Under consideration for publication in Formal Aspects of Computing

Relational Concurrent Refinement Part III: Traces, partial relations and automata

John Derrick[1] and Eerke Boiten[2]

[1] Department of Computer Science, University of Sheffield, Sheffield, S1 4DP, UK
[2] School of Computing, University of Kent, Canterbury, Kent, CT2 7NF, UK

Abstract. Data refinement in a state-based language such as Z is defined using a relational model in terms of the behaviour of abstract programs. Downward and upward simulation conditions form a sound and jointly complete methodology to verify relational data refinements, which can be checked on an event-by-event basis rather than per trace. In models of concurrency, refinement is often defined in terms of sets of observations, which can include the events a system is prepared to accept or refuse, or depend on explicit properties of states and transitions. By embedding such concurrent semantics into a relational framework, eventwise verification methods for such refinement relations can be derived.
In this paper we continue our program of deriving simulation conditions for process algebraic refinement by defining further embeddings into our relational model: traces, completed traces, failure traces and extension. We then extend our framework to include various notions of automata based refinement.

Keywords: Data refinement, Z, simulations, automata-based refinements, concurrency, traces, completed traces, failures, failure traces, extension.

1. Introduction

The last 15 years have seen significant research effort in comparing notions of refinement in different models of specification and computation, particularly motivated by the desire to integrate specification languages that use different paradigms. In particular, we have considered the integration of state-based and concurrent specification methods, and the introduction of relational verification methods for refinement into a concurrency context.

Correspondence and offprint requests to: J. Derrick and E.A. Boiten

In a process algebra such as CSP [24] a system is defined in terms of actions (or events) which represent the *inter*actions between a system and its environment. The exact way in which the environment is allowed to interact with the system varies between different semantics. Typical semantics are set-based, associating one or more sets with each process, for example traces, refusals, divergences. Refinement is then defined in terms of set inclusions and equalities between the corresponding sets for different processes. A survey and taxonomy of many prominent process algebraic refinement relations is given in [38, 37].

In state-based systems, specifications are considered to define abstract data types (ADTs), consisting of an initialisation, a collection of operations and a finalisation, all of which are relations. A program over an ADT is a sequential composition of these elements, transforming a global visible state into another one via a sequence of hidden local states. Refinement is defined to be the subset relation over program behaviours, where what is deemed visible (i.e., the domain of the initialisation and the range of the finalisation) is the input/output relation. Thus a (concrete) ADT C refines a (more abstract) ADT A if for every program and sequence of inputs, the outputs that C produces are outputs that A could also have produced. As can be seen this definition of refinement quantifies over program behaviour and, to make verification of refinements tractable, *simulations* have become the accepted approach [15]. For a complete method, often two kinds of simulations are defined: downward and upward simulations. In the literature these are sometimes also called forward and backward simulations.

Research on combining relational and concurrent refinement concentrated initially on providing joint semantics, and on identifying correspondences between variations of the relational models and concurrency semantics. In the latter category, see e.g. work by Bolton and Davies [8, 9] and Reeves and Streader [32]. Our work on relational concurrent refinement started [3, 17] from the powerful idea that the relational *finalisations* can encode the observations embedded in concurrency semantics. The relational simulation rules can then be used to extract simulations for concurrency. These provide a "canned induction" method of verifying concurrent refinement, by checking a fixed number of conditions for each possible action, rather than checking inclusion between potentially large sets. We derived simulation rules for failures-divergences refinement [3, 17], including also outputs and internal operations [6], and for readiness refinement [17]. These were mostly based on the total relations model (as described below). In all these cases, the refinement notions have been imported from a concurrency context, represented in a relational formalism, and then expressed in terms of Z data types. Thus it provides for an integration of paradigms by allowing specification using Z schemas and sets while adapting a concurrency-style semantics.

This paper continues the programme, by considering more concurrent refinement relations, many of them based on the *partial* relations model. In Section 2 we provide the basic definitions and background. In Section 3 we provide the simulation rules for a number of process algebraic preorders. In Section 4 we introduce automata and IO automata, their refinement notions, and derive their relational simulation rules. In Section 5 we consider what extensions are necessary in order to include internal events and the possibility of divergence. We conclude in Section 6.

2. Background

This background section presents the standard refinement theory [16] for abstract data types in a relational setting. The relational model of data refinement where all operations are total, as described in the 1986 paper by He, Hoare and Sanders [23], traditionally received the most attention. The standard refinement theory of Z [39, 16], for example, is based on this version of the theory. However, later publications by He and Hoare, in particular [22], dropped the restriction to total relations, and proved soundness and joint completeness of the same set of simulation rules in the more general case. De Roever and Engelhardt [15] also present the partial relations theory, without emphasizing this.

2.1. A partial relational model

A program (defined here as a sequence of operations) is given as a relation over a global state G, implemented using a local state State. The *initialisation* of the program takes a global state to a local state, on which the operations act, a *finalisation* translates back from local to global. In order to distinguish between relational formulations (which use Z as a meta-language) and expressions in terms of Z schemas etc., we use the convention that expressions and identifiers in the world of relational data types are typeset in a sans serif font. The following defines our notion of abstract data type.

Definition 1 (Data type).
A (partial) *data type* is a quadruple $(\mathsf{State}, \mathsf{Init}, \{\mathsf{Op}_i\}_{i \in J}, \mathsf{Fin})$. The operations $\{\mathsf{Op}_i\}$, indexed by $i \in J$, are relations on the set State; Init is a total relation from G to State; Fin is a total relation from State to G. If the operations are all total relations, we call it a *total* data type.

An *i-data type* is a quintuple $(\mathsf{State}, \mathsf{Init}, \{\mathsf{Op}_i\}_{i \in J}, i, \mathsf{Fin})$ such that $(\mathsf{State}, \mathsf{Init}, \{\mathsf{Op}_i\}_{i \in J}, \mathsf{Fin})$ is a data type and i (the *internal operation*) is a partial relation on State. □

Insisting that Init and Fin be total merely records the facts that we can always start a program sequence (the extension to partial initialisations is trivial) and that we can always make an observation (but see [6] for a variant that uses a partial finalisation).

Definition 2 (Program).
For a data type $\mathsf{D} = (\mathsf{State}, \mathsf{Init}, \{\mathsf{Op}_i\}_{i \in J}, \mathsf{Fin})$ a *program* is a sequence over J. The meaning of a program p over D is denoted by $\mathsf{p_D}$, and defined as follows. If $\mathsf{p} = \langle \mathsf{p_1}, ..., \mathsf{p_n} \rangle$ then $\mathsf{p_D} = \mathsf{Init} \fatsemi \mathsf{Op_{p_1}} \fatsemi ... \fatsemi \mathsf{Op_{p_n}} \fatsemi \mathsf{Fin}$. □

Usually, we assume that the data types being compared for refinement are *conformal*, i.e., they use the same index set for the operations.

Definition 3 (Data refinement).
For data types A and C, C *refines* A, denoted $\mathsf{A} \sqsubseteq_{data} \mathsf{C}$ (dropping the subscript if the context is clear), iff for each program p over J, $\mathsf{p_C} \subseteq \mathsf{p_A}$. □

Data refinement for i-data types will not be defined independently, but in terms of data refinement depending on different interpretations of internal operations, see Section 5.

Downward and *upward* simulations form a sound and jointly complete [23, 15] proof method for verifying refinements. In a simulation a step-by-step comparison is made of each operation in the data types, and to do so the concrete and abstract states are related by a retrieve relation.

Definition 4 (Downward simulation).
Assume data types $\mathsf{A} = (\mathsf{AState}, \mathsf{AInit}, \{\mathsf{AOp}_i\}_{i \in J}, \mathsf{AFin})$ and $\mathsf{C} = (\mathsf{CState}, \mathsf{CInit}, \{\mathsf{COp}_i\}_{i \in J}, \mathsf{CFin})$. A *downward* simulation is a relation R from AState to CState satisfying

$$\mathsf{CInit} \subseteq \mathsf{AInit} \fatsemi \mathsf{R}$$
$$\mathsf{R} \fatsemi \mathsf{CFin} \subseteq \mathsf{AFin}$$
$$\forall\, i : J \bullet \mathsf{R} \fatsemi \mathsf{COp}_i \subseteq \mathsf{AOp}_i \fatsemi \mathsf{R}$$

□

Any relational data types A and C in this paper are assumed to be defined as in the above definition (occasionally with extra conditions imposed).

Definition 5 (Upward simulation).

For data types A and C, an *upward* simulation is a relation T from CState to AState such that

$$\mathsf{CInit} \fatsemi \mathsf{T} \subseteq \mathsf{AInit}$$
$$\mathsf{CFin} \subseteq \mathsf{T} \fatsemi \mathsf{AFin}$$
$$\forall\, i : J \bullet \mathsf{COp}_i \fatsemi \mathsf{T} \subseteq \mathsf{T} \fatsemi \mathsf{AOp}_i$$

□

2.2. Totalisations

The natural encoding of particular programmes being "impossible", e.g. leading to a deadlock, in the partial relational model is through the empty relation. However, a non-deterministic choice (union of relations) may then ensure that *possible* rather than *certain* erroneous behaviour is not observable at all – see [18] for a detailed discussion. Sticking with the core idea of relational concurrent refinement, this can be solved by observing *more* (e.g. refusals) at the end of a program, as we have done elsewhere. A more traditional approach is to encode error behaviour explicitly in operations. This is often called "totalisation", as it typically increases operations' domains to become total, but here and elsewhere we also apply it resulting in relations that remain partial.

There are two main types of totalisation: the *non-blocking* (or non-strict, or chaotic) totalisation represents erroneous behaviour as leading to all possible states including a new error state; the *blocking* (or strict) totalisation maps error traces only to a "sink" state. The totalisations turn a partial relation on a set S into a total relation on a set $\mathsf{S}_\perp$, which is S extended with a distinguished value $\perp$ not in S.

Definition 6 (Totalisation).
For a partial relation Op on State, its totalisation is a total relation on $\mathsf{State}_\perp$, defined in the non-blocking model by

$$\widehat{\mathsf{Op}}^{\mathsf{nb}} == \mathsf{Op} \cup \{\mathsf{x}, \mathsf{y} : \mathsf{State}_\perp \mid \mathsf{x} \notin \operatorname{dom} \mathsf{Op} \bullet (\mathsf{x}, \mathsf{y})\}$$

or in the blocking model by

$$\widehat{\mathsf{Op}}^{\mathsf{b}} == \mathsf{Op} \cup \{\mathsf{x} : \mathsf{State}_\perp \mid \mathsf{x} \notin \operatorname{dom} \mathsf{Op} \bullet (\mathsf{x}, \perp)\}$$

Totalisations of initialisation and finalisation are defined analogously.

A relation R between AState and CState is extended to a relation $\widetilde{\mathsf{R}}$ between $\mathsf{AState}_\perp$ and $\mathsf{CState}_\perp$, defined in the non-blocking model by $\widetilde{\mathsf{R}} == \mathsf{R} \cup (\{\perp_{\mathsf{AState}}\} \times \mathsf{CState}_\perp)$ and in the blocking model by $\widetilde{\mathsf{R}} == \mathsf{R} \cup \{(\perp_{\mathsf{AState}}, \perp_{\mathsf{CState}})\}$ respectively. □

Characterisations of downward and upward simulations on these totalised relations can be simplified to remove any reference to $\perp$. This results in the standard definitions of downward and upward simulations for partial relations, see [16], e.g.:

Definition 7 (Downward simulation for totalised relations).
Given data types A and C where the operations may be partial. A *downward* simulation is a relation R from AState to CState satisfying, in the non-blocking model

$$\mathsf{CInit} \subseteq \mathsf{AInit} \fatsemi \mathsf{R}$$
$$\forall\, i : J \bullet \operatorname{ran}(\operatorname{dom} \mathsf{AOp}_i \lhd \mathsf{R}) \subseteq \operatorname{dom} \mathsf{COp}_i$$
$$\forall\, i : J \bullet (\operatorname{dom} \mathsf{AOp}_i \lhd \mathsf{R}) \fatsemi \mathsf{COp}_i \subseteq \mathsf{AOp}_i \fatsemi \mathsf{R}$$

In the blocking model, the last condition is strengthened to:

$$\forall\, i : J \bullet \mathsf{R} \fatsemi \mathsf{COp}_i \subseteq \mathsf{AOp}_i \fatsemi \mathsf{R}$$

□

Definition 8 (Upward simulation for totalised relations).
For data types A and C where the operations may be partial, an *upward* simulation is a relation T from CState to AState satisfying, in the non-blocking model

$$\begin{array}{l}
\mathsf{CInit} \mathbin{⨾} \mathsf{T} \subseteq \mathsf{AInit} \\
\forall\, \mathsf{c} : \mathsf{CState} \bullet \exists\, \mathsf{a} : \mathsf{AState} \bullet (\mathsf{c}, \mathsf{a}) \in \mathsf{T} \\
\forall\, i : J \bullet \overline{\mathrm{dom}\, \mathsf{COp}_i} \subseteq \mathrm{dom}(\mathsf{T} \rhd \mathrm{dom}\, \mathsf{AOp}_i) \\
\forall\, i : J \bullet \mathrm{dom}(\mathsf{T} \rhd \mathrm{dom}\, \mathsf{AOp}_i) \lhd \mathsf{COp}_i \mathbin{⨾} \mathsf{T} \subseteq \mathsf{T} \mathbin{⨾} \mathsf{AOp}_i
\end{array}$$

In the blocking model the last condition is strengthened to:

$$\forall\, i : J \bullet \mathsf{COp}_i \mathbin{⨾} \mathsf{T} \subseteq \mathsf{T} \mathbin{⨾} \mathsf{AOp}_i$$

□

The conditions imposed on all operations in Definitions 7 and 8 are called "applicability" and "correctness" in both cases.

Although in this paper we mostly employ the partial relation model, we will need, on occasion, elements of the kind of totalisation we have just described in order to give a relational counterpart to some of the refinement preorders we look at below.

2.3. Refinement in Z

The definition of refinement in a specification language such as Z is usually based on the totalised framework just given. Specifically, a Z specification can be thought of as a data type, defined as a tuple $(State, Init, \{Op_i\}_{i \in J})$. The operations Op_i are defined in terms of (the variables of) $State$ (its before-state) and $State'$ (its after-state). The initialisation is also expressed in terms of an after-state $State'$. In addition to this, operations can also consume inputs and produce outputs. As finalisation is implicit in these data types, it only has an occasional impact on specific refinement notions. If specifications have inputs and outputs, these are included in both the global and local state of the relational embedding of a Z specification. See [16] for the full details on this – in this paper we only consider data types without inputs and outputs. In concurrent refinement relations, inputs add little complication; outputs particularly complicate refusals, as described in [6].

In a context where there is no input or output, the global state contains no information and is a one point domain, i.e., $\mathsf{G} == \{*\}$, and the local state is $\mathsf{State} == State$. In such a context the other components of the embedding are as given below.

Definition 9 (Basic embedding of Z data types). The Z data type $(State, Init, \{Op_i\}_{i \in J})$ is interpreted relationally as $(\mathsf{State}, \mathsf{Init}, \{\mathsf{Op}_i\}_{i \in J}, \mathsf{Fin})$ where

$$\begin{array}{l}
\mathsf{Init} == \{Init \bullet * \mapsto \theta State'\} \\
\mathsf{Op} == \{Op \bullet \theta State \mapsto \theta State'\} \\
\mathsf{Fin} == \{State \bullet \theta State \mapsto *\}
\end{array}$$

Given these embeddings, we can translate the relational refinement conditions of downward simulations for totalised relations into refinement conditions for Z ADTs, where we note that the finalisation conditions are always satisfied in this Z interpretation.

Definition 10 (Standard downward simulation in Z).
Given Z data types $A = (AState, AInit, \{AOp_i\}_{i \in J})$ and $C = (CState, CInit, \{COp_i\}_{i \in J})$. The relation R

on $AState \wedge CState$ is a *downward simulation* from A to C in the non-blocking model if

$$\begin{array}{l} \forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \wedge R' \\ \forall\, i : J;\ AState;\ CState \bullet \mathrm{pre}\, AOp_i \wedge R \Rightarrow \mathrm{pre}\, COp_i \\ \forall\, i : J;\ AState;\ CState;\ CState' \bullet \mathrm{pre}\, AOp_i \wedge R \wedge COp_i \\ \qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i \end{array}$$

In the blocking model, the correctness (last) condition becomes

$$\forall\, i : J;\ AState;\ CState;\ CState' \bullet R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i$$

and then the applicability (second) condition above is equivalent to

$$\forall\, i : J;\ AState;\ CState \bullet R \Rightarrow (\mathrm{pre}\, AOp_i \Leftrightarrow \mathrm{pre}\, COp_i)$$ □

Any Z data types A and C in this paper are assumed to be defined as in the above definition.

The translation of the upward simulation conditions is similar, however this time the finalisation condition produces a requirement that the simulation is total on the concrete state.

Definition 11 (Standard upward simulation in Z).
For Z data types A and C, the relation T on $AState \wedge CState$ is an *upward simulation* from A to C in the non-blocking model if

$$\begin{array}{l} \forall\, AState';\ CState' \bullet CInit \wedge T' \Rightarrow AInit \\ \forall\, i : J;\ CState \bullet \exists\, AState \bullet T \wedge (\mathrm{pre}\, AOp_i \Rightarrow \mathrm{pre}\, COp_i) \\ \forall\, i : J;\ AState';\ CState;\ CState' \bullet \\ \qquad\qquad (COp_i \wedge T') \Rightarrow (\exists\, AState \bullet T \wedge (\mathrm{pre}\, AOp_i \Rightarrow AOp_i)) \end{array}$$

In the blocking model, the correctness condition becomes

$$\forall\, i : J;\ AState';\ CState;\ CState' \bullet (COp_i \wedge T') \Rightarrow \exists\, AState \bullet T \wedge AOp_i$$ □

3. Process algebraic based refinement

Process algebras [24, 31, 2] provide a means of describing and verifying concurrent systems and processes, and provide operators such as synchronisation, communication, and various flavours of composition. The semantics of a process algebra is often given by a labelled transition system (LTS). For example, for CSP or CCS the language is modelled as a LTS where the state space is the set of terms in the language. Equivalence, and preorders, can be defined over the semantics where two terms are identified whenever no observer can notice any difference between their external behaviours. Thus equivalences and preorders can be defined in terms of a function O that represents the *set* of observations one could make while interacting with a process. For every such O we can define $p \sqsubseteq_O q$ iff $O(q) \subseteq O(p)$ and $p =_O q$ iff $O(p) = O(q)$. Varying how the environment *interacts* with a process leads to differing observations and these can be thought of as differing *testing scenarios*, and therefore different preorders (i.e., refinement relations) – an overview and comprehensive treatment is provided by van Glabbeek in [38, 37]. For systems without internal evolution, the relationship between different semantics is given by the linear-time, branching-time spectrum given in Figure 1.

The *testing scenarios* described in [38] refer to an informal description of an experiment upon the process and its behaviour upon testing. To do so a process is thought of as a black box that contains an interface to the environment, via which tests are performed - which consist of stimulating the interface, e.g., 'pressing the button labelled a' - and observing the outcome. Varying the interface gives different testing scenarios, a full characterisation is given in [37], for example, the interface might contain a display in which the name of the action is shown that is currently carried out by the process, buttons might also be present (one for each action) so that the observer may determine which actions are free and which are blocked, or lamps which illuminate if the process is ready to engage in that action.

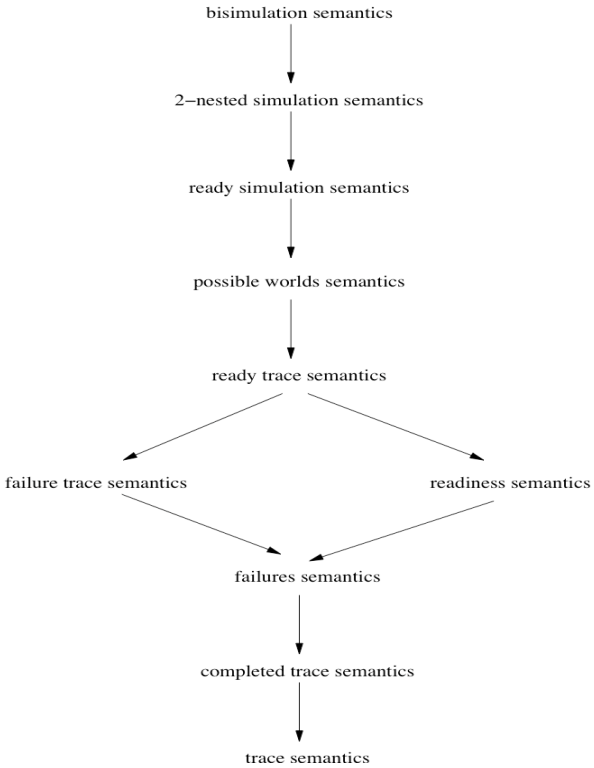

Fig. 1. The linear time - branching time spectrum [38]

We will need the usual notation for labelled transition systems (LTSs):

Definition 12 (Labelled Transition Systems (LTSs)).
A labelled transition system is a tuple $L = (States, Act, T, Init)$ where $States$ is a non-empty set of states, $Init \subseteq States$ is the set of initial states, Act is a set of actions, and $T \subseteq States \times Act \times States$ is a transition relation. The components of L are also accessed as $states(L) = States$ and $init(L) = Init$. □

Every state in the LTS represents a process itself – namely the one representing all possible behaviour from that point onwards. Specific notation needed includes the usual notation for writing transitions as $p \xrightarrow{a} q$ for $(p, a, q) \in T$ and the extension of this to traces (written $p \xrightarrow{tr} q$) and the set of enabled actions of a process which is defined as:

$next(p) = \{a \in Act \mid \exists\, q \bullet p \xrightarrow{a} q\}$.

In the remainder of this section we detail differing preorders and show how they are embedded into our relational model. For each we give its definition, its characterisation as a testing scenario as described by van Glabbeek, its embedding into a relational model, and thereby the definition of simulation rules to characterise the preorder.

3.1. Methodology

We have seen that Z specifications define data types, over which a definition of refinement is given. Simulations are then a tractable way of verifying such simulations due to results in [23, 15]. Alternatively, refinement in

a process algebra can be characterised in terms of the attributes of LTSs as we have just seen. In order to relate the two approaches we do the following:

1. define a relational embedding of the Z data type, that is, define a data type (specifically define the finalisation operation) so as to facilitate the proof that data refinement equals the event based semantics. The choice of finalisation is taken so that we observe the characteristics of interest. Thus in the context of trace refinement we are interested in observing traces, but in that of failures refinement we need to observe more.
2. We then describe how to calculate the relevant LTS aspect from the Z data type. For example, for trace refinement what denotes traces in the Z data type.
3. We then prove that data refinement equals the relevant event based definition of refinement.
4. Finally, we extract a characterisation of refinement as simulation rules on the operations of the Z data type.

The points of originality for each section are the proofs of equivalence of refinement followed by the characterisation of simulation rules that this allows us to give. These simulations are due to their construction guaranteed to provide a sound proof method for the given notion of refinement; however, their joint completeness requires a separate proof – either an independent one, or one that shows that a standard construction of an intermediate data type in a completeness proof occurs within the range of the given embedding. See [5] for a detailed discussion and an example.

The following subsection will give a flavour for the approach used throughout the paper.

3.2. Trace preorder

3.2.1. Definition and testing scenario

Definition 13. $\sigma \in Act^*$ is a trace of a process p if $\exists\, q \bullet p \xrightarrow{\sigma} q$. $\mathcal{T}(p)$ denotes the set of traces of p. The trace preorder is defined by $p \sqsubseteq_{tr} q$ iff $\mathcal{T}(q) \subseteq \mathcal{T}(p)$. □

Testing scenario: Observations consist of a sequence of actions performed by the process in succession, that is, the interface is just a display which shows the name of the action that is currently carried out by the process, and the name remains visible in the display if deadlock occurs (unless deadlock occurs initially).

3.2.2. Relational embedding

As observed previously [17] the partial relations model records exactly trace information for the embedding with trivial finalisation described in Section 2.3. Possible traces lead to the single global value; impossible traces have no relational image.

Definition 14 (Trace embedding).
A Z data type $(State, Init, \{Op_i\}_{i \in J})$ has the following trace embedding into the relational model.

$$\begin{aligned}
&\mathsf{G} == \{*\}\\
&\mathsf{State} == State\\
&\mathsf{Init} == \{Init \bullet * \mapsto \theta State'\}\\
&\mathsf{Op} == \{Op \bullet \theta State \mapsto \theta State'\}\\
&\mathsf{Fin} == \mathsf{State} \times \mathsf{G}
\end{aligned}$$

To distinguish between the different embeddings we denote the trace embedding of a data type A as $\mathsf{A}\,|_{tr}$. We drop the $|_{tr}$ if the context is clear. □

To prove the correspondence between trace preorder and data refinement we need to provide a definition of the traces (as in Definition 13) of an abstract data type.

Definition 15. The *traces* of a Z data type $(State, Init, \{Op_i\}_{i \in J})$ are all sequences $\langle i_1, \ldots, i_n \rangle$ such that

$$\exists\, State' \bullet Init \fatsemi Op_{i_1} \fatsemi \ldots \fatsemi Op_{i_n}$$

We denote the traces of an ADT A by $\mathcal{T}(A)$. □

Theorem 1. With the trace embedding, data refinement corresponds to trace preorder. That is, when Z data types A and C are embedded as A and C[1],

$$\mathsf{A}\,\big|_{tr} \sqsubseteq_{data} \mathsf{C}\,\big|_{tr} \quad \text{iff } \mathcal{T}(C) \subseteq \mathcal{T}(A)$$

Proof From the definition of traces for Z data types and the embedding given it is obvious that for any sequence p, $(*, *) \in p_{\mathsf{A}}$ iff $p \in \mathcal{T}(A)$. Also, for any p, $p_{\mathsf{A}} = \{(*, *)\}$ or $p_{\mathsf{A}} = \varnothing$. Thus, data refinement ($p_{\mathsf{A}} \subseteq p_{\mathsf{C}}$ for all p) corresponds to trace refinement. □

From this result it can be seen that observations in the testing scenario, here a display with an action name displayed, are distributed in the relational notion of refinement. That is, although finalisations are often taken to be the 'observations', in fact, some of the observations are implicit in the program p and the relational inclusion $p_{\mathsf{C}} \subseteq p_{\mathsf{A}}$ (since finalisations only contain the information as to whether the trace was defined or not).

We can now extract the simulation rules that correspond to this notion of refinement. These are of course the rules for standard Z refinement but omitting applicability of operations, as used also e.g. in Event-B [1].

3.2.3. Simulations

The conditions for a downward simulation in the partial relational model are (cf. Definition 4):

$$\mathsf{CInit} \subseteq \mathsf{AInit} \fatsemi \mathsf{R}$$
$$\mathsf{R} \fatsemi \mathsf{CFin} \subseteq \mathsf{AFin}$$
$$\forall\, i : I \bullet \mathsf{R} \fatsemi \mathsf{COp}_i \subseteq \mathsf{AOp}_i \fatsemi \mathsf{R}$$

The first and last of these are just the standard initialisation and correctness conditions, respectively. The finalisation condition in fact places no further requirements with the trace embedding. The same is true for upwards simulations. We thus have the following conditions for the trace embedding.

Definition 16 (Trace simulations in Z).
Given Z data types A and C, the relation R on $AState \wedge CState$ is a *trace downward simulation* from A to C if

$$\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \wedge R'$$
$$\forall\, i \in J \bullet \forall\, AState;\ CState;\ CState' \bullet R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i$$

The total relation T on $AState \wedge CState$ is a *trace upward simulation* from A to C if

$$\forall\, AState';\ CState' \bullet CInit \wedge T' \Rightarrow AInit$$
$$\forall\, i : J \bullet \forall\, AState';\ CState;\ CState' \bullet$$
$$(COp_i \wedge T') \Rightarrow (\exists\, AState \bullet T \wedge AOp_i)$$

□

[1] This association between Z and relational data types is left implicit in the rest of this paper.

3.3. Completed trace preorder

3.3.1. Definition and testing scenario

Definition 17. $\sigma \in Act^*$ is a completed trace of a process p if $\exists\, q \bullet p \xrightarrow{\sigma} q$ and $next(q) = \varnothing$. $\mathcal{CT}(p)$ denotes the set of completed traces of p. The completed trace preorder, $\sqsubseteq_{ctr}$, is defined by $p \sqsubseteq_{ctr} q$ iff $\mathcal{T}(q) \subseteq \mathcal{T}(p)$ and $\mathcal{CT}(q) \subseteq \mathcal{CT}(p)$. □

Testing scenario: Observations consist of a sequence of actions performed by the process in succession, that is, the interface is just a display which shows the name of the action that is currently carried out by the process, where the display becomes empty if deadlock occurs.

3.3.2. Relational embedding

Definition 18 (Completed trace embedding).
The Z data type $(State, Init, \{Op_i\}_{i \in J})$ has the following completed trace embedding into the relational model.

$$\begin{aligned}
&\mathsf{G} == \{*, \surd\} \\
&\mathsf{State} == State \\
&\mathsf{Init} == \mathsf{G} \times \{Init \bullet \theta State'\} \\
&\mathsf{Op} == \{Op \bullet \theta State \mapsto \theta State'\} \\
&\mathsf{Fin} == \{State \bullet \theta State \mapsto *\} \cup \{State \mid (\forall\, i : J \bullet \neg \operatorname{pre} Op_i) \bullet \theta State \mapsto \surd\}
\end{aligned}$$

This embedding is denoted $\mathsf{A}\,\big|_{ctr}$. □

Here the global state has been augmented with an additional element $\surd$, which denotes that the given trace is complete (i.e., no operation is applicable).

Definition 19. The *completed traces* of a Z data type $(State, Init, \{Op_i\}_{i \in J})$ are all sequences $\langle i_1, \ldots, i_n \rangle$ such that

$$\exists\, State' \bullet Init \fatsemi Op_{i_1} \fatsemi \ldots \fatsemi Op_{i_n} \wedge \forall\, i : J \bullet \neg(\operatorname{pre} Op_i)'$$

We denote the complete traces of an ADT A by $\mathcal{CT}(A)$. □

Theorem 2. With the completed trace embedding, data refinement corresponds to completed trace preorder. That is,

$$\mathsf{A}\,\big|_{ctr} \sqsubseteq \mathsf{C}\,\big|_{ctr} \text{ iff } \mathcal{CT}(C) \subseteq \mathcal{CT}(A) \text{ and } \mathcal{T}(C) \subseteq \mathcal{T}(A)$$

Proof 1. Suppose that $\mathcal{CT}(C) \subseteq \mathcal{CT}(A)$ and $\mathcal{T}(C) \subseteq \mathcal{T}(A)$. To show $A \sqsubseteq C$ we need $p_{\mathsf{C}} \subseteq p_{\mathsf{A}}$ for all programs p. Given p, if p is not a trace of C then $p_{\mathsf{C}} = \varnothing$, and thus the inclusion is trivial. Otherwise, either $(*, \surd)$ and $(*, *)$ are both in p_{C} or just $(*, *)$ is in p_{C}.

If $(*, \surd)$ is in p_{C} then p is a completed trace in C, and thus also in A. Hence $(*, \surd)$ is in p_{A}, and so is $(*, *)$. If just $(*, *)$ is in p_{C} then p is a trace which is not a completed trace in C. Since $\mathcal{T}(C) \subseteq \mathcal{T}(A)$, p is also a trace in A. Hence $(*, *)$ is in p_{A}.

2. Suppose $A \sqsubseteq C$.

Given $p \in \mathcal{CT}(C)$. Thus $(*, \surd) \in p_{\mathsf{C}} \subseteq p_{\mathsf{A}}$, and hence $p \in \mathcal{CT}(A)$. For a similar reason we also get trace inclusion. □

We now extract the simulation rules that correspond to this notion of refinement.

3.3.3. Simulations

Given the completed trace embedding in the relational model, only the finalisation is non-trivially altered from the embedding given in Section 3.2. Thus we just have to consider the effect of the finalisation requirement:

Downward simulations: $\mathsf{R} \fatsemi \mathsf{CFin} \subseteq \mathsf{AFin}$ is equivalent to

$$\forall\, AState;\ CState \bullet R \wedge (\forall\, i : J \bullet \neg \operatorname{pre} COp_i) \Rightarrow \forall\, i : J \bullet \neg \operatorname{pre} AOp_i$$

Upward simulations: $\mathsf{CFin} \subseteq \mathsf{T} \fatsemi \mathsf{AFin}$ is equivalent to

$$\forall\, CState \bullet (\forall\, i : J \bullet \neg \operatorname{pre} COp_i) \Rightarrow \exists\, AState \bullet T \wedge \forall\, i : J \bullet \neg \operatorname{pre} AOp_i$$

We thus have the following conditions for the completed trace embedding.

Definition 20 (Completed trace simulations in Z).
Given Z data types A and C. The relation R on $AState \wedge CState$ is a *completed trace downward simulation* from A to C if

$$\begin{array}{l}
\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \wedge R' \\
\forall\, i \in J \bullet \forall\, AState;\ CState;\ CState' \bullet R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i \\
\forall\, AState;\ CState \bullet R \wedge (\forall\, i : J \bullet \neg \operatorname{pre} COp_i) \Rightarrow \forall\, i : J \bullet \neg \operatorname{pre} AOp_i
\end{array}$$

The total relation T on $AState \wedge CState$ is a *completed trace upward simulation* from A to C if

$$\begin{array}{l}
\forall\, AState';\ CState' \bullet CInit \wedge T' \Rightarrow AInit \\
\forall\, i \in J \bullet \forall\, AState';\ CState;\ CState' \bullet \\
\qquad\qquad (COp_i \wedge T') \Rightarrow (\exists\, AState \bullet T \wedge AOp_i) \\
\forall\, CState \bullet (\forall\, i : J \bullet \neg \operatorname{pre} COp_i) \Rightarrow \exists\, AState \bullet T \wedge \forall\, i : J \bullet \neg \operatorname{pre} AOp_i
\end{array}$$

□

3.4. Failure preorder

3.4.1. Definition and testing scenario

The failures semantics records both the traces that a process can do, and also sets of actions which it can refuse, that is, actions which are not enabled. These are recorded as failures of a process.

Definition 21. $(\sigma, X) \in Act^* \times \mathbb{P}(Act)$ is a failure of a process p if there is a process q such that $p \xrightarrow{\sigma} q$, and $next(q) \cap X = \varnothing$. $\mathcal{F}(p)$ denotes the set of failures of p. The failures preorder, $\sqsubseteq_f$, is defined by $p \sqsubseteq_f q$ iff $\mathcal{F}(q) \subseteq \mathcal{F}(p)$. □

Testing scenario: The machine for testing failures has, in addition to the interface of the completed trace machine, a switch for each action in Act. One can then observe which actions are blocked. If the process reaches a state where all actions are blocked, then this can be observed by an empty display. Observations are thus the failures of a process.

3.4.2. Relational embedding

This was covered in detail in [3, 17, 6], although we used an embedding into the totalised relational model there. Lemma 3 in [6] suggested this was not necessary: $\perp$ appears as a possible outcome iff somewhere along the trace the next action of the trace could be refused. Thus, below we give a simpler embedding into the partial relations model.

Definition 22 (Failures embedding).
A Z data type $(State, Init, \{Op_i\}_{i \in J})$ in the refusals interpretation is embedded in the relational model as follows.

$$\begin{aligned}
&\mathsf{G} == \mathbb{P}\, J \\
&\mathsf{State} == State \\
&\mathsf{Init} == \{Init;\ E : \mathbb{P}\, J \bullet E \mapsto \theta State'\} \\
&\mathsf{Op} == \{Op \bullet \theta State \mapsto \theta State'\} \\
&\mathsf{Fin} == \{State;\ E : \mathbb{P}\, J \mid (\forall\, i \in E \bullet \neg \operatorname{pre} Op_i) \bullet \theta State \mapsto E\}
\end{aligned}$$

This embedding is denoted $\mathsf{A}\,\big|_f$. □

In the relational embedding failures are pairs (tr, X), where tr is a trace, and there exists states $(\mathsf{State}, \mathsf{State}') \in tr$ (with State being initial) such that $\forall\, i : X \bullet \mathsf{State}' \notin \operatorname{dom} \mathsf{Op}_i$.

Theorem 3. With the failures embedding, data refinement corresponds to the failures preorder. That is,

$$\mathsf{A}\,\big|_f \sqsubseteq \mathsf{C}\,\big|_f \ \text{ iff } \mathcal{F}(C) \subseteq \mathcal{F}(A)$$

The proof of this is an adaptation of that given in [17]. □

3.4.3. Simulations

Given the failures embedding the changes to the simulation conditions are as follows (these are derived in [17] - remember we have no input/output at this stage):

Downward simulations: $\mathsf{R} \fatsemi \mathsf{CFin} \subseteq \mathsf{AFin}$ is equivalent to

$$\forall\, i : J;\ AState;\ CState \bullet R \wedge \operatorname{pre} AOp_i \Rightarrow \operatorname{pre} COp_i$$

Upward simulations: $\mathsf{CFin} \subseteq \mathsf{T} \fatsemi \mathsf{AFin}$ is equivalent to

$$\forall\, CState \bullet \exists\, AState \bullet \forall\, i : J \bullet T \wedge (\operatorname{pre} AOp_i \Rightarrow \operatorname{pre} COp_i)$$

3.5. Failure trace preorder

3.5.1. Definition and testing scenario

The failure trace semantics considers refusal sets not only at the end of a trace, but also between each action in a trace.

Definition 23. $\sigma \in (Act \cup \mathbb{P}\, Act)^*$ is a failure trace of a process p if $\sigma = X_1 a_1 X_2 a_2 \ldots X_n a_n X_{n+1}$ where $a_1 a_2 \ldots a_n$ is a trace of p and each $(a_1 \ldots a_i, X_{i+1})$ is a failure of p. $\mathcal{FT}(p)$ denotes the set of failure traces of p. The failures traces preorder, $\sqsubseteq_{ftr}$, is defined by $p \sqsubseteq_{ftr} q$ iff $\mathcal{FT}(q) \subseteq \mathcal{FT}(p)$. □

Testing scenario: The display in the machine for testing failures traces is the same as that for failures. However, it does not halt if the process cannot proceed, rather it idles until the observer allows one of the actions the process is ready to perform. The observations are traces with idle periods in between, and for each idle period the set of actions that are not blocked by the observer.

It has been argued [26, 27] that this is a better notion for testing than simply observing failures of a process, and is appropriate when one can detect that a process refuses an action, and if this is the case, one has the ability to try another action.

3.5.2. Relational embedding

Definition 24 (Failure trace embedding).
A Z data type $(State, Init, \{Op_i\}_{i \in J})$ in the failure trace interpretation is embedded in the relational model in an obvious generalisation of the failures embedding. The observation of refusals at finalisation is retained, but a similar observation is also made before every operation; these observations are collected in a sequence, which is initialised as empty and copied to the global state at finalisation (similar to the standard treatment of outputs in Z [39, 16]).

$$\begin{aligned}
&\mathsf{G} == \mathrm{seq}\,\mathbb{P}\,J \\
&\mathsf{State} == \mathrm{seq}\,\mathbb{P}\,J \times State \\
&\mathsf{Init} == \mathsf{G} \times \{Init \bullet (\langle\,\rangle, \theta State')\} \\
&\mathsf{Op} == \{Op;\ fs : \mathrm{seq}\,\mathbb{P}\,J;\ E : \mathbb{P}\,J \mid (\forall\, i : E \bullet \neg\,\mathrm{pre}\,Op_i) \bullet \\
&\qquad (fs, \theta State) \mapsto (fs \frown \langle E \rangle, \theta State')\} \\
&\mathsf{Fin} == \{State;\ fs : \mathrm{seq}\,\mathbb{P}\,J;\ E : \mathbb{P}\,J \mid (\forall\, i : E \bullet \neg\,\mathrm{pre}\,Op_i) \bullet (fs, \theta State) \mapsto fs \frown \langle E \rangle\}
\end{aligned}$$

The embedding is denoted $\mathsf{A}\,|_{ft}$. □

In the relational embedding failures traces are the obvious generalisation of failures.

Theorem 4. With the failure traces embedding, data refinement corresponds to the failure traces preorder. That is,

$$\mathsf{A}\,|_{ft} \sqsubseteq \mathsf{C}\,|_{ft} \ \text{ iff } \mathcal{FT}(C) \subseteq \mathcal{FT}(A)$$

□

3.5.3. Simulations

In the failure trace embedding, both the correctness and finalisation conditions are potentially amended due to the record of failures at each operation step; initialisation conditions are unchanged from the trace simulations. The extended retrieve relation will necessarily relate only identical sequences of previous observations. The derivations of simulation conditions are very similar to those for failures refinement, as given in great detail in [3], except that we use the partial relations model here where there is only a single simulation condition ("correctness") for individual operations.

Downward simulations: The finalisation condition here leads to the traditional "applicability" condition for operations, namely

$$\forall\, AState;\ CState;\ i : J \bullet (R \wedge \mathrm{pre}\,AOp_i) \Rightarrow \mathrm{pre}\,COp_i$$

The correctness condition $\mathsf{R} \fatsemi \mathsf{COp}_i \subseteq \mathsf{AOp}_i \fatsemi \mathsf{R}$ expands to the following

$$\begin{aligned}
&\forall\, i : J;\ AState;\ CState;\ CState' \bullet \forall\, E \bullet \\
&\quad R \wedge COp_i \wedge Fcond(E, \theta CState) \ \Rightarrow \\
&\quad \exists\, AState' \bullet R' \wedge AOp_i \wedge Fcond(E, \theta AState)
\end{aligned}$$

where $Fcond(E, s) == \forall\, i : E \bullet \neg \exists\, Op_i \bullet s = \theta State$. However, taking the finalisation condition into account, this simplifies to the standard correctness condition:

$$\forall\, i : J;\ AState;\ CState;\ CState' \bullet \\ R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i$$

Upward simulations: The finalisation condition here expands to the stronger "applicability" condition also known from failures refinement, see [3, Appendix A] for the least obvious step in its derivation:

$$\forall\, CState \bullet \exists\, AState \bullet T \wedge Ref(CState) \subseteq Ref(AState)$$

where Ref denotes a maximal refused set of operations, i.e., $Ref(State) = \{i : J \mid \neg \operatorname{pre} Op_i\}$. The correctness condition for operations $\mathsf{COp}_i \fatsemi \mathsf{T} \subseteq \mathsf{T} \fatsemi \mathsf{AOp}_i$ expands to

$$\forall\, i : J;\ AState';\ CState;\ CState' \bullet \\ (COp_i \wedge T') \Rightarrow \exists\, AState \bullet T \wedge AOp_i \wedge Ref(CState) \subseteq Ref(AState)$$

which does not imply the finalisation condition (e.g. there may be states with $Ref(CState) = J$).

3.6. Extension and conformance

3.6.1. Definition and testing scenario

Here we consider a number of alternative preorders for process algebras which are not considered in [38]. These have been suggested motivated by testing and test generation from LOTOS specifications [7], specifically *extension* and *conformance* [11]. To define these formally we need the following notation which defines refusals sets after a particular trace (i.e., a failure, cf. earlier definition).

Definition 25 (Refusals after a trace).
Let p be a LTS, σ a trace of p, and $X \subseteq Act$. Then p **after** σ **ref** X iff

$$\exists\, q \bullet p \xrightarrow{\sigma} q \text{ and } X \cap next(q) = \varnothing$$

□

Testing scenario: Three definitions of refinement can be given on the basis of the idea behind Definition 25. These were motivated in [11, 12] by considering that there might be a number of different notions of implementation:

- implementation as a real/physical system;
- implementation as a (deterministic) reduction of a given specification;
- implementation as a (conforming) extension of a given specification;
- implementation as a refinement of a given specification.

These are formalised [10] by, respectively, conformance, reduction, extension and testing equivalence. Reduction (also called the testing preorder [13]) in our context (of no divergence) is identical to the failures preorder. Testing equivalence is the equivalence induced by that preorder.

Conformance has the following characteristics: if $p \sqsubseteq_{conf} q$ then q deadlocks less often than p in any environment whose traces are limited to those of p. Thus conformance restricts the quantification (of traces one must check refusals about) to be over the abstract specification (and this restriction gives rise to efficient test generation algorithms).

The *extension* preorder can be defined as conformance together with the additional property that traces can be extended. Thus, if $p \sqsubseteq_{ext} q$ then q has at least the same traces as p, but in an environment whose traces

are limited to those of p, it deadlocks less often. The equivalence induced by extension is the same as that by reduction (that is, testing equivalence). Leduc [28] documents the relationship between these relations in some detail. They can be defined as follows.

Definition 26 (Reduction, conformance, and extension).
Let p, q be LTSs. Then

$$p \sqsubseteq_{red} q \text{ iff } \forall \sigma : Act^*;\ X \subseteq Act \bullet q \textbf{ after } \sigma \textbf{ ref } X \text{ implies } p \textbf{ after } \sigma \textbf{ ref } X$$

$$p \sqsubseteq_{conf} q \text{ iff } \forall \sigma : \mathcal{T}(p);\ X \subseteq Act \bullet q \textbf{ after } \sigma \textbf{ ref } X \text{ implies } p \textbf{ after } \sigma \textbf{ ref } X$$

$$\begin{aligned} & p \sqsubseteq_{ext} q \text{ iff} \\ & \quad \mathcal{T}(p) \subseteq \mathcal{T}(q) \text{ and} \\ & \quad \forall \sigma : \mathcal{T}(p);\ X \subseteq Act \bullet q \textbf{ after } \sigma \textbf{ ref } X \text{ implies } p \textbf{ after } \sigma \textbf{ ref } X \end{aligned}$$

□

3.6.2. Relational embedding

The relational embedding we use to model extension is, in fact, a totalisation over the space of partial relations, and is the standard *non-blocking* model (e.g., as discussed in [16]).

Definition 27 (Extension embedding).
A Z data type $(State, Init, \{Op_i\}_{i \in J})$ in the extension interpretation is embedded in the relational model as follows.

$$\begin{aligned} & \mathsf{G} == \mathbb{P}\, J \cup \{\bot\} \\ & \mathsf{State} == State \cup \{\bot\} \\ & \mathsf{Init} == \mathsf{G} \times \{Init \bullet \theta State'\} \\ & \mathsf{Op} == \mathsf{OpB} \cup \{\mathsf{x}, \mathsf{y} : \mathsf{State} \mid \mathsf{x} \notin \operatorname{dom} \mathsf{OpB} \bullet (\mathsf{x}, \mathsf{y})\} \\ & \text{where } \mathsf{OpB} == \{Op \bullet \theta State \mapsto \theta State'\} \\ & \mathsf{Fin} == \{State;\ E : \mathbb{P}\, J \mid (\forall\, i : E \bullet \neg \operatorname{pre} Op_i) \bullet \theta State \mapsto E\} \cup \{\bot\} \times \mathsf{G} \end{aligned}$$

This embedding is denoted $\mathsf{A} \mid_{ext}$. □

Theorem 5. With the extension embedding, data refinement corresponds to the extension preorder. That is,

$$\begin{aligned} & \mathsf{A} \mid_{ext} \sqsubseteq \mathsf{C} \mid_{ext} \text{ iff} \\ & \quad \mathcal{T}(A) \subseteq \mathcal{T}(C) \text{ and} \\ & \quad \forall \sigma \in \mathcal{T}(A);\ X \subseteq Act \bullet C \textbf{ after } \sigma \textbf{ ref } X \text{ implies } A \textbf{ after } \sigma \textbf{ ref } X \end{aligned}$$

Proof 1. Suppose that $\mathcal{T}(A) \subseteq \mathcal{T}(C)$ and
$\forall \sigma \in \mathcal{T}(A);\ X \subseteq Act \bullet C$ **after** σ **ref** X implies A **after** σ **ref** X.

Consider a trace p. If $p \notin \mathcal{T}(A)$ then the embedding (through the totalisation) ensures that $p_{\mathsf{A}} = \mathsf{G} \times \mathsf{G}$ and thus $p_{\mathsf{C}} \subseteq p_{\mathsf{A}}$ trivially holds. If $p \in \mathcal{T}(A)$ then by assumption also $p \in \mathcal{T}(C)$. Now although p is a trace in C, it is still possible that $p_{\mathsf{C}} = \mathsf{G} \times \mathsf{G}$. In this case, p is "blocked along the way" in C, i.e., there is an action a and strings p' and p'' such that $p = p'ap''$ and C **after** p' **ref** $\{a\}$. Then refusal inclusion ensures that A **after** p' **ref** $\{a\}$ and the embedding ensures that $p_{\mathsf{A}} = \mathsf{G} \times \mathsf{G}$, thus $p_{\mathsf{C}} \subseteq p_{\mathsf{A}}$. When $p_{\mathsf{C}} \neq \mathsf{G} \times \mathsf{G}$, all observations made record genuine refusals, and $(g, X) \in p_{\mathsf{C}}$ implies C **after** p **ref** X, by assumption then also A **after** p **ref** X and thus $(g, X) \in p_{\mathsf{A}}$ and $p_{\mathsf{C}} \subseteq p_{\mathsf{A}}$.

2. Suppose $A \sqsubseteq C$. Then trace inclusion can be proved by induction over the length of the trace, and refusals subsetting follows as a consequence of using the non-blocking totalisation. □

Whilst we have found an embedding such that data refinement induces extension, this is not possible for conformance. This is because conformance is *not* a preorder (see any of the references given above), but data refinement *is* a preorder. Thus no combinations of embeddings as a data refinement theory will produce an embedding equivalent to it.

3.6.3. Simulations

The use of the non-blocking totalisation for modelling extension means we can extract simulation conditions by reference to above results. They are thus the following.

Definition 28 (Extension downward simulation in Z).
Given Z data types A and C. The relation R on $AState \wedge CState$ is a *extension downward simulation* from A to C if

$$\begin{array}{l} \forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \wedge R' \\ \forall\, i : J;\ AState;\ CState \bullet \operatorname{pre} AOp_i \wedge R \Rightarrow \operatorname{pre} COp_i \\ \forall\, i : J;\ AState;\ CState;\ CState' \bullet \operatorname{pre} AOp_i \wedge R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i \end{array}$$

□

Definition 29 (Extension upward simulation in Z).
Given Z data types A C. The total relation T on $AState \wedge CState$ is an *extension upward simulation* from A to C if

$$\begin{array}{l} \forall\, AState';\ CState' \bullet CInit \wedge T' \Rightarrow AInit \\ \forall\, CState \bullet \exists\, AState \bullet \forall\, i : J \bullet T \wedge (\operatorname{pre} AOp_i \Rightarrow \operatorname{pre} COp_i) \\ \forall\, i : J;\ AState';\ CState;\ CState' \bullet \\ \qquad\qquad (COp_i \wedge T') \Rightarrow (\exists\, AState \bullet T \wedge (\operatorname{pre} AOp_i \Rightarrow AOp_i)) \end{array}$$

□

4. Automata based refinement

Automata offer another perspective on refinement to those given by a process algebra or state-based context. In [30] Lynch and Vaandrager provide a comprehensive treatment of refinement for automata, defining a number of simulation definitions and results relating them. In this section we describe the relationship between automata based refinement and our relational characterisation, hence answering the question raised in [30] concerning their connection.

In Section 4.2 we subsequently consider IO-automata and thus provide a relational characterisation for IO-automata refinement and a set of simulation rules.

4.1. Basic definitions

For our purposes automata are simply LTSs. Initially we do not consider systems with internal evolution, thus there is no special element $\tau \in Act$.

Lynch and Vaandrager use the trace preorder as the definition of refinement; simulations are then used to provide sound and jointly complete techniques. However, slightly confusingly the term *refinement* is also used in [30] to mean a restricted form of downward simulation. To remain consistent with the notation introduced above we use refinement to mean data refinement in a relational setting. Lynch and Vaandrager define simulations in the standard fashion, that is, use Definitions 4 and 5 transcribed into the framework of automata. Thus we have (eliding some obvious quantification):

Definition 30 (Simulations for automata).
Let A and C be automata. A downward simulation from A to C is a relation[2] f over $states(A)$ and $states(C)$ such that

If $s \in init(A)$ then $f(s) \cap init(C) \neq \varnothing$
If $astate \xrightarrow{a} astate'$ and $cstate \in f(astate)$
then $\exists\, cstate' \in f(astate') \bullet cstate \xrightarrow{a} cstate'$

An upward simulation from A to C is a total relation f over $states(A)$ and $states(C)$ such that

If $s \in init(A)$ then $f(s) \subseteq init(C)$
If $astate \xrightarrow{a} astate'$ and $cstate' \in f(astate')$
then $\exists\, cstate \in f(astate) \bullet cstate \xrightarrow{a} cstate'$

□

Along with many other results and examples, the standard soundness and joint completeness results are given for these simulations with respect to the trace preorder.

Lynch and Vaandrager raise a number of questions regarding the relationship between the refinement theory and simulations given for automata and those for data refinement. In particular, they comment in [30]:

Surprisingly, the definition of refinement between data types is completely different from the definition of trace inclusion between automata: informally, one data type is refined by another if any program that uses the former would function at least as well using the latter.

Clearly, an important topic of future research is to study the connection between automata based simulation techniques and methods for data refinement.

As should be clear, the partial relational framework can be used to answer these questions. In particular, the most natural relational embedding of an automaton in that framework is the following.

Definition 31 (Automata embedding).
An automaton $(states(A), Act, \longrightarrow, init(A))$ has the following embedding into the relational model.

$\mathsf{G} == \{*\}$
$\mathsf{State} == states(A)$
$\mathsf{Init} == \{s : init(A) \bullet * \mapsto s\}$
$\mathsf{Op}_i == \{s, s' : states(A) \mid s \xrightarrow{i} s' \bullet s \mapsto s'\}$
$\mathsf{Fin} == \{s : states(A) \bullet s \mapsto *\}$

□

As can easily be seen, with this embedding the definitions in Definition 30 are equivalent to the trace simulations described in Definition 16. This answers the query in [30] in the following way. The automata embedding in Definition 31 is equivalent to the trace embedding given in Definition 9. Furthermore, the automata simulations are equivalent to the trace simulations (Definition 16). Thus with this embedding relational data refinement is trace inclusion (Theorem 1), and the "completely different" goes away, or put another way, with this automata embedding looking at consistency of program behaviour is the same as trace inclusion. The question for connections between automata based simulation techniques and methods for data refinement can now be seen as one of varying the embedding as has been described in this paper.

[2] In order to remain closer to the original formulation, we identify the relation with its corresponding set-valued function.

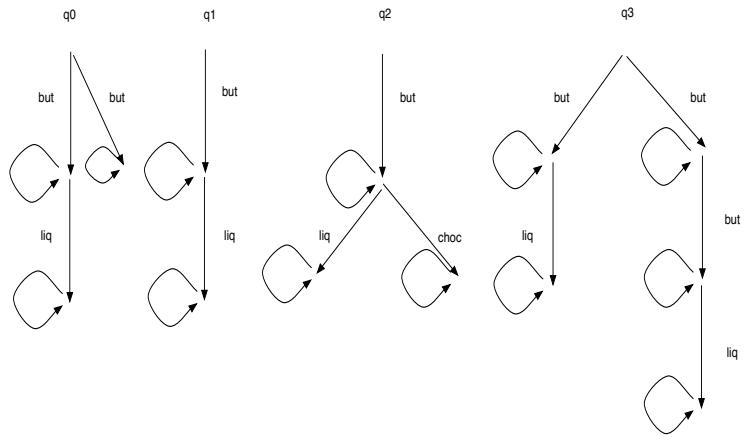

Fig. 2. Four IO automata

4.2. IO automata

IO automata [29] are a class of automata that distinguish explicitly between the input and output of a system, and thus share characteristics with both standard automata and state-based languages such as Z and B. In such a model the set of actions is partitioned into input and output actions. A particular computational interpretation is taken, viz: output actions are actions initiated by the system, while input actions are under the control of the environment. A system can never refuse to perform its input actions, and its output actions can never be blocked by the environment.

While we are considering systems without internal evolution, IO automata do not differ from IO transition systems as discussed by Tretmans in [35], and we use the notation introduced there.

Definition 32 (Partitioned automaton; IO automata).
A *partitioned automaton* is a LTS where the set of actions Act is partitioned into input actions L_I and output actions L_U ($L_I \cup L_U = Act$, $L_I \cap L_U = \varnothing$). An *IO automaton* p is a partitioned automaton for which all input actions are always enabled in any state. That is, for all states p:

$$\forall\, a \in L_I \bullet p \xrightarrow{a}$$

The class of IO automata with input and output actions L_I and L_U is denoted $\mathcal{IOTS}(L_I, L_U)$. □

Example 1. Four IO automata are given in Figure 2 (adapted from [35] where they model a candy dispensing machine for chocolate and liquorice), where $L_I = \{but\}$, $L_U = \{liq, choc\}$. Input actions are always enabled, but may have no effect in a particular state; where this occurs it is denoted graphically with a self-loop without explicit label. □

The input-output testing relation, $\sqsubseteq_{iot}$ is defined via the notion of weakly quiescent traces, which are traces after which no more outputs are possible.

Definition 33 (Weakly quiescent traces, IOTS preorder).
The weakly quiescent traces of a partitioned LTS A are denoted by $\delta\text{–}traces(A)$, and consist of all the traces $\sigma \in Act^*$ such that A **after** σ **ref** L_U. The IOTS preorder is defined for IOTSs A and C by:

$$A \sqsubseteq_{iot} C \text{ iff } \mathcal{T}(C) \subseteq \mathcal{T}(A) \text{ and } \delta\text{–}traces(C) \subseteq \delta\text{–}traces(A)$$

□

The definition of $\sqsubseteq_{iot}$ is the same as that given in [34, 33] for IO-automata, which is shown to be equivalent to the quiescent trace preorder of [36]. Introducing internal actions gives rise to some minor differences between the definitions which we do not repeat here, see Section 5 for a discussion.

The following hold between the systems introduced above: $q_0 \sqsubseteq_{iot} q_1$ but $q_1 \not\sqsubseteq_{iot} q_0$, $q_2 \sqsubseteq_{iot} q_1$, $q_3 \sqsubseteq_{iot} q_1$, but $q_1, q_3 \not\sqsubseteq_{iot} q_2$ and $q_1, q_2 \not\sqsubseteq_{iot} q_3$.

4.2.1. Relational characterisation of IOTS refinement

The IOTS preorder can be defined for arbitrary partitioned LTSs, in which case it is usual to interpret these as under-specified IOTSs, where some input actions are not specified in some states. One might define an alternate relation, $\sqsubseteq_{ioconf}$, specifically for partitioned LTSs. Another approach, given in [14], is to give a *demonic semantics* for process expressions. In this semantics a transition is added for each non-specified input, and after this transition any behaviour is possible. We will follow the latter approach here. We give a relational characterisation of $\sqsubseteq_{iot}$, and in doing so derive simulation rules for it. To do this we will use the partial relational framework, but with some elements of totalisation used to deal with the demonic process semantics.

To define $\sqsubseteq_{iot}$ between arbitrary partitioned LTSs, we define $A \sqsubseteq_{iot} C$ iff $\widehat{A} \sqsubseteq \widehat{C}$, where $\widehat{A}$ is an appropriate relational embedding – i.e., rather than explicitly constructing the IOTS representing its demonic semantics, we give its relational version directly. This relational embedding needs to totalise operations in L_I to represent the fact that they are always enabled, and include a modification of L_U to represent the fact that after an unspecified input any behaviour is possible, and an appropriate finalisation to ensure subsetting of δ–*traces*. We thus make the following definition.

Definition 34 (IOTS embedding).
A partitioned LTS $L = (states, L_I, L_U, \longrightarrow, init)$ is embedded into the relational model as
$\widehat{L} = (\mathsf{State}, \mathsf{Init}, \{\widehat{\mathsf{Op}}_i\}_{i \in L_I \cup L_U}, \mathsf{Fin})$, where

$$\begin{aligned}
&\mathsf{G} == \{*, \delta\}\\
&\mathsf{State} == states \cup \{\bot\}, \text{ where } \bot \notin states\\
&\mathsf{Init} == \{g : \mathsf{G};\ s : init \bullet g \mapsto s\}\\
&\widehat{\mathsf{Op}}_i == \xrightarrow{i} \ \cup \{\bot \mapsto \bot\} \cup \{x : states, y : \mathsf{State} \mid i \in L_I \wedge x \not\xrightarrow{i} \bullet\ x \mapsto y\}\\
&\mathsf{Fin} == \{x : \mathsf{State} \bullet x \mapsto *\} \cup \{(\bot, \delta)\}\\
&\qquad \cup \{x : states \mid (\forall\, i \in L_U \bullet x \not\xrightarrow{i}) \bullet x \mapsto \delta\}
\end{aligned}$$

□

Theorem 6. With the IOTS embedding, data refinement corresponds to the IOTS preorder. That is, let $\widetilde{A}$ denote the IOTS obtained by giving the partitioned LTS A a demonic semantics, then

$$\widehat{A} \sqsubseteq \widehat{C} \text{ iff } \mathcal{T}(\widetilde{C}) \subseteq \mathcal{T}(\widetilde{A}) \text{ and } \delta\text{–}traces(\widetilde{C}) \subseteq \delta\text{–}traces(\widetilde{A})$$

Proof The crucial point to note is that $*$ represents the observation of a trace, and δ the observation of a quiescent trace, i.e., we have that

$$\begin{aligned}
&(g, *) \in tr_{\widehat{A}} \equiv tr \in \mathcal{T}(\widetilde{\mathcal{A}})\\
&(g, \delta) \in tr_{\widehat{A}} \equiv tr \in \delta\text{–}traces(\widetilde{A})
\end{aligned}$$

The latter means that either A **after** tr **ref** L_U, or tr contains an input action that was impossible in A (encoded in the pair $(\bot, \delta) \in \mathsf{Fin}$).

1. Suppose $\widehat{A} \sqsubseteq \widehat{C}$, i.e., for all tr we have $tr_{\widehat{C}} \subseteq tr_{\widehat{A}}$.

Given $tr \in \mathcal{T}(\widetilde{C})$. Then we have $(g, *) \in tr_{\widehat{C}} \subseteq tr_{\widehat{A}}$. Thus $tr \in \mathcal{T}(\widetilde{A})$.

Given $tr \in \delta\text{–}traces(\widetilde{C})$. Then $(*, \delta) \in tr_{\widehat{C}} \subseteq tr_{\widehat{A}}$. Thus $tr \in \delta\text{–}traces(\widetilde{A})$.

2. Suppose that $\mathcal{T}(\widetilde{C}) \subseteq \mathcal{T}(\widetilde{A})$ and $\delta\text{–}traces(\widetilde{C}) \subseteq \delta\text{–}traces(\widetilde{A})$.

Consider a program tr. If $tr_{\widehat{C}}$ is empty (due to some output action being impossible in tr) then $tr_{\widehat{C}} \subseteq tr_{\widehat{A}}$ as required. If $(g, *) \in tr_{\widehat{C}}$ then $tr \in \mathcal{T}(\widetilde{C})$. Thus $tr \in \mathcal{T}(\widetilde{A})$ and consequently $(g, *) \in tr_{\widehat{A}}$. If $(g, \delta) \in tr_{\widehat{C}}$ then $tr \in \delta\text{–}traces(\widetilde{C})$. Thus $tr \in \delta\text{–}traces(\widetilde{A})$ and consequently $(g, \delta) \in tr_{\widehat{A}}$.

Thus $tr_{\widehat{C}} \subseteq tr_{\widehat{A}}$ for any tr, and $\widehat{A} \sqsubseteq \widehat{C}$ as required. □

We now extract the simulation rules that correspond to this notion of refinement.

4.2.2. Simulations

We have embedded an IOTS into a partial relational model, but one augmented with both refusals and a distinguished element, $\perp$. The downward simulation conditions for this data type are, of course:

$$\mathsf{CInit} \subseteq \mathsf{AInit} \mathbin{⨾} \widehat{\mathsf{R}}$$
$$\widehat{\mathsf{R}} \mathbin{⨾} \mathsf{CFin} \subseteq \mathsf{AFin}$$
$$\forall\, i : J \bullet \widehat{\mathsf{R}} \mathbin{⨾} \widehat{\mathsf{COp}}_i \subseteq \widehat{\mathsf{AOp}}_i \mathbin{⨾} \widehat{\mathsf{R}}$$

We will extract the underlying conditions in the usual fashion, however, one will obtain different conditions depending on whether an operation is in L_I or L_U.

First, the initialisation condition, which under the totalisation adds no extra constraints beyond normal. Second, if $i \in L_U$, then $\widehat{\mathsf{Op}}_i == \mathsf{Op}_i \cup \{(\perp, \perp)\}$, so that

$$\widehat{\mathsf{R}} \mathbin{⨾} \widehat{\mathsf{COp}}_i \subseteq \widehat{\mathsf{AOp}}_i \mathbin{⨾} \widehat{\mathsf{R}} \quad \text{iff} \quad \mathsf{R} \mathbin{⨾} \mathsf{COp}_i \subseteq \mathsf{AOp}_i \mathbin{⨾} \mathsf{R}$$

Third, if $i \in L_I$, then $\widehat{\mathsf{Op}}_i$ is the non-blocking totalisation over $states \cup \{\perp\}$, thus

$$\widehat{\mathsf{R}} \mathbin{⨾} \widehat{\mathsf{COp}}_i \subseteq \widehat{\mathsf{AOp}}_i \mathbin{⨾} \widehat{\mathsf{R}} \quad \text{iff} \quad (\mathrm{dom}\, \mathsf{AOp}_i \lhd \mathsf{R}) \mathbin{⨾} \mathsf{COp}_i \subseteq \mathsf{AOp}_i \mathbin{⨾} \mathsf{R} \quad \text{and} \quad \mathrm{ran}\,(\mathrm{dom}\, \mathsf{AOp}_i \lhd \mathsf{R}) \subseteq \mathrm{dom}\, \mathsf{COp}_i$$

Note, that for an IOTS (as opposed to an arbitrary partitioned LTS), input actions are always enabled, and thus in that case this correctness condition reduces to $\mathsf{R} \mathbin{⨾} \mathsf{COp}_i \subseteq \mathsf{AOp}_i \mathbin{⨾} \mathsf{R}$ for L_I.

Finally, the finalisation condition adds in the condition to check for refusals as needed for $\delta\text{–}trace$ inclusion. So $\widehat{\mathsf{R}} \mathbin{⨾} \mathsf{CFin} \subseteq \mathsf{AFin}$ will become

$$\forall\, R \bullet (\forall\, i \in L_U \bullet \neg\, \mathrm{pre}\, COp_i) \Rightarrow (\forall\, i \in L_U \bullet \neg\, \mathrm{pre}\, AOp_i)$$

That is, if states are linked by the retrieve relation and C refuses output actions, then so must A.

For upwards simulations, we use a similar line of reasoning to find that one requires the standard initialisation, blocking correctness for output actions, non-blocking applicability and correctness for input actions together with the refusal condition

$$\forall\, CState \bullet (\forall\, i \in L_U \bullet \neg\, \mathrm{pre}\, COp_i) \Rightarrow \exists\, AState \bullet T \wedge (\forall\, i \in L_U \bullet \neg\, \mathrm{pre}\, AOp_i)$$

which can be combined with the usual totality of upward simulation to give

$$\forall\, CState \bullet \exists\, AState \bullet T \wedge ((\forall\, i \in L_U \bullet \neg\, \mathrm{pre}\, COp_i) \Rightarrow (\forall\, i \in L_U \bullet \neg\, \mathrm{pre}\, AOp_i))$$

These are summarised in the following definition.

Definition 35 (IOTS simulations in Z).

Given Z data types A and C, both representing partitioned LTSs, $J = L_I \cup L_U$. The relation R on $AState \wedge CState$ is an *IOTS downward simulation* from A to C if

$$\begin{array}{l}
\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \wedge R' \\
\forall\, i : L_U;\ AState;\ CState;\ CState' \bullet R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i \\
\forall\, i : L_I;\ AState;\ CState \bullet \operatorname{pre} AOp_i \wedge R \Rightarrow \operatorname{pre} COp_i \\
\forall\, i : L_I;\ AState;\ CState;\ CState' \bullet \operatorname{pre} AOp_i \wedge R \wedge COp_i \\
\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i \\
\forall\, R \bullet (\forall\, i : L_U \bullet \neg \operatorname{pre} COp_i) \Rightarrow (\forall\, i : L_U \bullet \neg \operatorname{pre} AOp_i)
\end{array}$$

The relation T on $AState \wedge CState$ is an *IOTS upward simulation* from A to C if

$$\begin{array}{l}
\forall\, AState';\ CState' \bullet CInit \wedge T' \Rightarrow AInit \\
\forall\, i : L_U;\ AState';\ CState;\ CState' \bullet (COp_i \wedge T') \Rightarrow (\exists\, AState \bullet T \wedge AOp_i) \\
\forall\, i : L_I;\ CState \bullet \exists\, AState \bullet T \wedge (\operatorname{pre} AOp_i \Rightarrow \operatorname{pre} COp_i) \\
\forall\, i : L_I;\ AState';\ CState;\ CState' \bullet \\
\qquad\qquad\qquad (COp_i \wedge T') \Rightarrow (\exists\, AState \bullet T \wedge (\operatorname{pre} AOp_i \Rightarrow AOp_i)) \\
\forall\, CState \bullet \exists\, AState \bullet T \wedge ((\forall\, i : L_U \bullet \neg \operatorname{pre} COp_i) \Rightarrow (\forall\, i : L_U \bullet \neg \operatorname{pre} AOp_i))
\end{array}$$

□

4.2.3. Angelic process semantics

Above we used a totalisation to define $\sqsubseteq_{iot}$ between an LTS and an IOTS, specifically the demonic process semantics discussed in [14]. An alternative view of under-specified input actions is that the under-specification represents an implicit *skip*. Such an interpretation was introduced in [36] and discussed in [14], where it is called the angelic process semantics.

The relational embedding of such a semantics only alters the input action component from that we defined above. Thus, when deriving simulation conditions for such an embedding, the initialisation, refusal conditions and correctness for output actions remain the same.

For input actions, they are embedded as

$$\widehat{\mathsf{Op}}_i == \mathsf{Op}_i \cup \{(state, state) \mid state \stackrel{i}{\not\longrightarrow}\}$$

and the downward simulation condition

$$\widehat{\mathsf{R}} \fatsemi \widehat{\mathsf{COp}}_i \subseteq \widehat{\mathsf{AOp}}_i \fatsemi \widehat{\mathsf{R}}$$

evaluates to

$$\mathsf{R} \fatsemi (\mathsf{COp}_i \cup (\overline{\operatorname{dom} \mathsf{COp}_i} \lhd skip)) \subseteq (\mathsf{AOp}_i \cup (\overline{\operatorname{dom} \mathsf{AOp}_i} \lhd skip)) \fatsemi \mathsf{R}$$

However, this does not have a particular interesting simplification.

5. Internal events and divergence

The consideration above has dealt with the basic structure of automata and IO automata, but without internal events or the potential consideration of divergence that they can give rise to. We briefly discuss some of these aspects here.

The definitions of automata and refinement given by Lynch and Vaandrager in [30] have in their full generality an internal action $\tau \in Act$. Their definitions of refinement and simulation use this in the standard way, that is, allow silent evolution before and after external events, formally characterised by a change from $\stackrel{a}{\longrightarrow}$ to $\stackrel{a}{\Longrightarrow}$ in relevant places in the definitions. Thus, for example, the definition of a downward simulation becomes:

If $s \in init(A)$ then $f(s) \cap init(C) \neq \varnothing$
If $astate \xrightarrow{a} astate'$ and $cstate \in f(astate)$
then $\exists\, cstate' \in f(astate') \bullet cstate \overset{a}{\Longrightarrow} cstate'$

All the standard results carry over, and in a relational embedding the simulations can be adapted to allow such internal behaviour. Details of how to do this are in [6]. Lynch and Vaandrager do not, however, consider divergence, that is their simulation conditions are ones which ignore divergence.

As shown in [4] it is possible to extend the relational framework with the capability to model various types of divergence. To do so the embeddings that are used need to incorporate internal behaviour into the operations, initialisation and possibly finalisation. Once that is done, if a refinement relation ignores divergence, then there are obviously no further requirements. Otherwise, the embeddings need to ensure the correct observations are made when the final state records divergence. In the case of catastrophic interpretations (i.e., ones where divergence is propagated to all further behaviours) the embeddings need to generate arbitrary behaviour from the point of divergence onwards, and propagate this into all subsequent operations.

We record divergence using a special value ω which as usual is assumed not to be included in any local or global state space. For any set S, let $\mathsf{S}_\omega = \mathsf{S} \cup \{\omega\}$. Then we can make the following embeddings. First, the refinement relations that ignore divergence.

Trace refinement As discussed also in [18], trace refinement in the absence of internal operations "is" the partial relations model. Including also internal operations is relatively simple. We only need to include internal operations after initialisation and operations.

Definition 36 (Embedding trace refinement ignoring divergence).
An i-data type $\mathsf{D} = (\mathsf{State}, \mathsf{G}, \mathsf{Init}, \{\mathsf{Op}_k\}_{k \in J}, i, \mathsf{Fin})$ is embedded as the data type $\widehat{\mathsf{D}} = (\mathsf{State}, \mathsf{G}, \widehat{\mathsf{Init}}, \{\widehat{\mathsf{Op}_k}\}_{k \in J}, \mathsf{Fin})$ where

$\widehat{\mathsf{Op}} = \mathsf{Op} \fatsemi i^*$
$\widehat{\mathsf{Init}} = \mathsf{Init} \fatsemi i^*$

□

If D only makes trivial observations, i.e., $\mathsf{G} = \{*\}$, then so does $\widehat{\mathsf{D}}$, and furthermore their traces are identical, i.e., for every sequence p over J

$$p_{\widehat{\mathsf{D}}} \;=\; \bigcup_{q \in (J \cup \{i\})^* \wedge q \restriction J = p} q_{\mathsf{D}}$$

(where $s \restriction A$ is the largest subsequence of s whose elements are all in A) or equivalently (recall that a non-empty result indicates a trace being possible in this basic model):

$$p_{\widehat{\mathsf{D}}} \neq \varnothing \;\equiv\; \exists\, q \bullet q \restriction J = p \wedge q_{\mathsf{D}} \neq \varnothing$$

This can easily be proved by induction over the length of p. The simulation rules deriving from this are those of Definitions 4 and 5 with internal behaviour inserted after all occurrences of operations and initialisation. In the absence of (observed) divergence, joint completeness of the simulations follows from joint completeness of the partial relations simulations, plus the fact that the data type with internal operations is refinement equivalent to its embedding as in Definition 36, see also [16] for the latter point.

Note that this trace refinement relation is different from trace inclusion in the CSP failures-divergences model, as that does take divergence into account. An embedding for CSP trace refinement would be a simplification of the failures-divergences embedding, with a trivial observation at finalisation instead of refusals, as follows:

Definition 37 (Embedding trace refinement (CSP f-d model)).

An i-data type $\mathsf{D} = (\mathsf{State}, \mathsf{G}, \mathsf{Init}, \{\mathsf{Op}_k\}_{k \in J}, i, \mathsf{Fin})$ is embedded as the data type $\widehat{\mathsf{D}} = (\mathsf{State}_\omega, \mathsf{G}, \widehat{\mathsf{Init}}, \{\widehat{\mathsf{Op}_k}\}_{k \in J}\}, \widehat{\mathsf{Fin}})$ where

$$\begin{aligned}
&\widehat{\mathsf{Init}} = \mathsf{Init} \fatsemi i^* \ \cup \ \textbf{if} \ \mathrm{divi}\,\mathsf{Init} \ \textbf{then} \ \mathsf{G} \times \mathsf{State}_\omega \\
&\widehat{\mathsf{Op}} = \mathsf{Op} \fatsemi i^* \ \cup \ \mathrm{div}\,\mathsf{Op} \times \mathsf{State}_\omega \\
&\widehat{\mathsf{Fin}} = \mathsf{Fin} \ \cup \ \{\omega\} \times \mathsf{G} \\
&\mathrm{div}\,\mathsf{Op} =_{def} \{s : \mathsf{State} \mid \exists\, s' : \mathsf{State} \bullet (s, s') \in \mathsf{Op} \wedge s' \xrightarrow{i^\infty}\} \\
&\mathrm{divi}\,\mathsf{Init} =_{def} \exists\, s : \mathrm{ran}\,\mathsf{Init} \bullet s \xrightarrow{i^\infty}
\end{aligned}$$

□

The derivation of simulation rules leads to the following definition.

Definition 38 (Simulations for trace refinement (CSP f-d model)).
A relation R between AState and CState is a downward simulation between i-data types A and C iff $\forall\, k : J$ we have:

$$\begin{aligned}
&\textbf{if} \ \mathrm{divi}\,\mathsf{CInit} \ \textbf{then} \ \mathrm{divi}\,\mathsf{AInit} \ \textbf{else} \ \mathsf{CInit} \fatsemi i_{\mathsf{C}}^* \subseteq \mathsf{AInit} \fatsemi i_{\mathsf{A}}^* \fatsemi \mathsf{R} \\
&\mathsf{R} \fatsemi \mathsf{CFin} \subseteq \mathsf{AFin} \\
&(\mathrm{div}\,\mathsf{AOp}_k) \lhd \mathsf{R} \fatsemi \mathsf{COp}_k \fatsemi i_{\mathsf{C}}^* \subseteq \mathsf{AOp}_k \fatsemi i_{\mathsf{A}}^* \fatsemi \mathsf{R} \\
&\mathrm{dom}(\mathsf{R} \rhd \mathrm{div}\,\mathsf{COp}_k) \subseteq \mathrm{div}\,\mathsf{AOp}_k
\end{aligned}$$

A relation T between CState and AState is an upward simulation between i-data types A and C iff $\forall\, k : J$

$$\begin{aligned}
&\textbf{if} \ \mathrm{divi}\,\mathsf{CInit} \ \textbf{then} \ \mathrm{divi}\,\mathsf{AInit} \ \textbf{else} \ \mathsf{CInit} \fatsemi i_{\mathsf{C}}^* \fatsemi \mathsf{T} \subseteq \mathsf{AInit} \fatsemi i_{\mathsf{A}}^* \\
&\mathsf{CFin} \subseteq \mathsf{T} \fatsemi \mathsf{AFin} \\
&\mathrm{dom}(\mathsf{T} \rhd \mathrm{div}\,\mathsf{AOp}_k) \lhd \mathsf{COp}_k \fatsemi i_{\mathsf{C}}^* \fatsemi \mathsf{T} \subseteq \mathsf{T} \fatsemi \mathsf{AOp}_k \fatsemi i_{\mathsf{A}}^* \\
&\mathrm{div}\,\mathsf{COp}_k \subseteq \mathrm{dom}(\mathsf{T} \rhd \mathrm{div}\,\mathsf{AOp}_k)
\end{aligned}$$

□

Reduction The embedding for reduction is a simplification of that for failures-divergences refinement, e.g. as given in [6], introducing an extra component E recording refused events, but removing the case distinctions and special treatment arising from infinite internal evolution.

Definition 39 (Embedding reduction).
An i-data type $\mathsf{D} = (\mathsf{State}, \mathsf{G}, \mathsf{Init}, \{\mathsf{Op}_k\}_{k \in J}, i, \mathsf{Fin})$ is embedded as the data type $\widehat{\mathsf{D}} = (\mathsf{State}, \mathsf{G} \times \mathbb{P}\, J, \widehat{\mathsf{Init}}, \{\widehat{\mathsf{Op}_k}\}_{k \in J}, \widehat{\mathsf{Fin}})$ where

$$\begin{aligned}
&\widehat{\mathsf{Init}} = \{((g, E), s) : (\mathsf{G} \times \mathbb{P}\, J) \times \mathsf{State} \mid (g, s) \in \mathsf{Init} \fatsemi i^*\} \\
&\widehat{\mathsf{Op}} = \mathsf{Op} \fatsemi i^* \\
&\widehat{\mathsf{Fin}} = \{(s, (g, E)) : \mathsf{State} \times (\mathsf{G} \times \mathbb{P}\, J) \mid \\
&\qquad (s, g) \in \mathsf{Fin} \wedge \forall\, k : E \bullet s \notin \mathrm{dom}(i^* \fatsemi \mathsf{Op}_k)\}
\end{aligned}$$

□

Note that the change to initialisation is only to account for the extra component E in the global state. The resulting simulation rules are identical to those for failures refinement with internal evolution added after all operations and initialisation, and before operations in precondition (refusal) computation. The multiple components observed in finalisation imply that the simulations are not in general complete: the simulations as given impose separate conditions on each component, whereas due to dependencies between the components weaker conditions may suffice. However, for *trivial* original finalisations, due to the same normal form argument as given for trace refinement above, these rules inherit the joint completeness of the failures refinement rules proved e.g., by Josephs [25].

Non-catastrophic divergence In a non-catastrophic interpretation, divergence is a property only of the state (whether it admits infinite internal evolution) and not of the trace (whether it may have come through such a state). Thus, embeddings for associated refinement relations are significantly simpler, not having to propagate divergence from one state to the next, nor having to introduce arbitrary behaviour in such states.

Automata with additional structure over the basic model given in [30] include IO automata [29] and IO transition systems [35]. Although they coincide in the absence of internal events, in general they offer slightly differing models.

IO transition systems offer the same perspective as the basic automata of [30] in that they include internal evolution in standard ways in their definition of refinement and simulation, and ignore divergence (indeed, only divergence-free systems are considered in [35]). Although internal actions do not alter the standard view of refinement, IO transition systems differ marginally from IO automata even for convergent systems since IO transition systems only require weak input enabling as opposed to the strong input enabling as required in [29].

As noted above the use of weakly quiescent traces differs from the (original) definition of quiescence given in [36], where quiescence requires the absence of both output and internal actions. Tretmans [35] comments that for divergent-free systems the two notions coincide but views the stronger quiescence to be counter-intuitive in the presence of divergence. The exact role of divergence in IO transition systems is still unresolved: "*for a precise comparison [of IOTS with IO automata] a more elaborate investigation of divergence in IOTS is necessary*" [35].

In Section 4.2.1 we provided a relational characterisation of IOTS refinement via a demonic semantics for under-specification of input actions. Specifically in this semantics a transition is added for each non-specified input, and after this transition any behaviour is possible. Clearly, one could link this demonic behaviour to divergence arising from internal evolution. However, [14] takes a much simpler approach and although it includes internal events (and a more elaborate treatment of fairness), it does not view the transition that is added for each non-specified input as divergence in the sense of potential unbounded internal evolution. One avenue of future work would be to incorporate such a view and to understand what the consequences of doing so are.

6. Conclusions

In this paper we have derived simulations for relational embeddings of a number of refinement preorders found in process algebras, and then explored the relation between automata based refinement and notions of refinement for relational data types and process algebras.

Although downward and upward simulations (Definitions 4 and 5) are complete, their totalised versions are not. However, complete simulations can be given for each semantics, e.g. the failures semantics simulations are known to be complete. A separate completeness proof for simulations is needed in each embedding, this is a line for future exploration.

The notions of trace refinement and basic refinement for automata were shown to coincide through sharing the same sound and complete set of simulation rules. Refinement for IO automata (IO transition systems [35]) was shown to be different from any refinement relation considered so far in our relational concurrent refinement programme [17, 6, 18, 4]. This was due to the separation of input and output actions, requiring a different treatment in refinement, each sharing some characteristics with previously considered methods of "totalising" operations.

There are still some unanswered questions. Some deal with the notion of divergence in IOTS and IO automata as described in the last section, others deal with the issue of granularity of transition, that is, non-atomic refinement. This is particularly pertinent to the structure offered by IOTS and IO automata since they already distinguish between input and output actions, and how this can be incorporated into theories of non-atomic refinement (e.g., as in those offered in [19, 20, 21]) remains to be seen.

References

[1] J.-R. Abrial. *Modelling in Event-B*. CUP, 2010.

[2] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science Inc., New York, NY, USA, 2001.

[3] E. A. Boiten and J. Derrick. Unifying concurrent and relational refinement. *ENTCS*, 70, 2002. Proceedings REFINE'02, Editors: J. Derrick, E. A. Boiten, J. von Wright and J. C. P. Woodcock.

[4] E.A. Boiten and J. Derrick. Modelling divergence in relational concurrent refinement. In M. Leuschel and H. Wehrheim, editors, *IFM 2009: Integrated Formal Methods*, volume 5423 of *LNCS*, pages 183–199. Springer Verlag, February 2009.

[5] E.A. Boiten and J. Derrick. Incompleteness of relational simulations in the blocking paradigm. *Science of Computer Programming*, 75(12):1262–1269, 2010.

[6] E.A. Boiten, J. Derrick, and G. Schellhorn. Relational concurrent refinement II: Internal operations and outputs. *Formal Aspects of Computing*, 21(1-2):65–102, 2009.

[7] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1988.

[8] C. Bolton and J. Davies. Refinement in Object-Z and CSP. In M. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods (IFM 2002)*, volume 2335 of *Lecture Notes in Computer Science*, pages 225–244. Springer-Verlag, 2002.

[9] C. Bolton and J. Davies. A singleton failures semantics for Communicating Sequential Processes. *Formal Aspects of Computing*, 18:181–210, 2006.

[10] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification, VIII*, pages 63–74, Atlantic City, USA, June 1988. North-Holland.

[11] E. Brinksma and G. Scollo. Formal notions of implementation and conformance in LOTOS. Technical Report INF-86-13, Dept of Informatics, Twente University of Technology, 1986.

[12] E. Brinksma, G. Scollo, and C. Steenbergen. Process specification, their implementation and their tests. In B. Sarikaya and G. v. Bochmann, editors, *Protocol Specification, Testing and Verification, VI*, pages 349–360, Montreal, Canada, June 1986. North-Holland.

[13] R. de Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24(2):211–237, 1987.

[14] R. de Nicola and R. Segala. A process algebraic view of I/O automata. *Theoretical Computer Science*, 138:391–423, 1995.

[15] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. CUP, 1998.

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

[17] J. Derrick and E.A. Boiten. Relational concurrent refinement. *Formal Aspects of Computing*, 15(1):182–214, November 2003.

[18] J. Derrick and E.A. Boiten. More relational refinement: traces and partial relations. *Electronic Notes in Theoretical Computer Science*, 214:255–276, May 2008. Proceedings of REFINE 2008 (Turku, May 2008).

[19] J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In J. Davies and J. Gibbons, editors, *IFM*, volume 4591 of *Lecture Notes in Computer Science*, pages 195–214. Springer, 2007.

[20] J. Derrick and H. Wehrheim. Using coupled simulations in non-atomic refinement. In *ZB 2003: Formal Specification and Development in Z and B*, number 2651 in LNCS, pages 127–147. Springer, 2003.

[21] J. Derrick and H. Wehrheim. Non-atomic refinement in Z and CSP. In *ZB2005*, LNCS. Springer, 2005.

[22] He Jifeng and C.A.R. Hoare. Prespecification and data refinement. In *Data Refinement in a Categorical Setting*, Technical Monograph, number PRG-90. Oxford University Computing Laboratory, November 1990.

[23] He Jifeng, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986.

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

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

[26] R. Langerak. A testing theory for LOTOS using deadlock detection. In *Protocol Specification Testing and Verification IX*, pages 87–98. North-Holland, 1989.

[27] R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, University of Twente, The Netherlands, 1992.

[28] G. Leduc. *On the Role of Implementation Relations in the Design of Distributed Systems using LOTOS*. PhD thesis, University of Liège, Liège, Belgium, June 1991.

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

[30] N. Lynch and F. Vaandrager. Forward and backward simulations I.: untimed systems. *Information and Computation*, 121(2):214–233, 1995.

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

[32] S. Reeves and D. Streader. Data refinement and singleton failures refinement are not equivalent. *Formal Aspects of Computing*, 20(3):295–301, 2008.

[33] R. Segala. Quiescence, fairness, testing, and the notion of implementation (extended abstract). In *International Conference on Concurrency Theory*, pages 324–338, 1993.

[34] R. Segala. Quiescence, Fairness, Testing, and the Notion of Implementation. *Information and Computation*, 138(2):194–210, 1997.

[35] J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In T. Margaria and B. Steffen, editors, *Second*

Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96), volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996.

[36] F.W. Vaandrager. On the relationship between process algebra and input/output automata. In *Logic in Computer Science*, pages 387–398, 1991.

[37] R.J. van Glabbeek. The linear time – branching time spectrum II; the semantics of sequential systems with silent moves (extended abstract). In E. Best, editor, Proceedings *CONCUR'93,* 4th International Conference on *Concurrency Theory*, Hildesheim, Germany, August 1993, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.

[38] R.J. van Glabbeek. The linear time - branching time spectrum I. The semantics of concrete sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. North-Holland, 2001.

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