Non-atomic Refinement in Z

John Derrick and Eerke Boiten

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
J.Derrick@ukc.ac.uk

Abstract. This paper discusses the refinement of systems specified in Z when we relax the assumption that the refinement will preserve the atomicity of operations. Data refinement is a well established technique for transforming specifications of abstract data types into ones which are closer to an eventual implementation. To verify a refinement a retrieve relation is used which relates the concrete to abstract states and allow the comparison between the data types to be made on a step by step basis by comparing an abstract operation with its concrete counterpart. A step by step comparison is possible because the two abstract data types are assumed to be *conformal*, i.e. there is a one-one correspondence between abstract and concrete operations, so each abstract operation has a concrete counterpart. In this paper we relax that assumption to discuss refinements where an abstract operation is refined by, not one, but a sequence of concrete operations. Such non-conformal or non-atomic refinements arise naturally in a number of settings and we illustrate our derivations with a simple example of a bank accounting system.

Keywords: Specification; Refinement; Z; Non-atomic refinement; Non-atomic operations.

1 Introduction

This paper discusses the refinement of systems specified in state-based specification languages such as Z [8] when we relax the assumption that refinements preserve the atomicity of operations.

State-based languages have gained a certain amount of acceptance in the software community as an industrial strength formal method. As a canonical example, we will concentrate on Z in this paper, although the methods we derive could be applied to other state-based languages. Z is a state-based language whose specifications are written using set theory and first order logic. Abstract data types are specified in Z using the so called "state plus operations" style, where a collection of operations describe changes to the state space. The state space, initialisation and operations are described as *schemas*, and the schema calculus has proved to be an enduring structuring mechanism for specifying complex systems. These schemas, and the operations that they represent, can be understood as (total or partial) relations on the underlying state space.

In addition to specifying a system, we might also wish to develop, or *refine*, it further. This idea of data refinement is a well established technique for transforming specifications of abstract data types into ones which are closer to an eventual implementation. Such a refinement might typically weaken the precondition of an operation, remove some non-determinism or even alter the state space of the specification. The conditions under which a development is a correct refinement are encapsulated into two refinement (or simulation) rules: downward and upward simulations [10]. To verify a refinement the simulations use a retrieve relation which relates the concrete to abstract states and allow the comparison between the data types to be made on a step by step basis by comparing an abstract operation with its concrete counterpart. Versions of the simulation rules for Z are given in [10].

The step by step comparison that a simulation makes is possible because the two abstract data types are assumed to be *conformal* [6], i.e. there is a one-one correspondence between abstract and concrete operations, so each abstract operation has a concrete counterpart. In this paper we relax that assumption to discuss refinements where an abstract operation is refined by, not one, but a sequence of concrete operations. The motivation for such a refinement is twofold: we might wish to reflect the structure of the eventual implementation in a specification without having to make that choice at an initial abstract level, and going further, we might wish to allow interleavings of concrete operations for the sake of efficiency. For example, we might want to describe an abstract operation AOp in the first instance, but in a subsequent development describe how AOp is implemented as a sequence of concrete operations: COp_1 followed by COp_2.

Such non-conformal or non-atomic refinements arise naturally in a number of settings. For example, a protocol might be specified abstractly as a single operation, but in a later development refined into a sequence of operations describing the structure of how the protocol works in more detail. Another example might be a (coffee!) machine which has an operation that requires a sequence of inputs (or generates a sequence of outputs). At the abstract level this is described as a single atomic operation, but at the concrete level we may wish to dispense with this assumption and specify the process of entering the inputs (generating the outputs) one by one.

Such non-atomic refinements have been extensively studied in the context of process algebras, usually under the name of action refinement [2]. Examples of simple non-atomic refinements are beginning to emerge for state-based specifications, however, we are not aware of any systematic study of state-based non-atomic refinement. (Although there has been some study of hiding sets of actions in action systems [7].) The purpose of this paper is to contribute to such a discussion.

The simplest approach to non-atomic refinement is to introduce a *skip* operation in the abstract specification, such an operation produces no change in the abstract state. One of the concrete operations, say COp_1, can refine AOp whilst the other refines *skip*. Examples of applications of such an approach include protocol refinements in B [1], in Z [10] and buffers in B [4]. In Section 3 of this paper we derive the relational basis for refinements of this kind and give a Z formulation for the appropriate simulation conditions.

However, not all non-atomic refinements can be verified in such a manner. Consider a refinement where we would like to split a collection of inputs or outputs across several concrete operations. Because we are transforming the inputs/outputs in this fashion, such a refinement cannot in general be verified using abstract steps of *skips*. A more complex example which illustrates some of the problems will be given in Section 4.

In Section 5 we consider how such refinements can be verified in general. The initial condition we consider decomposes an abstract operation into a sequence of concrete operations $COp_1 \fatsemi COp_2$, where no requirement is made that either of the concrete operations refines *skip*. In order to distribute an abstract operation's inputs and outputs across a sequence of concrete operations we apply current work on I/O refinement described in [3, 9], extending it where necessary to provide the required generalisation. This generalisation is derived in Section 6. The resulting refinement rules are given in Z and we show how they can be applied to the example in Section 4. In Section 7 we summarise the rules and in Section 8 we make some concluding remarks. We begin by describing the traditional view of refinement in Z based upon the standard relational semantics, throughout the paper we will work at the relational level only using the Z schema calculus to give the final refinement conditions.

2 A Relational View of Refinement in Z

In this section we discuss the relational view of refinement and describe how it treats partiality, leading to the standard presentation of refinement in a language such as Z [8, 10]. In doing so we present a summary of results in [6, 10] to which the reader is directed for more detailed explanation if necessary.

The underlying model of a state based system is a relational model, where the components of an abstract data type (ADT) are relations (assumed total for the moment). An ADT is a quadruple $\mathcal{A} = (Astate, ai, \{aop_i\}_{i \in I}, af)$ which acts on a global state space G such that: $Astate$ is the space of values; $ai \in G \leftrightarrow Astate$ is an initialisation; $af \in Astate \leftrightarrow G$ is a finalisation; aop_i are operations in $Astate \leftrightarrow Astate$.

A program P is a sequence of operations upon a data type beginning with an initialisation and ending with a finalisation, e.g.

$$P(\mathcal{A}) = ai \fatsemi aop_1 \fatsemi aop_2 \fatsemi af$$

The standard derivation of refinement assumes that the abstract and concrete data types are conformal, i.e. they have the same global state space G and that the indexing sets for the operations coincide (so every abstract operation has a concrete counterpart and vice versa).

Definition 1. *A data type $\mathcal{C}$ refines a data type $\mathcal{A}$ if, for every program P, $P(\mathcal{C}) \subseteq P(\mathcal{A})$.*

This has the effect (for total relations) of refinement being the reduction of non-determinism. This definition of refinement involves quantification over all programs, and in order to verify such refinements, *simulations* are used which consider values produced at each step of a program's execution. Simulations are thus the means to make the verification of a refinement feasible. In order to consider values produced at each step we need a relation r between the two state spaces *Astate* and *Cstate*, this relation is known as the *retrieve* relation.

Partiality

In the relational framework we have described so far the relations were assumed to be total relations. However, not all operations are total, and the traditional meaning of an operation ρ specified as a partial relation is that ρ behaves as specified when used within its precondition (domain), and outside its precondition, anything may happen.

In order to deal with this partial relations are totalised, i.e. we add a distinguished element $\perp$ to the state space, denoting undefinedness, and we denote such an augmented version of X by $X^{\perp}$. Thus if ρ is a partial relation between X and Y, we add the following sets of pairs to ρ: $\{x : X^{\perp}, y : Y^{\perp} \mid x \notin \operatorname{dom} \rho \bullet x \mapsto y\}$, and call this new (total) relation $\overset{\bullet}{\rho}$.

We also require that the retrieve relation be strict, i.e., that r propagates undefinedness and we ensure this by considering the lifted form of $r \in X \leftrightarrow Y$:

$$\overset{\circ}{r} = r \cup (\{\perp\} \times Y^{\perp})$$

The retrieve relation gives rise to two types of step by step comparisons: downwards simulation and upwards simulation [10]. These simulation relations are the basis for refinement methods in Z and other state based languages. Their usefulness lies in the fact that they are sound and jointly complete [6].

In this paper we restrict our attention to the more commonly occurring downward simulations. A downward simulation is a relation r from *Astate* to *Cstate* such that

$$\overset{\bullet}{ci} \subseteq \overset{\bullet}{ai} \, ⨾ \, \overset{\circ}{r}$$
$$\overset{\circ}{r} \, ⨾ \, \overset{\bullet}{cf} \subseteq \overset{\bullet}{af}$$
$$\overset{\circ}{r} \, ⨾ \, \overset{\bullet}{cop_i} \subseteq \overset{\bullet}{aop_i} \, ⨾ \, \overset{\circ}{r} \qquad \text{for each index } i \in I$$

The simulation rules are defined in terms of augmented relations. We can extract the underlying rules for the original partial relations as follows. For example, for a downwards simulation these rules are equivalent to the following:

$$ci \subseteq ai \, ⨾ \, r$$
$$r \, ⨾ \, cf \subseteq af$$
$$(\operatorname{dom} aop_i \lhd r \, ⨾ \, cop_i) \subseteq aop_i \, ⨾ \, r$$
$$\operatorname{ran}((\operatorname{dom} aop_i) \lhd r) \subseteq \operatorname{dom} cop_i$$

The last two conditions (where $\lhd$ is domain restriction [8]) mean that: the effect of cop_i must be consistent with that of aop_i; and, the operation cop_i is defined for every value that can be reached from the domain of aop_i using r.

Inputs and Outputs
We can use this relational semantics to model systems in which operations have input and output (IO) by providing all inputs at initialisation and delaying outputs until finalisation. To do so we augment the state by adding two sequences, an input sequence and an output sequence. Initially, the output sequence is empty; in the final state, the input sequence is empty. Every time an operation is executed, (if the operation has an input) the first value is removed from the input sequence, and (if the operation has an output) a value is added to the end of the output sequence. The outcome of the operation does not (directly) depend on any other value in the input or output sequence.

Above conditions for a downward simulation were derived for use between operations that have no inputs or outputs. We can now derive similar downward simulation conditions for operations that do have inputs and outputs, by augmenting the state with extra components representing the sequence of inputs still to be dealt with and the sequence of outputs already computed.

Let operations *aop* and *cop* consume input and produce output. Let us denote the equivalent operations that expect input and output sequences by aop_s and cop_s. It is now possible to translate the conditions for a downwards simulation between aop_s and cop_s into conditions between *aop* and *cop*. Given a relation r between states without input and output sequences, we must construct an equivalent relation that acts on the enhanced form of the state. We use the following retrieve relation on the extended state

$$r_s = r \| id[Inp] \| id[Outp]$$

where $\|$ is a relational parallel composition (see [10]) and *Inp* and *Outp* are the types of the input and output sequences of *aop*. $id[Inp]$ maps an abstract input sequence to an identical concrete input sequence, and similarly for the output (see figure 1).

Using such a retrieve relation, [10] derive equivalent simulation rules for *aop* and *cop* which are as follows (because any operation can produce output the finalisation condition is no longer required):

$$ci \subseteq ai \mathbin{⨾} r$$
$$(\mathrm{dom}\, aop \lhd (r \| id) \mathbin{⨾} cop) \subseteq aop \mathbin{⨾} (r \| id)$$
$$\mathrm{ran}((\mathrm{dom}\, aop) \lhd (r \| id)) \subseteq \mathrm{dom}\, cop$$

These rules can now be transformed from their relational setting to simulation rules for Z specifications by writing them in the Z schema calculus. This formalisation is the same as the rules given in standard presentations of refinement in Z, e.g. [8].

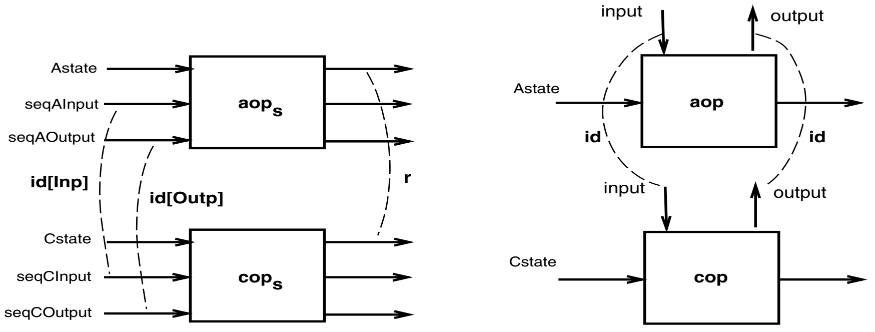


Fig. 1. Refinement of operations with input and output

Definition 1 *Let R be the retrieve relation between data types $(Astate, Ainit, \{AOp\})$ and $(Cstate, Cinit, \{COp\})$. Suppose that the operations have an input $x? : X$ and output $y! : Y$. R is a downwards simulation if the following hold.*

$$\begin{array}{l}
\forall\, Cstate \bullet CInit \Rightarrow (\exists\, Astate \bullet AInit \land R) \\
\forall\, Astate;\ Cstate;\ x? : X \bullet pre\, AOp \land R \Rightarrow pre\, COp \\
\forall\, Astate;\ Cstate;\ Cstate';\ x? : X;\ y! : Y \bullet \\
\qquad\qquad pre\, AOp \land COp \land R \Rightarrow \exists\, Astate' \bullet R' \land AOp
\end{array}$$

In the subsequent sections of this paper we will relax two assumptions made above. The starting point will be to consider the consequences of refining an abstract operation into more than one concrete operation. In doing so we will need the generality of IO refinement which assumes a general mapping between the pairs of input and output streams as opposed to the identities $id[Inp]$ and $id[Outp]$ used above.

3 Simple Non-atomic Refinement

We begin our derivation with the same definition of refinement, namely that for every program P, $P(\mathcal{C}) \subseteq P(\mathcal{A})$. Let us now suppose that in the two data types the indexes coincide except that abstract operation *aop* is refined by the sequence $cop_1;\ cop_2$. We now have two sets of potential programs, those drawn from the abstract indexes and those from the concrete indexes. Let us denote these P_A and P_C respectively. So $ai \fatsemi aop \fatsemi af$ and $ci \fatsemi cop_1 \fatsemi cop_2 \fatsemi cf$ are programs in $P_A(\mathcal{A})$ and $P_A(\mathcal{C})$ respectively, whereas $ci \fatsemi cop_2 \fatsemi cf$, $ci \fatsemi cop_1 \fatsemi cf$ and $ci \fatsemi cop_2 \fatsemi cop_1 \fatsemi cf$ are programs in $P_C(\mathcal{C})$. Thus for non-atomic refinement there are two conditions which can perhaps be considered as liveness and safety conditions respectively:

$$P_A(\mathcal{C}) \subseteq P_A(\mathcal{A}) \qquad \text{and} \qquad P_C(\mathcal{C}) \subseteq P_C(\mathcal{A})$$

The first requires that if we take abstract indexes then the equivalent concrete program reduces non-determinism, e.g. $ci \mathbin{\text{;}} cop_1 \mathbin{\text{;}} cop_2 \mathbin{\text{;}} cf \subseteq ai \mathbin{\text{;}} aop \mathbin{\text{;}} af$. The second implies that to every concrete program there must be some abstract equivalent that it refines, e.g. there will be an abstract equivalent to $ci \mathbin{\text{;}} cop_2 \mathbin{\text{;}} cf$.

To begin we consider the case when both these conditions are required. Simulations can be used to make step-by-step comparisons as before. Quantification over all abstract programs leads to the requirement that

$$\overset{\circ}{r} \mathbin{\text{;}} \overset{\bullet}{cop_1} \mathbin{\text{;}} \overset{\bullet}{cop_2} \subseteq \overset{\bullet}{aop} \mathbin{\text{;}} \overset{\circ}{r} \tag{1}$$

whilst quantification over all concrete programs requires that we find abstract counterparts to cop_1 and cop_2 which we denote p_A^1 and p_A^2 such that

$$\overset{\circ}{r} \mathbin{\text{;}} \overset{\bullet}{cop_1} \subseteq p_A^1 \mathbin{\text{;}} \overset{\circ}{r} \qquad \text{and} \qquad \overset{\circ}{r} \mathbin{\text{;}} \overset{\bullet}{cop_2} \subseteq p_A^2 \mathbin{\text{;}} \overset{\circ}{r}$$

The obvious choice for p_A^1 and p_A^2 are for one to be the original abstract operation $\overset{\bullet}{aop}$ and for the other to be $\overset{\bullet}{skip_R}$ (the subscript R will be explained in a moment). Clearly these choices are sufficient, but not necessary, however whilst it is possible to construct examples where the concrete operations are refining different abstract operations it is difficult to construct realistic examples. Thus, without loss of generality taking cop_1 to refine aop, let us consider the requirement that

$$\overset{\circ}{r} \mathbin{\text{;}} \overset{\bullet}{cop_1} \subseteq \overset{\bullet}{aop} \mathbin{\text{;}} \overset{\circ}{r} \qquad \text{and} \qquad \overset{\circ}{r} \mathbin{\text{;}} \overset{\bullet}{cop_2} \subseteq \overset{\bullet}{skip_R} \mathbin{\text{;}} \overset{\circ}{r} \tag{2}$$

The abstract operation $\overset{\bullet}{skip_R}$ can be chosen to be any operation satisfying (1) with the property that $\overset{\bullet}{aop} \mathbin{\text{;}} \overset{\bullet}{skip_R} = \overset{\bullet}{aop}$. For then if (2) holds we have

$$\overset{\circ}{r} \mathbin{\text{;}} \overset{\bullet}{cop_1} \mathbin{\text{;}} \overset{\bullet}{cop_2} \subseteq \overset{\bullet}{aop} \mathbin{\text{;}} \overset{\circ}{r} \mathbin{\text{;}} \overset{\bullet}{cop_2} \subseteq \overset{\bullet}{aop} \mathbin{\text{;}} \overset{\bullet}{skip_R} \mathbin{\text{;}} \overset{\circ}{r} = \overset{\bullet}{aop} \mathbin{\text{;}} \overset{\circ}{r}$$

Thus (2) represents sufficient conditions for the action refinement of aop into cop_1; cop_2. We can now extract the underlying conditions on the partial relations in the usual manner. The first is the standard condition for refining aop by cop_1, namely that (we elide the identities over input and output streams for the moment)

$$(\operatorname{dom} aop \lhd r \mathbin{\text{;}} cop_1) \subseteq aop \mathbin{\text{;}} r$$
$$\operatorname{ran}((\operatorname{dom} aop) \lhd r) \subseteq \operatorname{dom} cop_1$$

The requirement that $\overset{\bullet}{aop} \mathbin{\text{;}} \overset{\bullet}{skip_R} = \overset{\bullet}{aop}$ could be satisfied by $skip_R = skip$, however, this is unnecessarily restrictive and in fact we can take $skip_R = A \lhd skip$ for any A with $\operatorname{ran} aop \subseteq A$. Possible choices for $skip_R$ then range from $\operatorname{ran} aop \lhd skip$ to $skip$ itself. The second requirement in (2) is equivalent to

$$(\operatorname{dom} skip_R \lhd r \mathbin{\text{;}} cop_2) \subseteq skip_R \mathbin{\text{;}} r$$
$$\operatorname{ran}((\operatorname{dom} skip_R) \lhd r) \subseteq \operatorname{dom} cop_2$$

Taking $skip_R = \operatorname{ran} aop \lhd skip$ these become

$$\begin{array}{l}(\operatorname{ran} aop \lhd r \mathbin{;} cop_2) \subseteq \operatorname{ran} aop \lhd r \\ \operatorname{ran}(\operatorname{ran} aop \lhd r) \subseteq \operatorname{dom} cop_2\end{array}$$

and when $skip_R = skip$ they are: $r \mathbin{;} cop_2 \subseteq r$ and $\operatorname{ran} r \subseteq \operatorname{dom} cop_2$.

These can be translated into Z in the usual manner. It is in this context that the non-atomic refinements given in [10, 4] are verified.

For example, in [4] a specification is given of an unordered buffer together with a refinement of it. The refinement introduces an additional operation, *mid*, which is a refinement of *skip* at the abstract level.

However, some desirable non-atomic refinements are more complex than this, and we illustrate the problem with an example which will motivate our need for more general refinement conditions.

4 Example - A Bank Account

We specify a bank consisting of a number of electronic booths where users may deposit money and check their balances. At an abstract level we are given a mapping from names to Money ($= \mathbb{N}$), and operations allowing money to be deposited and balances checked. The example illustrates nicely many of the issues involved in non-atomic refinement⋆.

$$\begin{array}{l} ABank \mathrel{\widehat=} [act : Name \pfun Money] \\ ABankInit \mathrel{\widehat=} [ABank' \mid act' = \varnothing] \\ AOpenAcct \mathrel{\widehat=} [\Delta ABank;\ n? : Name \mid act' = act \oplus \{n? \mapsto 0\}] \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{\mathit{Deposit}} \\ \hline \Delta ABank \\ n? : Name \\ p? : Money \\ \hline n? \in \operatorname{dom} act \\ act' = act \oplus \{n? \mapsto act(n?) + p?\} \\ \hline \end{array}
\qquad
\begin{array}{|l} \multicolumn{1}{l}{\mathit{Balance}} \\ \hline \Xi ABank \\ n? : Name \\ b! : Money \\ \hline n? \in \operatorname{dom} act \\ b! = act\ n? \\ \hline \end{array}$$

At the concrete level an atomic *Deposit* operation is unrealistic and we would like the amounts to be transferred coin by coin at every booth thus allowing interleaving of these operations with actions at other booths, where $Coin = \{1, 2, 5, 10\}$ say. To specify this we use a collection of temporary accounts *tct* and split the *Deposit* operation into a transaction consisting of a *Start*, a succession of *Next* operations transferring the amount coin by coin with a *Stop* operation ending the process. A temporary account is now represented by sequences of coins. The *Stop* operation takes this sequence and sums the coins entered, updating the concrete account with the result of this calculation

⋆ and is adapted from an example in [10] which specifies a distributed file store.

(remember that $+/.$ represents distributed summation over a sequence). The concrete specification is as follows, where $\ntriangleleft$ is domain subtraction.

$$\begin{array}{|l}
\hline \textit{CBank} \\
cct : Name \pfun Money \\
tct : Name \pfun \operatorname{seq} Coin \\
\hline
\operatorname{dom} tct \subseteq \operatorname{dom} cct \\
\hline
\end{array}$$

$$CBankInit \mathrel{\widehat=} [CBank' \mid cct' = tct' = \varnothing]$$

$$\begin{array}{|l}
\hline \textit{Start} \\
\Delta CBank \\
n? : Name \\
\hline
n? \in \operatorname{dom} cct \\
tct' = tct \oplus \{n? \mapsto \langle\,\rangle\} \\
cct' = cct \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \textit{Next} \\
\Delta CBank \\
n? : Name \\
c? : Coin \\
\hline
n? \in \operatorname{dom} tct \\
tct' = tct \oplus \{n? \mapsto (tct\ n?) \frown \langle c? \rangle\} \\
cct' = cct \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \textit{Stop} \\
\Delta CBank \\
n? : Name \\
\hline
n? \in \operatorname{dom} tct \\
tct' = \{n?\} \ntriangleleft tct \\
cct' = cct \oplus \\
\quad \{n? \mapsto cct(n?) + (+/.(tct\ n?))\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \textit{Balance} \\
\Xi CBank \\
n? : Name \\
b! : Money \\
\hline
n? \in \operatorname{dom} cct \\
b! = cct\ n? \\
\hline
\end{array}$$

The link between the abstract and concrete state spaces will be via the relation R

$$\begin{array}{|l}
\hline \textit{R} \\
ABank \\
CBank \\
\hline
act = cct \\
\hline
\end{array}$$

Clearly at some level the abstract *Deposit* operation is being refined by the sequence $Start \fatsemi Next \ldots Next \fatsemi Stop$. However, the refinement isn't simply a matter of one of the concrete operations corresponding to *Deposit* whilst the others correspond to *skip*.

At issue is the following. The retrieve relation links *act* and *cct*, therefore abstract *skip* operations can be refined by concrete operations which only change the temporary account *tct*. Therefore *Start* and *Next* look suitable candidates to refine *skip*. There are however two problems. The first is that although *Start* and *Next* do not alter *cct* they do consume input, conceptually taking values off the input stream. Therefore at the level of an augmented state complete with input stream they do not simply correspond to *skip*. The second, and related, problem is that if *Stop* corresponds to *Deposit* then $\mathrm{pre}\, Deposit \wedge R$ needs to imply the precondition of *Stop*. However, the precondition of *Stop* is that $n? \in \mathrm{dom}\, tct$, which isn't a consequence of $\mathrm{pre}\, Deposit \wedge R$. The issue is that $n? \in \mathrm{dom}\, tct$ is assuming that at least a *Start* operation has already happened, and that the system is now ready to *Stop*. *Stop* can in fact be amended to overcome this problem. However to do this you need to put sufficient functionality into it that the other concrete operations are then unnecessary.

As we can see there are many issues involved in such a refinement, not least is the problem that the inputs of *Deposit* are distributed throughout the concrete operations, this means that we must develop machinery in addition to that discussed in the last section. This is what we seek to do next.

5 General Non-atomic Refinement

In this section we will consider more general refinements than considered in Section 3, in particular we drop the requirement that $P_C(\mathcal{C}) \subseteq P_C(\mathcal{A})$. This means that we can consider decomposing an abstract operation into a sequence of concrete operations without requiring that any of these concrete operations refine an abstract operation of *skip*. This opens the way to providing methods of refinement that can tackle some of the issues highlighted in the previous section. In this section we also consider various properties of non-atomic refinement. In particular, we show that non-atomic refinement is transitive and we consider conditions on the concrete operations that will allow interleaving of the components of a non-atomic decomposition.

To verify a general non-atomic refinement we must also address in some detail how we treat inputs and outputs. The bank account example is particularly interesting in this respect because it has taken an input amount $p? : Money$ and broken it down into a single input $c? : Coin$ provided a number of times via the *Next* operation. To verify such refinements we will use the technique of IO-refinement and to apply it we extend current work in this area [3, 9]. These points are discussed in Section 6, we begin now with the general conditions for a non-atomic refinement.

5.1 Conditions for a Non-atomic Refinement

We begin by dropping the safety requirement that $P_C(\mathcal{C}) \subseteq P_C(\mathcal{A})$, so in particular the requirements of (2) disappear and the single requirement is that:

$$\overset{\circ}{r} \fatsemi \overset{\bullet}{cop_1} \fatsemi \overset{\bullet}{cop_2} \subseteq \overset{\bullet}{aop} \fatsemi \overset{\circ}{r} \qquad (3)$$

With this single requirement we can extract the underlying conditions on the partial relations as before to find that this is equivalent to three conditions, namely that

$$(\mathrm{dom}\, aop \lhd r \mathbin{⨾} cop_1 \mathbin{⨾} cop_2) \subseteq aop \mathbin{⨾} r \tag{4}$$

$$\mathrm{ran}((\mathrm{dom}\, aop) \lhd r) \subseteq \mathrm{dom}\, cop_1 \tag{5}$$

$$\mathrm{ran}((\mathrm{dom}\, aop) \lhd r \mathbin{⨾} cop_1) \subseteq \mathrm{dom}\, cop_2 \tag{6}$$

If cop_1 is deterministic we can replace the last two (applicability) conditions by a single condition.

Proposition 1 *If cop_1 is deterministic then*

$$ran((dom\, aop) \lhd r) \subseteq dom\, cop_1 \quad \wedge$$
$$ran((dom\, aop) \lhd r \mathbin{⨾} cop_1) \subseteq dom\, cop_2$$

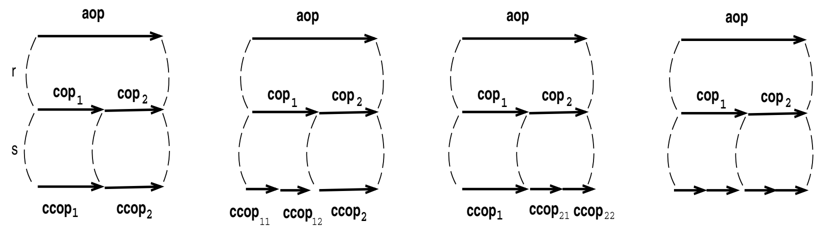
is equivalent to the condition $ran((dom\, aop) \lhd r) \subseteq dom(cop_1 \mathbin{⨾} cop_2)$.

The requirement of cop_1 being deterministic is necessary to ensure that the resultant condition implies $\mathrm{ran}((\mathrm{dom}\, aop) \lhd r \mathbin{⨾} cop_1) \subseteq \mathrm{dom}\, cop_2$, the other implications always hold.

Before we proceed any further it is important to check whether non-atomic refinement is transitive, that is further non-atomic or atomic refinements should give rise to an overall refinement. This is indeed the case.

Theorem 1. *Non-atomic refinement is transitive.*

Proof. There are four cases to consider which are illustrated in the following diagram.



In each case it is easy to see that we have transitivity. □

Without considering any input and output transformations at this stage we can express the relational conditions given in (4-6) in the Z schema calculus. The formulation is as follows.

Definition 2 *R is a non-atomic downwards simulation if the following hold.*

$$\forall\, Astate;\ Cstate;\ Cstate' \bullet$$
$$\qquad pre\, AOp \wedge (COp_1 \mathbin{⨾} COp_2) \wedge R \Rightarrow \exists\, Astate' \bullet R' \wedge AOp$$
$$\forall\, Astate;\ Cstate \bullet pre\, AOp \wedge R \Rightarrow pre\, COp_1$$
$$\forall\, Astate;\ Cstate \bullet pre\, AOp \wedge R \wedge COp_1 \Rightarrow pre\, COp_2$$

These conditions generalise to a non-atomic refinement with an arbitrary number of abstract and concrete operations in the obvious manner.

Let us consider the bank account example when the deposit $p?$ consists of a single coin. We then have three operations $Start \fatsemi Next \fatsemi Stop$ in our concrete refinement (we will consider an arbitrary amount deposited later when we have a way to transform inputs). To verify such a refinement we have to demonstrate four conditions:

$$\begin{array}{l}
\forall\, Astate;\ Cstate;\ Cstate' \bullet \\
\qquad\qquad \mathrm{pre}\, Deposit \land (Start \fatsemi Next \fatsemi Stop) \land R \Rightarrow \exists\, Astate' \bullet R' \land Deposit \\
\forall\, Astate;\ Cstate \bullet \mathrm{pre}\, Deposit \land R \Rightarrow \mathrm{pre}\, Start \\
\forall\, Astate;\ Cstate \bullet \mathrm{pre}\, Deposit \land R \land Start \Rightarrow \mathrm{pre}\, Next \\
\forall\, Astate;\ Cstate \bullet \mathrm{pre}\, Deposit \land R \land (Start \fatsemi Next) \Rightarrow \mathrm{pre}\, Stop
\end{array}$$

We will consider the three applicability conditions first. The predicate of $\mathrm{pre}\, Deposit \land R$ will be the condition that $n? \in \mathrm{dom}\, cct$, which is the precondition of *Start*. Similarly $\mathrm{pre}\, Deposit \land R \land Start$ implies $n? \in \mathrm{dom}\, tct$ which is the precondition of *Next*. The precondition of *Stop* works in a similar way. Thus even without IO transformations the applicability conditions can be verified.

The correctness condition requires that we calculate the schema composition $(Start \fatsemi Next \fatsemi Stop)$ which results in

$$\begin{array}{|l}
\hline
\Delta CBank \\
n? : Name \\
c? : Coin \\
\hline
n? \in \mathrm{dom}\, cct \\
tct' = \{n?\} \ndres tct \\
cct' = cct \oplus \{n? \mapsto cct(n?) + c?\} \\
\hline
\end{array}$$

Given a very simple input transformation of a deposit $p?$ into a single coin this can be seen to satisfy (at an intuitive level) the criteria for decomposing *Deposit* into these three operations as long as we assume inputs correspond to a single coin. In the next section we will see how this intuition can be formalised and how we can verify the general case of an arbitrarily large deposit.

6 Input and Output Transformations

In this section we consider the input and output transformations that are needed to support non-atomic refinements. We begin with a discussion of IO refinement which generalises the standard refinement conditions by allowing inputs and outputs to alter under refinement. We apply this work to non-atomic refinement in Section 6.2 resulting in a set of conditions that allow inputs and outputs to be distributed throughout a concrete decomposition.

To understand the issues let us consider our running example again. In order to verify a refinement we have to prove a correctness condition between *Deposit* and the concrete decomposition. At the end of the previous section we considered the case when the input deposit was composed of a single coin, and we calculated the schema composition $(Start \fatsemi Next \fatsemi Stop)$ to verify the correctness criteria.

Even at this point there is an issue to consider, for this composition has an input $c? : Coin$ whereas *Deposit* has an input $p? : Money$. Although at an intuitive level we can see the correspondence between these schemas, a strict interpretation of standard refinement does not allow the inputs and outputs or their types to be changed[**]. This is a direct consequence of the use of the identities $id[Inp]$ and $id[Outp]$ in the retrieve relation

$$r_s = r \| id[Inp] \| id[Outp]$$

discussed in section 2. These identities map abstract input and output sequences to identical concrete input and output sequences, because they are identical, the types of the input and output cannot change.

6.1 IO Refinement

Recent work on IO refinement [3, 9] has tackled this issue, and provides a solution to this problem by generalising the retrieve relation r_s. Here we follow the formalisation of [3] although [9] provides an alternative characterisation.

IO refinement is a generalisation of standard (atomic) refinement. Let us consider the refinement of an abstract operation *aop* into a concrete one *cop*. Suppose further that r is the retrieve relation which links the abstract and concrete state spaces. In order to allow the types of inputs and outputs to change IO refinement replaces the identities with arbitrary relations *it* and *ot* between the input and output elements respectively. Thus *it* and *ot* are essentially retrieve relations between the inputs and outputs, hence allowing these to change under a refinement in a similar way to changing the state space. The full retrieve relation r_s between the enhanced state is then

$$r_s = r \| it^* \| ot^*$$

where it^* applies *it* along each element in the input sequence.

It is necessary to impose some conditions on *it* and *ot*. The first is that for r_s not to exclude combinations of states in r, we need to require that *it* and *ot* are total on the abstract input and output types. Secondly, *ot* must be injective. This condition guarantees that different abstract ("original") outputs can be distinguished in the concrete case because their concrete representations will be different as well.

[**] The file store example given in [10] contains another example of such a transformation where an input file is decomposed into a sequence of bytes.

The conditions for an IO refinement between aop_s and cop_s can be given an equivalent formulation in terms of aop and cop (see figure 2):

$$\operatorname{dom} aop \lhd ((r \| it) \mathbin{⨟} cop) \subseteq aop \mathbin{⨟} (r \| ot)$$
$$\operatorname{ran}(\operatorname{dom} aop \lhd (r \| it)) \subseteq \operatorname{dom} cop$$

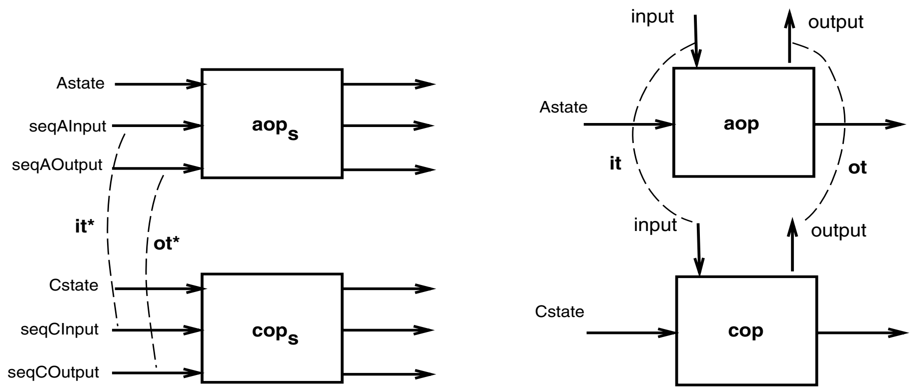


Fig. 2. IO refinement of operations

These conditions can be expressed as conditions on Z schemas as follows. The relations *it* and *ot* between the inputs and outputs become schemas called *input and output transformers*. An input transformer for a schema is an operation whose outputs exactly match the schema's inputs, and whose signature is made up of input- and output components only; similarly for output transformers. These are applied to the abstract and concrete operations using piping ($\gg$).

With these notions in place we can re-phrase the conditions of IO refinement in the Z schema calculus. We use an overlining operator, which extends componentwise to signatures and schemas: $\overline{x?} = x!, \overline{x!} = x?$. Thus $\overline{IT}$ denotes the schema where all inputs become outputs with the same basename, and all outputs inputs.

Definition 3 *Let IT be an input transformer for COp which is total on the abstract inputs. Let OT be a total injective output transformer for AOp. The retrieve relation R defines an IO refinement if:*

applicability $\forall Astate; \ Cstate \bullet pre(\overline{IT} \gg AOp) \land R \Rightarrow pre\, COp$

correctness *wherever AOp is defined, COp with the input transformation should produce a result related by R and the output transformation to one that AOp could have produced:*

$$\forall Astate; \ Cstate; \ Cstate' \bullet \\ pre\, AOp \land R \land (IT \gg COp) \Rightarrow \exists Astate' \bullet R' \land (AOp \gg OT)$$

IO refinement allows inputs and outputs to be refined in a controlled manner. Controlled because since inputs and outputs are observable we must be able to reconstruct the original behaviour from a concrete refinement. This reconstruction is achieved by using the input and output transformers which essentially act as wrappers to a concrete operation, converting abstract inputs to concrete ones and similarly for the output. Hayes and Sanders [5] use piping in much the same way: to represent the equivalent of relational composition for inputs and outputs in Z schemas. They use the term "representation schema" for what we call "transformers".

We can apply these ideas to our example. The input transformer schema that we need to use is thus given by

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ IT \_\_} \\
p? : Money \\
c! : Coin \\
n?, n! : Name \\
\hline
c! = p? \land n! = n? \\
\hline
\end{array}
$$

Here $c!$ is an output so that it matches the input $c?$ of the composition $(Start \mathbin{;} Next \mathbin{;} Stop)$, and no changes are made to the name so that is passed through unchanged. There are no outputs so the output transformer is the identity. With this in place it is easy to see that we have the correct transformations in place to deal with the change of input when each input $p?$ is entered as a sequence consisting of one single coin, and we can verify the condition

$$\mathrm{pre}\, Deposit \land (IT \gg Start \mathbin{;} Next \mathbin{;} Stop) \land R \Rightarrow \exists\, Astate' \bullet R' \land Deposit$$

However, in reality deposits can be arbitrarily large (i.e. not provided by a single coin), and to deal with this we need further generalisations. The next subsection considers how to do this by integrating IO refinement into the non-atomic refinement conditions we have already derived.

6.2 General IO Transformations

Consider the case when the input deposit is given as two coins. We will now have to verify a correctness condition between *Deposit* and the composition $(Start \mathbin{;} Next \mathbin{;} Next \mathbin{;} Stop)$ to show that the non-atomic refinement holds. However, if we calculate this composition we result in

$$
\begin{array}{|l}
\hline
\Delta CBank \\
n? : Name \\
c? : Coin \\
\hline
n? \in \mathrm{dom}\, cct \\
tct' = \{n?\} \mathbin{\lhd} tct \\
cct' = cct \oplus \{n? \mapsto cct(n?) + c? + c?\} \\
\hline
\end{array}
$$

We have lost the differentiation needed between the inputs of distinct applications of the *Next* operation. Furthermore, our input transformation is now not just between two operations, but a whole sequence of concrete operations, the length of which is only determined by the input $p?$ (the number of *Next* operations needed is in fact determined by the coins used as long as they sum to the correct amount $p?$), and this can continually vary.

To deal with this we will generalise IO refinement in the following way. IO refinement was derived as a condition between one abstract and one concrete operation, because of that a simple element by element mapping *it* sufficed. In our world of non-atomic refinement we wish to decompose one abstract operation into a sequence of concrete operations. Therefore we need a mapping between an abstract input and a sequence of concrete inputs representing the inputs needed in the decomposition. We thus replace the maps *it* and *ot* by r_{in} and r_{out} where

$$r_{in} : Ainput \longleftrightarrow \mathrm{seq}\, Cinput$$
$$r_{out} : Aoutput \longleftrightarrow \mathrm{seq}\, Coutput$$

and r_{in} is total on *Ainput*, and r_{out} is total on seq *Coutput*. For example, suppose that an amount $p?$ is entered as the sequence of coins $\langle c_1?, \ldots, c_m? \rangle$, then an abstract input $(n?, p?)$ for the *Deposit* operation will be mapped to the input sequence $\langle n?, (n?, c_1?), \ldots, (n?, c_m?), n? \rangle$ to be consumed by $(Start \fatsemi Next, \ldots, Next \fatsemi Stop)$.

Given a decomposition of *aop* into $cop_1 \fatsemi cop_2$ let us denote operations acting on the augmented state space be denoted by, as before, aop_s, cop_{1s} and cop_{2s}. With mappings r_{in} and r_{out} describing how the inputs and outputs of *aop* are turned into those for cop_1 and cop_2, and a retrieve relation r between the state spaces, the retrieve relation r_s on the augmented state will be given by

$$r_s = r \,\|\, {}^\frown/.r_{in}^* \,\|\, {}^\frown/.r_{out}^*$$

Here ${}^\frown/.r_{in}^*$ takes an input sequence seq *Ainput* and creates a concrete input sequence by concatenating together the effect of r_{in} for each item in seq *Ainput*. If there are two concrete operations in the refinement, then r_{in} maps each abstract input into a pair of concrete inputs, the first for consumption by cop_1 the second for cop_2 (see figure 3).

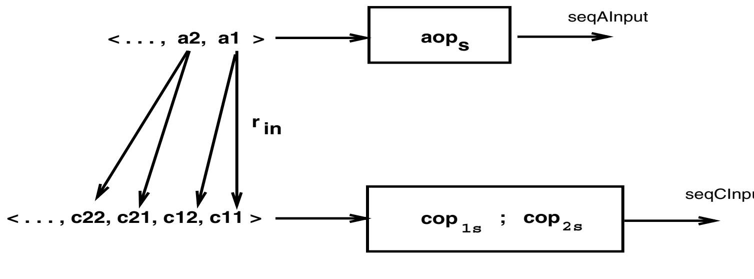


Fig. 3. Splitting the abstract input

We can now take the three non-atomic refinement conditions described in terms of an augmented state:

$$(\mathrm{dom}\, aop_s \lhd r_s \fatsemi cop_{1s} \fatsemi cop_{2s}) \subseteq aop_s \fatsemi r_s \tag{7}$$
$$\mathrm{ran}((\mathrm{dom}\, aop_s) \lhd r_s) \subseteq \mathrm{dom}\, cop_{1s} \tag{8}$$
$$\mathrm{ran}((\mathrm{dom}\, aop_s) \lhd r_s \fatsemi cop_{1s}) \subseteq \mathrm{dom}\, cop_{2s} \tag{9}$$

and turn these into equivalent conditions on the operations with input and output at each step: aop, cop_1 and cop_2 in the usual way. It is easy to see that they become:

$$\mathrm{dom}\, aop \lhd (r \| r_{in}) \fatsemi (id \| cop_1) \fatsemi (cop_2 \| id) \subseteq aop \fatsemi (r \| r_{out}) \tag{10}$$
$$\mathrm{ran}(\mathrm{dom}\, aop \lhd (r \| r_{in})) \subseteq \mathrm{dom}\, cop_1 \tag{11}$$
$$\mathrm{ran}(\mathrm{dom}\, aop \lhd (r \| r_{in}) \fatsemi (id \| cop_1)) \subseteq \mathrm{dom}\, cop_2 \tag{12}$$

where again we require that r_{out}, like ot, is injective.

In the formalisation of these conditions we need to write $(id \| cop_1)$ and $(cop_2 \| id)$ because a single abstract input has become a pair of concrete inputs, one for cop_1 and one for cop_2. In order to correctly select its input we need to write $(id \| cop_1)$ and $(cop_2 \| id)$ in the relational formalisation. These manipulations will appear in a different form when we express these conditions in the Z schema calculus.

To illustrate how this is done let us return for the moment to our example. For an arbitrary large deposit the input transformer IT is something like

$$\begin{array}{|l}
\hline
\quad IT \\
\hline
p? : Money \\
c! : \mathrm{seq}\, Coin \\
n?, n! : Name \\
\hline
+/.(c!) = p? \land n! = n? \\
\hline
\end{array}$$

where now we will output the deposit as a sequence of coins $c!$. However, we need to represent one more bit of information, namely that expressed in $(id \| cop_1)$ which says the concrete operations take the transformed input one at a time. Let us suppose a deposit comprises m coins. Then the cleanest way to express this is to observe that $c! = \langle c_1, \ldots, c_m \rangle$, and describe the process explicitly as substitutions in the operations, i.e. as $(Start \fatsemi Next[c_1/c?] \fatsemi \ldots \fatsemi Next[c_m/c?] \fatsemi Stop)$. With this in place we can express the refinement conditions that have to be verified, e.g. we require

$$\begin{aligned}
&\mathrm{pre}\, Deposit \land (IT \gg Start \fatsemi Next[c_1/c?] \fatsemi \ldots \fatsemi Next[c_m/c?] \fatsemi Stop) \land R \Rightarrow \\
&\qquad \exists\, Astate' \bullet R' \land Deposit
\end{aligned}$$

The general formalisation in Z effectively combines our three conditions needed for a non-atomic refinement of AOp into $COp_1 \fatsemi COp_2$ with the use of input and output transformers from IO refinement. Explicit substitutions (as in the *Next* operation) are only necessary when the decomposition of AOp involves more than one occurrence of the same concrete operation. If COp_1 and COp_2 are distinct operations then the formalisation is the following:

Definition 4 *Non-atomic refinement with IO transformations*
Let IT be an input transformer for $COp_1 \fatsemi COp_2$ which is total on the abstract inputs. Let OT be a total injective output transformer for AOp. The retrieve relation R defines a non-atomic IO refinement if:

$$\begin{array}{l}
\forall Astate;\ Cstate;\ Cstate' \bullet \\
\qquad pre\, AOp \wedge (IT \gg COp_1 \fatsemi COp_2) \wedge R \Rightarrow \exists Astate' \bullet R' \wedge (AOp \gg OT) \\
\forall Astate;\ Cstate \bullet pre(\overline{IT} \gg AOp) \wedge R \Rightarrow pre\, COp_1 \\
\forall Astate;\ Cstate \bullet pre(\overline{IT} \gg AOp) \wedge R \wedge (IT \gg COp_1) \Rightarrow pre\, COp_2
\end{array}$$

If COp_1 and COp_2 are not distinct (e.g. two *Next* operations) then explicit substitutions are needed to control the inputs and outputs together with a predicate in the input transformer describing which operation receives which input.

Finally consider the situation where deposits can be arbitrary large. Now we do not know the number of operations in the concrete decomposition at the outset, and we have to describe it as follows. Given the abstract *Deposit* operation we use the same input transformer IT as before and decompose *Deposit* into the sequence

$$Start \fatsemi (\fatsemi/\{(i, Next[c!.i/c?]) \mid i \in \mathrm{dom}\, c!\}) \fatsemi Stop$$

Here $\fatsemi/$ denotes distributed schema composition along the sequence $\langle Next[c!.1/c?], \ldots, Next[c!.m/c?]\rangle$ where $m = \#c!$. This expression produces a schema composition of the correct number of *Next* operations according to the size of $c!$ as required ($c!$ can be any sequence that adds up to the correct amount). We can calculate such a schema composition, and it is easy to see that all the conditions for a non-atomic refinement are met.

7 Summary

It is worth summarising the criteria for non-atomic refinement as we have derived them gradually throughout the paper. In this summary we do not mention the initialisation condition which is identical to that of standard refinement. Let AOp be decomposed into the sequence $COp_1 \fatsemi COp_2$.

Simple non-atomic refinement requires that one of the concrete operations (COp_1 say) refines AOp and the other refines a restricted *skip*. The requirements on COp_1 refining AOp are the standard ones whilst those on COp_2 are that for some abstract state A with $\mathrm{ran}\, AOp \Rightarrow A$ we have

$$\begin{array}{l}
\forall Astate;\ Cstate;\ A \bullet A \wedge R \Rightarrow \mathrm{pre}\, COp_2 \\
\forall Astate;\ Cstate;\ Cstate';\ A \bullet A \wedge R \wedge COp_2 \Rightarrow \exists A' \bullet \Xi A \wedge R'
\end{array}$$

For a general non-atomic refinement where we drop the requirement that concrete operations directly refine abstract counterparts we have three basic conditions. They are:

$$\begin{array}{l}\forall\, Astate;\ Cstate;\ Cstate' \bullet \\ \qquad\qquad \mathrm{pre}\, AOp \wedge (COp_1 \mathbin{⨾} COp_2) \wedge R \Rightarrow \exists\, Astate' \bullet R' \wedge AOp \\ \forall\, Astate;\ Cstate \bullet \mathrm{pre}\, AOp \wedge R \Rightarrow \mathrm{pre}\, COp_1 \\ \forall\, Astate;\ Cstate \bullet \mathrm{pre}\, AOp \wedge R \wedge COp_1 \Rightarrow \mathrm{pre}\, COp_2\end{array}$$

These conditions do not allow any input or output transformations. If we require abstract inputs and outputs to be distributed over the concrete operations it is necessary to use input and output transformers IT and OT such that:

$$\begin{array}{l}\forall\, Astate;\ Cstate;\ Cstate' \bullet \\ \qquad \mathrm{pre}\, AOp \wedge (IT \gg COp_1 \mathbin{⨾} COp_2) \wedge R \Rightarrow \exists\, Astate' \bullet R' \wedge (AOp \gg OT) \\ \forall\, Astate;\ Cstate \bullet \mathrm{pre}(\overline{IT} \gg AOp) \wedge R \Rightarrow \mathrm{pre}\, COp_1 \\ \forall\, Astate;\ Cstate \bullet \mathrm{pre}(\overline{IT} \gg AOp) \wedge R \wedge (IT \gg COp_1) \Rightarrow \mathrm{pre}\, COp_2\end{array}$$

where IT is total on the abstract inputs and is an input transformer for $COp_1 \mathbin{⨾} COp_2$ and OT is a total and injective output transformer for AOp.

If the concrete decomposition involves more than one occurrence of the same concrete operation (as in *Next* above), then it may be necessary to use explicit schema substitutions for the input names in this last formalisation.

8 Conclusions

In this paper we have presented the beginnings of a study of state based non-atomic refinement. This led to a number of conditions for such a refinement given in the summary above. All of these conditions are derived from the basic definition of refinement as the reduction of non-determinism. Differences between the sets of conditions arise firstly from whether we require both $P_A(\mathcal{C}) \subseteq P_A(\mathcal{A})$ and $P_C(\mathcal{C}) \subseteq P_C(\mathcal{A})$ to hold, or just the former.

Considering just the former allowed us to consider how abstract inputs and outputs could be distributed over the sequence of concrete operations. To do so we applied the theory of IO refinement which extends standard refinement by allowing the retrieve relation to be extended to input and output types in addition to relating the state spaces.

The result is three sets of conditions. The first can be used when one of the concrete operations refines *skip* and the other refines the original abstract operation. The second defines conditions for a general decomposition into a number of concrete operations where the inputs and outputs are not altered. The third used IO transformers to relax this last condition. Although the use of IO transformers looks at first sight complex, they are merely wrappers which explain how an abstract input (or output) gets turned into its concrete counterpart. Although our illustrative example concentrated on input transformations, similar transformations are feasible for the outputs of an operation.

Further work to be done in this area includes looking at the relationship between upward simulations and non-atomic refinement, where we expect similar rules could be developed. It would also be useful to develop syntactic support for non-atomic refinement. For example, if an abstract operation is specified at the abstract level as $AOp \mathrel{\widehat=} AOp_1 \mathbin{⨾} AOp_2$ under what conditions is $AOp_1 \mathbin{⨾} AOp_2$ a non-atomic refinement of AOp.

References

[1] Jean-Raymond Abrial and Louis Mussat. Specification and design of a transmission protocol by successive refinements using B. In Manfred Broy and Birgit Schieder, editors, *Mathematical Methods in Program Development*, volume 158 of *NATO ASI Series F: Computer and Systems Sciences*, pages 129–200. Springer, 1997.

[2] L. Aceto. *Action refinement in process algebras.* CUP, London, 1992.

[3] E.A. Boiten and J. Derrick. IO - refinement in Z. In *3rd BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer Verlag, September 1998.

[4] M. Butler. An approach to the design of distributed systems with B AMN. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: The Z formal specification notation*, LNCS 1212, pages 223–241, Reading, April 1997. Springer-Verlag.

[5] I.J. Hayes and J.W. Sanders. Specification by interface separation. *Formal Aspects of Computing*, 7(4):430–439, 1995.

[6] 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.

[7] J. Sinclair and J. Woodcock. Event refinement in state-based concurrent systems. *Formal Aspects of Computing*, 7:266–288, 1995.

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

[9] S. Stepney, D. Cooper, and J. C. P. Woodcock. More powerful data refinement in Z. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *LNCS*, pages 284–307. Springer-Verlag, 1998.

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