Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 259 (2009) 21–34
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Relational Concurrent Refinement: Automata

John Derrick[1]

Department of Computer Science, University of Sheffield, Sheffield, UK

Eerke Boiten[2]

School of Computing, University of Kent, Canterbury, Kent, 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. In models of concurrency, refinement takes a number of different forms depending on the exact notion of observation chosen, which can include the events a system is prepared to accept or refuse, or depend on explicit properties of states and transitions. In this paper we continue our program of deriving relational simulation conditions for behavioural notions of refinement by defining embeddings into the relational model that extend our framework to include various notions of automata based refinement.

Keywords: Data refinement, Z, simulations, automata-based refinements.

1 Introduction

The last 10 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.

In a process algebra such as CSP [15] 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

[1] Email: J.Derrick@dcs.shef.ac.uk
[2] Email: E.A.Boiten@kent.ac.uk

doi:10.1016/j.entcs.2009.12.015

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 of many prominent process algebraic refinement relations is given in [25].

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 inclusion of behaviour for all programs, and is normally verified through *simulations* [9]. For a complete method, often two kinds of simulations are defined: downward and upward 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 [6,7] and Reeves and Streader [19]. Our work on relational concurrent refinement started [5,10] 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 [10], including also outputs and internal operations [4], and for readiness refinement [10]. These were mostly based on the total relations model (as described below). Trace refinement and other relations based on the partial relations model were considered in [11], and different interpretations of divergent behaviour in [3]. 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 concurrent refinement notions in the context of automata based specification. In Section 2 we provide the basic definitions and background. In Section 3 we introduced automata and IO automata, their refinement notions, and derive their relational simulation rules. We conclude in Section 4.

2 Background

This background section presents the standard refinement theory [12] 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

[14], traditionally received the most attention. The standard refinement theory of Z [26,12], for example, is based on this version of the theory. However, later publications by He and Hoare, in particular [13], 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 [9] also present the partial relations theory, without putting much emphasis on this aspect.

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 $\mathsf{sans\ serif\ font}$.

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.*

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* $\mathsf{p_D} = \mathsf{Init} \mathbin{⨾} \mathsf{Op}_{\mathsf{p_1}} \mathbin{⨾} ... \mathbin{⨾} \mathsf{Op}_{\mathsf{p_n}} \mathbin{⨾} \mathsf{Fin}$*.*

As usual we assume that the data types 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*,* $\mathsf{p_C} \subseteq \mathsf{p_A}$*.*

Downward and *upward* simulations [9] form a sound and jointly complete [14,9] 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*

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

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*

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

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 [11] 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

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

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 [12].

Although in this paper we explore 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 [12] 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 [4].

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 7 (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*

$\mathsf{Init} == \{Init \bullet * \mapsto \theta State'\}$

$\mathsf{Op} == \{Op \bullet \theta State \mapsto \theta State'\}$

$\mathsf{Fin} == \{State \bullet \theta State \mapsto *\}$

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 8 (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*

$$\forall CState' \bullet CInit \Rightarrow \exists AState' \bullet AInit \wedge R'$$

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

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

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

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

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 produces a condition that the simulation is total on the concrete state.

Definition 9 (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*

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

In the blocking model, the correctness condition becomes

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

2.4 Process algebraic based refinement

Process algebras [15,18,2] provide a means to describe and verify 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 means of a semantics which associates a labelled transition system (LTS) to each term. 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 [24,25]. We will need the usual notation for labelled transition systems (LTSs):

Definition 10 (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 $states(L) = A$ 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 [11] we showed how different process algebraic preorders can be embedded into the relational model. Here we review how this is achieved for the trace preorder which defines refinement as trace inclusion. In the next section, we provide that type of characterisation for each notion of automata refinement.

Definition 11 *$\sigma \in Act^*$ 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 $\mathcal{T}(q) \subseteq \mathcal{T}(p)$.*

As observed previously [10] the partial relations model records exactly trace information for the embedding with trivial finalisation in Definition 7: possible traces lead to the single global value; impossible traces produce the empty relation. To prove the correspondence between trace preorder and data refinement we need to provide a definition of the traces of an abstract data type.

Definition 12 *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* $\exists\, State' \bullet Init \mathbin{⨾} Op_{i_1} \mathbin{⨾} \ldots \mathbin{⨾} Op_{i_n}$. *We denote the traces of an ADT A by* $\mathcal{T}(A)$.

Then the following can be proved [11]:

Theorem 2.1 *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,

$$\mathsf{A} \sqsubseteq_{data} \mathsf{C} \textit{ iff } \mathcal{T}(C) \subseteq \mathcal{T}(A)$$

On the basis of this result, we can extract the simulation rules that correspond to this notion of refinement from the partial relation simulations as applied to this embedding (i.e., without a totalisation in between). These are of course the rules for standard Z refinement but omitting applicability of operations, as used also e.g., in Event-B [1]. The conditions for a downward simulation in the partial relational model are (c.f. Definition 4):

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

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, hence we have:

Definition 13 (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*

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

3 Automata based refinement

Automata [17] offer another perspective on refinement to those given by a process algebra or state-based context. In [17] 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 [17] concerning their connection.

In Section 3.2 we subsequently consider IO-automata and thus provide a relational characterisation for IO-automata refinement and a set of simulation rules.

3.1 Basic definitions

For our purposes automata are simply LTSs. 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 [17] 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[3]. Thus we have (eliding some obvious quantification):

Definition 14 (Simulations for automata) *Let A and C be automata. A downward simulation from A to C is a relation 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 [17]:

> 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.

[3] Downward and upward simulations are called forward and backward simulations, respectively, as is sometimes the case.

Definition 15 (Automata embedding) *An automaton* $(states(A), Act, \longrightarrow, init(A))$ *has the following embedding into the relational model.*

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

As can easily be seen, with this embedding the definitions in Definition 14 are equivalent to the trace simulations described in Definition 13. This answers the query in [17] in the following way. The automata embedding in Definition 15 is equivalent to the trace embedding given in Definition 7. Furthermore, the automata simulations are equivalent to the trace simulations (Definition 13). Thus with this embedding relational data refinement is trace inclusion (Theorem 2.1), 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.

3.2 IO automata

IO automata [16] 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.

Since we are considering systems without internal evolution in this paper, IO automata do not differ from IO transition systems as discussed by Tretmans in [22], and we use the notation introduced there.

Definition 16 (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 3.1 Four IO automata are given in Figure 1 (adapted from [22]), 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

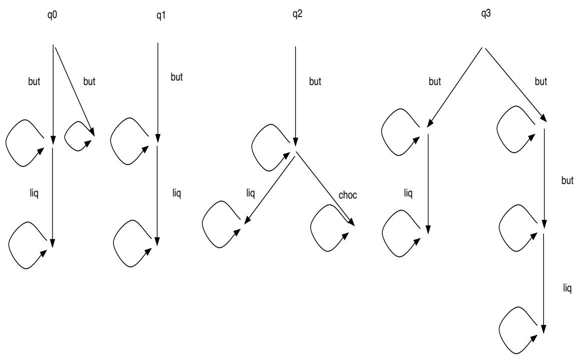

Fig. 1. Four IO automata

self-loop without explicit label.

To define refinement between IOTSs we use the idea of refusals sets after a particular trace given in the definition below.

Definition 17 (Refusals after a trace) *Let p be an LTS, σ a trace of p, and $X \subseteq Act$. Then p **after** σ **ref** X iff $\exists\, q \bullet p \xrightarrow{\sigma} q$ and $X \cap next(q) = \varnothing$*

This was used in [11] to define conformance and extension, however, it can also be used to define the input-output testing relation, $\sqsubseteq_{iot}$. It is defined via the notion of weakly quiescent traces, which are traces after which no more outputs are possible.

Definition 18 (Weakly quiescent traces, IOTS preorder) *The weakly quiescent traces of a partitioned LTS A are denoted by δ–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: $A \sqsubseteq_{iot} C$ iff $\mathcal{T}(C) \subseteq \mathcal{T}(A)$ and δ–traces$(C) \subseteq \delta$–traces(A)*

The definition of $\sqsubseteq_{iot}$ is the same as that given in [21,20] for IO-automata, which is shown to be equivalent to the quiescent trace preorder of [23]. Introducing internal actions gives rise to some minor differences between the definitions which we do not repeat here, see references given above for more details.

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 $q_1, q_2 \not\sqsubseteq_{iot} q_3$.

3.2.1 *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 [8], 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 19 (IOTS embedding) *A partitioned LTS* $L = (\mathit{states}, L_I, L_U, \longrightarrow, \mathit{init})$ *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*

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

Theorem 3.2 *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* $\widehat{A} \sqsubseteq \widehat{C}$ *iff* $\mathcal{T}(\widetilde{C}) \subseteq \mathcal{T}(\widetilde{A})$ *and* $\delta\text{–}\mathit{traces}(\widetilde{C}) \subseteq \delta\text{–}\mathit{traces}(\widetilde{A})$.

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

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

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 $(\bot, L_U) \in \mathsf{Fin}$).

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

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

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

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

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

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

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

3.2.2 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:

$$\begin{array}{l} \mathsf{CInit} \subseteq \mathsf{AInit} \operatorname{\text{;}} \widehat{\mathsf{R}} \\ \widehat{\mathsf{R}} \operatorname{\text{;}} \mathsf{CFin} \subseteq \mathsf{AFin} \\ \forall\, i : I \bullet \widehat{\mathsf{R}} \operatorname{\text{;}} \widehat{\mathsf{COp}_i} \subseteq \widehat{\mathsf{AOp}_i} \operatorname{\text{;}} \widehat{\mathsf{R}} \end{array}$$

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

$$\widehat{\mathsf{R}} \operatorname{\text{;}} \widehat{\mathsf{COp}_i} \subseteq \widehat{\mathsf{AOp}_i} \operatorname{\text{;}} \widehat{\mathsf{R}} \quad \text{iff} \quad \mathsf{R} \operatorname{\text{;}} \mathsf{COp}_i \subseteq \mathsf{AOp}_i \operatorname{\text{;}} \mathsf{R}$$

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

$$\widehat{\mathsf{R}} \operatorname{\text{;}} \widehat{\mathsf{COp}_i} \subseteq \widehat{\mathsf{AOp}_i} \operatorname{\text{;}} \widehat{\mathsf{R}} \quad \text{iff} \quad \begin{array}{l} (\operatorname{dom} \mathsf{AOp}_i \lhd \mathsf{R}) \operatorname{\text{;}} \mathsf{COp}_i \subseteq \mathsf{AOp}_i \operatorname{\text{;}} \mathsf{R} \quad \text{and} \\ \operatorname{ran} (\operatorname{dom} \mathsf{AOp}_i \lhd \mathsf{R}) \subseteq \operatorname{dom} \mathsf{COp}_i \end{array}$$

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 $\mathsf{R} \operatorname{\text{;}} \mathsf{COp}_i \subseteq \mathsf{AOp}_i \operatorname{\text{;}} \mathsf{R}$ for L_I.

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

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

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

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

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

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

These are summarised in the following definition.

Definition 20 (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*

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

3.2.3 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 [8]. An alternative view of under-specified input actions is that the under-specification represents an implicit *skip*. Such an interpretation was introduced in [23] and discussed in [8], 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

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

and the downward simulation condition

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

evaluates to

$$\mathsf{R} \mathbin{⨾} (\mathsf{COp}_i \cup (\overline{\text{dom}\, \mathsf{COp}_i} \lhd skip)) \subseteq (\mathsf{AOp}_i \cup (\overline{\text{dom}\, \mathsf{AOp}_i} \lhd skip)) \mathbin{⨾} \mathsf{R}$$

However, this does not have a particular interesting simplification.

4 Conclusions

This paper has explored the relation between automata based refinement and notions of refinement for relational data types and process algebras. 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 [22]) was shown to be different from any refinement relation considered so far in our relational concurrent refinement programme [10,4,11,3]. 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.

References

[1] Abrial, J.-R., D. Cansell and D. Méry, *Refinement and reachability in event-B*, in: H. Treharne, S. King, M. C. Henson and S. A. Schneider, editors, *ZB*, Lecture Notes in Computer Science **3455** (2005), pp.

222–241.

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

[3] Boiten, E. and J. Derrick, *Modelling divergence in relational concurrent refinement*, in: M. Leuschel and H. Wehrheim, editors, *IFM 2009: Integrated Formal Methods*, LNCS **5423** (2009), pp. 183–199. URL http://www.cs.kent.ac.uk/pubs/2009/2838

[4] Boiten, E., J. Derrick and G. Schellhorn, *Relational concurrent refinement II: Internal operations and outputs*, Formal Aspects of Computing **21** (2009), pp. 65–102. URL http://www.cs.kent.ac.uk/pubs/2007/2633

[5] Boiten, E. A. 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.

[6] Bolton, C. and J. Davies, *Refinement in Object-Z and CSP*, in: M. Butler, L. Petre and K. Sere, editors, *Integrated Formal Methods (IFM 2002)*, Lecture Notes in Computer Science **2335** (2002), pp. 225–244.

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

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

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

[10] Derrick, J. and E. Boiten, *Relational concurrent refinement*, Formal Aspects of Computing **15** (2003), pp. 182–214.

[11] Derrick, J. and E. Boiten, *More relational refinement: traces and partial relations*, Electronic Notes in Theoretical Computer Science **214** (2008), pp. 255–276, proceedings of REFINE 2008 (Turku, May 2008). URL http://www.cs.kent.ac.uk/pubs/2008/2722

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

[13] 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, 1990 .

[14] He Jifeng, C. A. R. Hoare and J. W. Sanders, *Data refinement refined*, in: B. Robinet and R. Wilhelm, editors, *Proc. ESOP 86*, Lecture Notes in Computer Science **213** (1986), pp. 187–196.

[15] Hoare, C. A. R., "Communicating Sequential Processes," Prentice Hall, 1985.

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

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

[18] Milner, R., "Communication and Concurrency," Prentice-Hall, 1989.

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

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

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

[22] Tretmans, J., *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)*, Lecture Notes in Computer Science **1055** (1996), pp. 127–146.

[23] Vaandrager, F. W., *On the relationship between process algebra and input/output automata*, in: *Logic in Computer Science*, 1991, pp. 387–398. URL citeseer.ist.psu.edu/vaandrager91relationship.html

[24] van Glabbeek, R. J., *The linear time - branching time spectrum*, in: J. C. M. Baeten and J. W. Klop, editors, *CONCUR 90*, Lecture Notes in Computer Science **458** (1990), pp. 278–297.

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

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