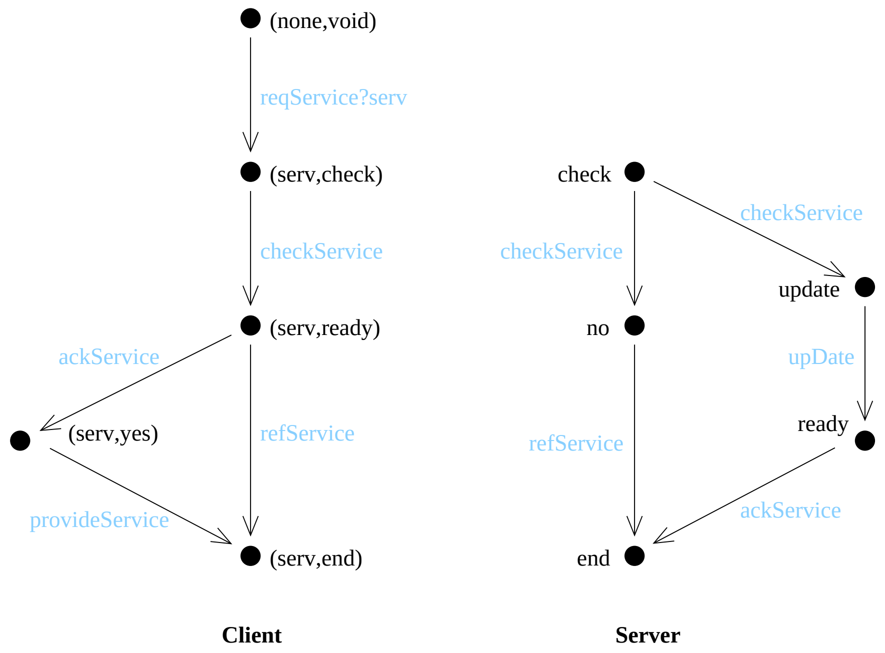

Figure 6: State-transition diagrams for data types *Client* and *Server*.

Given the type Service including the element none:

$[Service]$ | $none : Service$

and the type Pos that can take the seven distinct values, void, check, ready, update, yes, no and end:

$Pos ::= void \mid check \mid update \mid ready \mid yes \mid no \mid end$

the data type corresponding to Client may be defined as follows:

$$
\begin{array}{l}
\mathit{Client} \ = \\
\quad (\, \mu\, \mathit{CDT}[\mathit{Service} \times \mathit{Pos}, \mathit{Name}, \mathit{Service}] \bullet \\
\qquad \mathit{state} \ = \ \mathit{Service} \times \{\mathit{void}, \mathit{ready}, \mathit{yes}, \mathit{end}\} \ \wedge \\
\qquad \mathit{init} \ = \ \{(\mathit{none}, \mathit{void})\} \ \wedge \\
\qquad \mathrm{dom}\, \mathit{ops} \ = \ \{\mathit{reqService}, \mathit{checkService}, \mathit{ackService}, \\
\qquad\qquad\qquad\qquad\qquad \mathit{refService}, \mathit{provideService}\} \ \wedge \\
\qquad \mathit{ops}\, \mathit{reqService} \ = \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet ((\mathit{none}, \mathit{void}), \mathit{serv}) \mapsto ((\mathit{serv}, \mathit{check}), \mathit{null})\} \ \wedge \\
\qquad \mathit{ops}\, \mathit{checkService} \ = \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet ((\mathit{serv}, \mathit{check}), \mathit{serv}) \mapsto ((\mathit{serv}, \mathit{ready}), \mathit{null})\} \ \wedge \\
\qquad \mathit{ops}\, \mathit{ackService} \ = \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet ((\mathit{serv}, \mathit{ready}), \mathit{null}) \mapsto ((\mathit{serv}, \mathit{yes}), \mathit{null})\} \ \wedge \\
\qquad \mathit{ops}\, \mathit{refService} \ = \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet ((\mathit{serv}, \mathit{ready}), \mathit{null}) \mapsto ((\mathit{serv}, \mathit{end}), \mathit{null})\} \ \wedge \\
\qquad \mathit{ops}\, \mathit{provideService} \ = \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet ((\mathit{serv}, \mathit{yes}), \mathit{null}) \mapsto ((\mathit{serv}, \mathit{end}), \mathit{serv})\}\,).
\end{array}
$$

Similarly, the data type corresponding to the server may be defined as follows:

$$
\begin{array}{l}
\mathit{Server} \ = \\
\quad (\, \mu\, \mathit{CDT}[\mathit{Pos}, \mathit{Name}, \mathit{Service}] \bullet \\
\qquad \mathit{state} \ = \ \{\mathit{check}, \mathit{update}, \mathit{ready}, \mathit{no}, \mathit{end}\} \ \wedge \\
\qquad \mathit{init} \ = \ \{\mathit{check}\} \ \wedge \\
\qquad \mathrm{dom}\, \mathit{ops} \ = \ \{\mathit{checkService}, \mathit{upDate}, \mathit{ackService}, \mathit{refService}\} \ \wedge \\
\qquad \mathit{ops}\, \mathit{checkService} \ = \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet (\mathit{check}, \mathit{serv}) \mapsto (\mathit{update}, \mathit{null})\} \\
\qquad\quad \cup \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet (\mathit{check}, \mathit{serv}) \mapsto (\mathit{no}, \mathit{null})\} \ \wedge \\
\qquad \mathit{ops}\, \mathit{upDate} \ = \ \{(\mathit{update}, \mathit{null}) \mapsto (\mathit{ready}, \mathit{null})\} \ \wedge \\
\qquad \mathit{ops}\, \mathit{ackService} \ = \ \{(\mathit{ready}, \mathit{null}) \mapsto (\mathit{end}, \mathit{null})\} \ \wedge \\
\qquad \mathit{ops}\, \mathit{refService} \ = \ \{(\mathit{no}, \mathit{null}) \mapsto (\mathit{end}, \mathit{null})\}\,).
\end{array}
$$

We may then use the parallel operator to obtain a data type that displays the combined be-

haviour of the server and the client.

$$
\begin{array}{l}
\mathit{parallel}\,(\mathit{Client}, \mathit{Server}) \;=\; \\
\quad (\,\mu\, \mathit{CDT}[\mathit{Service} \times \mathit{Pos} \times \mathit{Pos}, \mathit{Name}, \mathit{Service}] \bullet \\
\qquad \mathit{state} \;=\; \mathit{Service} \times \{\mathit{void}, \mathit{check}, \mathit{ready}, \mathit{end}\} \times \\
\qquad\qquad\qquad\qquad \{\mathit{check}, \mathit{update}, \mathit{yes}, \mathit{no}, \mathit{end}\} \;\wedge \\
\qquad \mathit{init} \;=\; \{(\mathit{none}, \mathit{void}, \mathit{check})\} \;\wedge \\
\qquad \mathrm{dom}\, \mathit{ops} \;=\; \{\mathit{reqService}, \mathit{checkService}, \mathit{upDate}, \\
\qquad\qquad \mathit{ackService}, \mathit{refService}, \mathit{provideService}\} \;\wedge \\
\qquad \mathit{ops}\; \mathit{reqService} \;=\; \\
\qquad\quad \{\mathit{serv} : \mathit{Service};\; \mathit{pos} : \{\mathit{check}, \mathit{update}, \mathit{ready}, \mathit{no}, \mathit{end}\} \bullet \\
\qquad\qquad ((\mathit{none}, \mathit{void}, \mathit{pos}), \mathit{serv}) \mapsto ((\mathit{serv}, \mathit{check}, \mathit{pos}), \mathit{null})\} \;\wedge \\
\qquad \mathit{ops}\; \mathit{checkService} \;=\; \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet \\
\qquad\qquad ((\mathit{serv}, \mathit{check}, \mathit{check}), \mathit{serv}) \mapsto ((\mathit{serv}, \mathit{ready}, \mathit{update}), \mathit{null})\} \\
\qquad\quad \cup \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet \\
\qquad\qquad ((\mathit{serv}, \mathit{check}, \mathit{check}), \mathit{serv}) \mapsto ((\mathit{serv}, \mathit{ready}, \mathit{no}), \mathit{null})\} \;\wedge \\
\qquad \mathit{ops}\; \mathit{upDate} \;=\; \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet \\
\qquad\qquad ((\mathit{none}, \mathit{void}, \mathit{update}), \mathit{null}) \mapsto ((\mathit{none}, \mathit{void}, \mathit{ready}), \mathit{null})\} \\
\qquad\quad \cup \\
\qquad\quad \{\mathit{serv} : \mathit{Service};\; \mathit{pos} : \{\mathit{void}, \mathit{ready}, \mathit{yes}, \mathit{end}\} \bullet \\
\qquad\qquad ((\mathit{serv}, \mathit{pos}, \mathit{update}), \mathit{null}) \mapsto ((\mathit{serv}, \mathit{pos}, \mathit{ready}), \mathit{null})\} \;\wedge \\
\qquad \mathit{ops}\; \mathit{ackService} \;=\; \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet \\
\qquad\qquad ((\mathit{serv}, \mathit{ready}, \mathit{ready}), \mathit{null}) \mapsto ((\mathit{serv}, \mathit{yes}, \mathit{end}), \mathit{null})\} \;\wedge \\
\qquad \mathit{ops}\; \mathit{refService} \;=\; \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet \\
\qquad\qquad ((\mathit{serv}, \mathit{ready}, \mathit{no}), \mathit{null}) \mapsto ((\mathit{serv}, \mathit{end}, \mathit{end}), \mathit{null})\} \;\wedge \\
\qquad \mathit{ops}\; \mathit{provideService} \;=\; \\
\qquad\quad \{\mathit{serv} : \mathit{Service};\; \mathit{pos} : \{\mathit{check}, \mathit{update}, \mathit{ready}, \mathit{no}, \mathit{end}\} \bullet \\
\qquad\qquad ((\mathit{serv}, \mathit{yes}, \mathit{pos}), \mathit{null}) \mapsto ((\mathit{serv}, \mathit{end}, \mathit{pos}), \mathit{serv})\}\,).
\end{array}
$$

In Example 1 above we illustrated how two data types, a client and a server, may be put in parallel. In Example 3 we will suppose that at an earlier stage in the specification and development of the system a single *ServiceProvider* component with operations *reqService*, *upDate* and *provideService* as illustrated in Figure 7 had been specified, and that this was then split into the subcomponents *Client* and *Server*. We will show how refinement checks may be used to verify that the combined behaviour of the *Client* and *Server* components satisfies the specified behaviour of the *ServiceProvider* component.

First observe that the data type *parallel* (*Client*, *Server*) obtained in Example 1 above is unnecessarily complicated. This is because the state space is significantly larger than it need be. This may often be the case for data types that have been put in parallel: each operation that was originally defined on precisely one of the data types will often be defined on the resultant data type for many unreachable states. If this is the case then it may be convenient to look for a simpler yet equivalent data type and the appropriate retrieve relation. We illustrate this in Example 2 below.

Example 2 *Reducing the state space.*

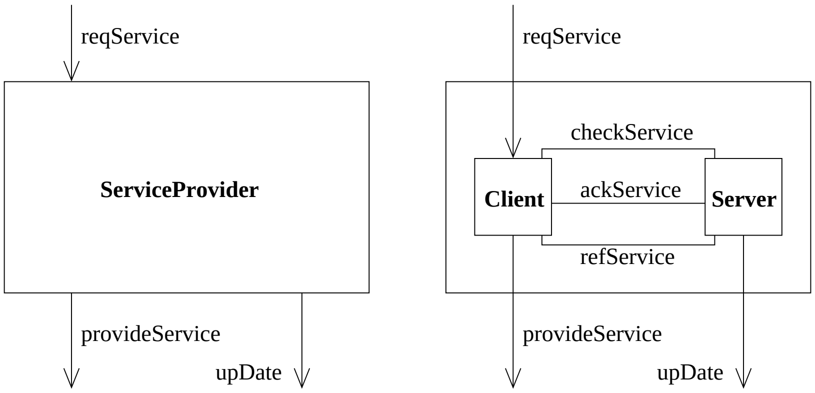

Figure 7: The *ServiceProvider* component may be split into *Client* and *Server* Components.

In this example we identify a data type ClientServer that is equivalent in behaviour to the data type parallel $(\mathit{Client}, \mathit{Server})$*, as constructed in Example 1 but which has a significantly smaller state space. We identify a retrieve relation that can be used to show the equivalence of the two data types.*

Consider Figure 8 below. The states illustrated in this state transition system correspond to all those states within the data type parallel $(\mathit{Client}, \mathit{Server})$ *that are reachable.*

Having identified the reachable states of our combined data type we may then define a simpler data type with a smaller state space that has the same behaviour. The data type ClientServer whose state transition system is presented in Figure 8 may be formally defined as follows:

$$
\begin{array}{l}
\mathit{ClientServer} \ = \\
\quad (\mu\, \mathit{CDT}[\mathit{Service} \times \mathit{Pos}, \mathit{Name}, \mathit{Service}] \bullet \\
\qquad \mathit{state} \ = \ \mathit{Service} \times \mathit{Pos} \ \wedge \\
\qquad \mathit{init} \ = \ \{(\mathit{none}, \mathit{void})\} \ \wedge \\
\qquad \mathrm{dom}\, \mathit{ops} \ = \ \{\mathit{reqService}, \mathit{checkService}, \mathit{upDate}, \\
\qquad\qquad\qquad\quad \mathit{ackService}, \mathit{refService}, \mathit{provideService}\} \ \wedge \\
\qquad \mathit{ops}\ \mathit{reqService} \ = \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet ((\mathit{none}, \mathit{void}), \mathit{serv}) \mapsto ((\mathit{serv}, \mathit{check}), \mathit{null})\} \ \wedge \\
\qquad \mathit{ops}\ \mathit{checkService} \ = \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet ((\mathit{serv}, \mathit{check}), \mathit{serv}) \mapsto ((\mathit{serv}, \mathit{update}), \mathit{null})\} \\
\qquad\quad \cup \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet ((\mathit{serv}, \mathit{check}), \mathit{serv}) \mapsto ((\mathit{serv}, \mathit{no}), \mathit{null})\} \ \wedge \\
\qquad \mathit{ops}\ \mathit{upDate} \ = \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet ((\mathit{serv}, \mathit{update}), \mathit{null}) \mapsto ((\mathit{serv}, \mathit{ready}), \mathit{null}) \ \wedge \\
\qquad \mathit{ops}\ \mathit{ackService} \ = \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet ((\mathit{serv}, \mathit{ready}), \mathit{null}) \mapsto ((\mathit{serv}, \mathit{yes}), \mathit{null})\} \ \wedge \\
\qquad \mathit{ops}\ \mathit{refService} \ = \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet ((\mathit{serv}, \mathit{no}), \mathit{null}) \mapsto ((\mathit{serv}, \mathit{end}), \mathit{null})\} \ \wedge \\
\qquad \mathit{ops}\ \mathit{provideService} \ = \\
\qquad\quad \{\mathit{serv} : \mathit{Service} \bullet ((\mathit{serv}, \mathit{yes}), \mathit{null}) \mapsto ((\mathit{serv}, \mathit{end}), \mathit{serv})\}\ ).
\end{array}
$$

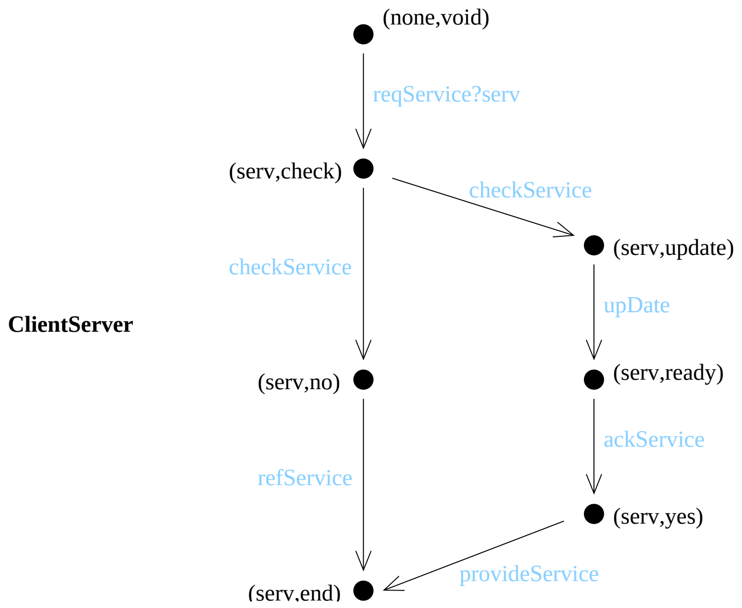

Figure 8: State-transition diagram for *ClientServer*.

Using the retrieve relation Retr described below:

$$\begin{aligned} Retr \;\; = \;\; \{\, & serv : Service \bullet \\ & (none, void, check) \mapsto (none, void), (serv, check, check) \mapsto (serv, check), \\ & (serv, ready, update) \mapsto (serv, update), (serv, ready, no) \mapsto (serv, no), \\ & (serv, ready, ready) \mapsto (serv, ready), (serv, yes, end) \mapsto (serv, yes), \\ & (serv, end, end) \mapsto (serv, end) \,\} \end{aligned}$$

we can easily show that parallel $(Client, Server)$ *forward simulates ClientServer and that ClientServer backward simulates parallel* $(Client, Server)$*. From this we conclude that the two data types are equivalent.*

In Example 3 below we verify that the combined behaviour of the subcomponents *Client* and *Server* satisfies the specification of their parent component *ServiceProvider*.

Example 3 *Verifying consistency.*
The state transition diagram for the data type SericeProvider is illustrated in Figure 9 below. Having received a request for a service, the service provider may non-deterministically refuse to participate in any further operations or may update its records and provide the requested service.

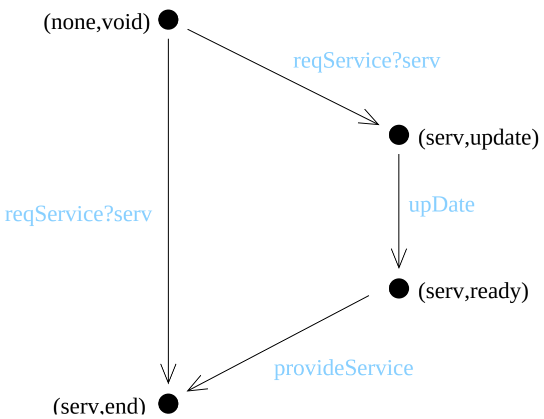

Figure 9: State-transition diagram for $ServiceProvider$.

The data type ServiceProvider is formally defined as follows:

$$
\begin{array}{l}
ServiceProvider \ = \\
\quad (\, \mu\, CDT[Service \times Pos, Name, Service] \bullet \\
\qquad state \ = \ \{(none, void)\} \\
\qquad \qquad \quad \cup \\
\qquad \qquad \quad \{serv : Service;\ pos : \{update, ready, end\} \bullet (serv, pos)\} \ \wedge \\
\qquad init \ = \ \{(none, void)\} \ \wedge \\
\qquad \mathrm{dom}\ ops \ = \ \{reqService, upDate, provideService\} \ \wedge \\
\qquad ops\ reqService \ = \\
\qquad \quad \{serv : Service;\ pos : \{end, update\} \bullet \\
\qquad \qquad ((none, void), serv) \mapsto ((serv, pos), null)\} \ \wedge \\
\qquad ops\ upDate \ = \\
\qquad \quad \{serv : Service \bullet ((serv, update), null) \mapsto ((serv, ready), null) \ \wedge \\
\qquad ops\ provideService \ = \\
\qquad \quad \{serv : Service \bullet ((serv, ready), null) \mapsto ((serv, end), serv)\}\,).
\end{array}
$$

We prove that the parallel combination of data types Client and Server satisfies the description of data type ServiceProvider reasoning as follows:

$$
\begin{array}{ll}
ServiceProvider & \\
\quad \sqsubseteq_{relation} & \\
\qquad hide\,(\, parallel\,(Client, Server), \{checkService, refService, ackService\}\,) & \\
\Leftrightarrow & \text{[Example 2]} \\
\quad ServiceProvider & \\
\qquad \sqsubseteq_{relation} & \\
\qquad \quad hide\,(\ ClientServer, \{checkService, refService, ackService\}\,) & \\
\Leftrightarrow & \text{[Theorem 2]} \\
\quad process_{io}\ ServiceProvider & \\
\qquad \sqsubseteq_{single} & \\
\qquad \quad process_{io}\,(hide\,(\ ClientServer, \{checkService, refService, ackService\}\,)) &
\end{array}
$$

$\Leftrightarrow$ *[Theorem 4]*

$$process_{io}\ ServiceProvider \sqsubseteq_{single} (process_{io}\ ClientServer) \setminus \{checkService, refService, ackService\}$$

$\Leftarrow$ *[Theorem 1]*

$$process_{io}\ ServiceProvider \sqsubseteq_{failures} (process_{io}\ ClientServer) \setminus \{checkService, refService, ackService\}$$

$\Leftrightarrow$ *[FDR]*

true

B.6 Discussion

In Section B.5.1 we defined a parallel operator on our communicating data types and in Theorem 5 we proved that this definition is consistent with the definition of the parallel operator on CSP processes: within the singleton failures semantic model the process corresponding to the parallel combination of two communicating data types is equivalent to the parallel combination of the processes corresponding to the two data types. In this section we ask: Is this consistency enough? What do we really mean by the parallel combination of two communicating data types? Are our data types simply *sharing* values in situations when the intention is that one data type is *passing* a chosen value to the other data type?

Consider Example 1 above. We put the data types *Client* and *Server* in parallel and in the operation *checkService* the client and the server shared the value *serv* identifying the requested service. Observe that the *checkService* operation on the parallel combination of the two data types is a symmetric operation whereas the intuition behind it is that the name of the chosen service is passed *from* the client component *to* the server component. The server component would then determine whether or not the client may provide the requested service.

In this particular example the symmetry did not matter: our definition of the *Client* and *Server* components ensured that the client—the component passing the chosen value—was willing only to communicate the value of *serv* that it had stored in its state space whereas the server was willing to communicate any value. Since the external choice over a singleton set is equivalent to the internal choice over the same singleton set, the effect was equivalent to the client passing the chosen value to the server.

And perhaps, if we define our data types sensibly, it will always be the case that the symmetry does not matter. Or indeed, perhaps the situation is asymmetrical after all: the values which the data type "passing" values is willing to communicate will be determined by its current state and the relation describing the operation on that data type. If this relation is suitably restrictive whereas the relation describing the corresponding operation on the other data type is not at all restrictive we will have achieved our desired asymmetry.

Furthermore, if such value-passing operations are internal between two sub-components that have been put in parallel and these internal operations are subsequently hidden, then whether the value is shared or passed is immaterial.

In the short term we will conclude that our current definition and interpretation of the parallel combination of communicating data types, whilst not intuitively elegant, is mathematically adequate. We will save the re-definition of our data types—expressing them in a form that, when considering their parallel combination, can *explicitly* handle the asymmetry of passing values between data types—for future work.