IO-refinement in Z

Eerke Boiten and John Derrick
Computing Laboratory, University of Kent
Canterbury, Kent, U.K.*

Abstract

We present a generalisation of data refinement in Z, called *IO-refinement*, that allows changes in input and output parameters of operations. Several informal motivations for the desirability of such a refinement relation are given, followed by a formal derivation that demonstrates its theoretical soundness. It is proved that IO-refinement indeed generalizes data refinement. Several theorems are presented that give sufficient conditions for IO-refinement to hold in simpler situations, e.g. just adding inputs and outputs. Some examples of the use of IO-refinement are also given.

Keywords: data refinement, Z, interface refinement, input/output

1 Introduction

Data refinement is a well-established technique for transforming formal specifications in an abstract data type style into ones which are perceived to be closer to an eventual implementation. In particular, for the formal specification notation Z [16, 18] rules for data refinement have been given. These rules also allow for the operations of an abstract data type to have input and output parameters. However, they allow only very limited changes to these parameters in refinement. (In fact, most representations of these rules suggest no change is possible at all. See for an example of what *is* allowed Section 3.1 of this paper.)

Our particular interest in Z refinement is for its use as a semantics for partial specification in Z, as we will explain in more detail in Section 2.1. In brief, a collection of ("partial") specifications is satisfied by any (full) specification which is a *development* of each of them. Traditional data refinement in Z comes very close to having all desirable properties for such a development relation. One problem, however, is that it forces us to specify all possible inputs and outputs of every operation in each partial specification. This does not seem very convenient, and thus we have looked for a generalisation of Z data refinement which allows a little more freedom in this respect. Such a generalisation (called *IO-refinement*) is discussed in this paper, and we expect its relevance goes beyond providing semantics to partial specification in Z.

The next section extensively discusses possible motivations for allowing change of IO-parameters, by giving various answers to the question "Why IO-refinement?". The last of these is that it is possible to generalise the derivation of Z refinement given in the most authoritative account of Z refinement [18]. Section 3 contains this generalised derivation. In subsequent sections, a definition of IO-refinement is given, based on this derivation. We show that it generalises ordinary Z refinement. The conditions for IO-refinement turn out to be very general, and as a consequence of that, fairly hard to use. For this reason, we also present rules for using IO-refinement in a restricted way which are much easier to use. Also, some examples of its use are presented.

Before we move on to answering "Why IO-refinement?", let us recall the conditions for data refinement as given in [16], and present *the* traditional Z example, so some of the answers can refer to that.

*This work was partially funded by EPSRC under grant number GR/K13035. More information about our work can be found at `http://www.cs.ukc.ac.uk/people/staff/eab2/refine/`.

1.1 Traditional Data Refinement: Conditions

Given abstract data types A=($AState$,$AOps$,$AInit$) and C=($CState$,$COps$,$CInit$), then C is a (forwards simulation[1]) data refinement of A if there exists an abstraction schema

Abs

$AState; CState$

$Pred$

such that the following conditions hold:

initialisation $\forall CState \bullet CInit \Rightarrow (\exists AState \bullet AInit \land Abs)$

and the operations in $AOps$ and $COps$ can be matched in pairs AOp,COp both with input $x? : X$ and output $y! : Y$, such that for each of those pairs the following two conditions hold:

applicability COp should be defined on all representatives of $AState$ on which AOp is defined:

$$\forall AState ; CState ; x? : X \bullet \text{pre}\, AOp \land Abs \Rightarrow \text{pre}\, COp$$

correctness wherever AOp is defined, COp should produce a result related by Abs to one that AOp could have produced:

$$\forall AState ; CState ; CState' ; x? : X ; y! : Y \bullet$$
$$\text{pre}\, AOp \land COp \land Abs \Rightarrow \exists AState' \bullet Abs' \land AOp$$

The abstraction schema is often given as an additional implicit parameter of the data refinement relation, and is sometimes called a *retrieve* relation; the applicability condition is sometimes called *termination*. If the concrete and abstract state spaces are identical, the latter two conditions reduce to

applicability COp should be defined whenever AOp is defined:

$$\forall AState ; x? : X \bullet \text{pre}\, AOp \Rightarrow \text{pre}\, COp$$

correctness wherever AOp is defined, COp should produce a result that AOp could have produced:

$$\forall AState ; x? : X ; y! : Y \bullet \text{pre}\, AOp \land COp \Rightarrow AOp$$

and we call this *operation* refinement.

1.2 Infamous Example

Every area in formal methods has its canonical examples. For Z, it is not the factorial function, nor the alternating bit protocol. Any new technique for Z should preferably be illustrated with Spivey's $BirthdayBook$ example [16], which is

$[NAME, DATE]$

$BirthdayBook$

$known : \mathbb{P}\, NAME$
$birthday : NAME \pfun DATE$

$known = \text{dom}\, birthday$

$AddBirthday$

$\Delta BirthdayBook$
$name? : NAME$
$date? : DATE$

$name? \notin known$
$birthday' = birthday \cup \{name? \mapsto date?\}$

[1] Not all valid refinements can be proved using forwards simulation, in the most general case one needs in addition the rules for *backwards* simulation (cf. [12]). Most presentations of refinement in Z including this one concentrate on forwards simulation only ([18] being an exception). We suspect however that the solution may not lie in giving additional rules for backwards simulation, but rather that a definition of action refinement in Z will encompass both.

In a section of the book called "Strengthening the specification" Spivey describes how an indication of success or failure can be added to this:

$$REPORT ::= ok \mid \textit{already-known} \mid \textit{not-known}$$

$$\begin{array}{|l} Success \\ \hline result! : REPORT \\ \hline result! = ok \\ \hline \end{array}$$

$$\begin{array}{|l} AlreadyKnown \\ \hline \Xi BirthdayBook \\ name? : NAME \\ result! : REPORT \\ \hline \textit{name}? \in known \\ result! = \textit{already-known} \\ \hline \end{array}$$

Now a robust version of $AddBirthday$ is given by:

$$RAddBirthday \mathrel{\widehat=} ((AddBirthday \wedge Success) \vee AlreadyKnown)$$

Formally, however, none of the operations described so far is a refinement of $AddBirthday$[2], so in what sense is it strengthened? Maybe a notion of refinement exists which formalises the "strengthening" that Spivey had in mind there.

2 Why IO-refinement?

We could think of (at least) six possible reasons for wanting or allowing refinement of input and output parameters in Z operations. One aspect of the question can be rephrased as: why would we call $AddBirthday \wedge Success$ (which only adds a constant output) a refinement of $AddBirthday$? However, some of the answers indicate the possibility or desirability of IO-refinements much more radical than that – for example, to match notions of conformance of object interfaces.

A reader who is already convinced of the usefulness of IO-refinement may safely skip the rest of this section.

2.1 The Non-conformist Answer

Surely it's only a convention?

We have a particular interest in alternative definitions of refinement for Z, because we wish to use Z for *partial specification* (or *viewpoint* specification), as explained in [4, 3]. In our framework [7], the meaning of a partial specification is the set of all its possible developments according to a particular development relation – and the obvious development relation for Z is (data) refinement. Composition of two partial specifications, in that approach, is finding their most general common refinement. In [4] we have shown how to generate a least common refinement of two Z specifications with this purpose in mind. This type of partial specification, however, only allows viewpoints which satisfy the following restrictions:

1. Each operation in every partial specification can be invoked by the environment, i.e. none of them are internal. (Refinement allows weakening of preconditions, which is undesirable for internal operations. At ZUM'97 [6], there were at least four papers which touched upon this issue.)
2. All partial specifications are at the same level of granularity, e.g. no single operation in one partial specification corresponds to a larger collection of operations in another.
3. All partial specifications of one operation have exactly the same inputs and outputs.

[2] However, $RAddBirthday$ is a refinement of $AddBirthday \wedge Success$ and of $AlreadyKnown$. In fact, due to the preconditions of these operations being disjoint, it is their least common refinement.

The first of these restrictions is removed by a slight generalisation of Z refinement, called *weak refinement* [9]. The second one suggests a move towards *action refinement.* This paper aims at relaxing the third restriction, which appears to be orthogonal to the other two. We wish to allow viewpoints which need only describe part of the required functionality of an operation – it seems logical to apply this principle to inputs and outputs as well, particularly in the object-based contexts in which we wish to use viewpoint specification (cf. [7]). For example, in a system which logs all user requests, the specification of a particular request in the "user" viewpoint should not have to have a $log!$ output – these should only occur in the specification of that same request in the "logger" viewpoint.[3]

A comparable argument can be found in [11], which demonstrates the usefulness of separating user interface and functionality in Z. Our work can be viewed as a natural extension of theirs, in that it studies how refinement combines with this separation.

Clearly one way out for us would be to leave Z refinement for what it is, and define a "new" refinement relation for Z from scratch. It might solve our problem, but we think it would be much more profitable to stay as close as possible to generally accepted approaches. Fortunately, as will be made clear later, this poses no serious problems.

2.2 The Informal Answer

Adding a constant output shouldn't hurt, should it?

From the perspective of "information content", there is no difference at all between observing that a particular operation occurs, and observing that it occurs with a particular output which is known always to have the same value. This seems similar to giving an operation a different name – which *is* allowed in Z data refinement. Also, one might wonder whether the *name* of an output should have relevance in Z refinement – in most languages outputs do not have names, only types. Another branch of our research is concerned with relating specifications in Z with specifications in other formal notations. For some of these, e.g. LOTOS [8] we have indeed needed to adopt the convention that names of outputs (and inputs) are irrelevant in Z.

The argument here will be continued in a more formal way in other answers. "There is no difference at all" would suggest more than just refinement – it suggests equivalence, for example in the sense of refinement in both directions. Note that here we talk about "observing" an operation happening – this is fully in accordance with the model used in [18] to derive Z refinement, which we will also be using later to derive a more general refinement relation.

2.3 The Perspective of Operation Interfaces

If no one is looking at $result!$, there is no difference.

The standard Z view of refinement (simulation) between abstract data types is that the environment observes the occurrence of operations, and for each of those, all its inputs and outputs with their names and types. As a consequence of this, the *types* of input and output parameters may not be changed[4], though the refinement conditions do allow their declarations to change[5].

Looking at these restrictions from the perspectives of operation interfaces and object-oriented style typing, this all seems more restrictive than necessary. We would, like in interface definition languages, an operation's interface, viewed as a type, to be satisfied by operations whose interfaces are *subtypes* of that type: taking inputs and outputs from extended sets; adding inputs and outputs (by generalising imagined constant inputs and outputs to variable ones). The type system of Z (being essentially flat) is not well suited for defining subtyping directly. We will instead (implicitly)

[3] One might point out that this could be resolved by having the log as a state variable in the logger viewpoint only, which is correct. However, such a quasi-solution illustrates exactly the point we wish to make: Z refinement allows near-unlimited change of state variables, and none of input/output variables.

[4] Strictly speaking, in this interpretation the names of operations should not be changed in refinement either. However, giving the "concrete" and "abstract" operations different names solves the problem of determining which "version" the operation name (e.g. in a refinement proof obligation) refers to. A somewhat cleaner approach to this is taken in B [1], where operation names are explicitly required to be the same after refinement, but there the proof obligations need to refer to more constituents of the operation definition than the ones for Z.

[5] This is an obvious consequence from the possibility to move information between declarations and predicates of a schema. The sets the input and output parameters are taken from may be extended to any larger set lying within the same type, or restricted to any subset containing all values actually allowed by the operation.

define subtyping as the existence of functions with particular properties that relate the types. The conditions that emerge for inputs and outputs are different, as one might have expected from issues of covariance and contravariance in function subtyping.

In the research area of object-oriented languages, the need for interface refinement has indeed been acknowledged. The most striking example of this is the work of Mikhajlova and Sekerinski [14] on class refinement and interface refinement, which independently of our work comes up with very similar rules and conditions for alteration of inputs and outputs in refinement.

2.4 A Schema Calculus Answer

Schema conjunction generally precludes refinement – but through applicability, not through correctness.

It is well known in the Z community that the interplay between refinement and the schema calculus is not always smooth. The conjunction of two operations, for example, may not be a refinement of either even when a common refinement of the two exists [4]. Conversely, the disjunction of two operations is only the least common refinement of both if their signatures are identical and they are identical where their preconditions overlap (e.g. when the preconditions are exclusive [2], like for $AlreadyKnown$ and $AddBirthday \wedge Success$ above). A more positive result is the following.

Theorem 1 The operation $COp \mathrel{\widehat=} (AOp \wedge P)$ is an operation refinement of AOp if

1. COp has the same signature as AOp, i.e. P does not refer to variables not in AOp;
2. pre $COp =$ pre AOp, i.e. P does not have the effect of excluding before-values that AOp allowed.

Proof The correctness condition is automatically guaranteed by the second condition. The second condition above is only a *formal* strengthening of the applicability condition, since pre$(AOp \wedge P) \Rightarrow$ pre AOp follows from the first condition. □

Of the two conditions of the above theorem, surely the second one should be considered the major one. Now look at $AddBirthday$ and $AddBirthday \wedge Success$ ... Their preconditions are equal, so they only fail to be in the refinement relation because of the first, minor syntactical, condition. Clearly from this perspective removing that condition, and thus putting these operations in a refinement relation, would not be very revolutionary.

This line of reasoning, however, will not be pursued much further in the rest of this paper, because even if we manage to overcome the problem described above, it still seems impossible to completely remove all problems between schema calculus and refinement.

2.5 A Mathematical Answer

$A \times \mathbb{1} \cong A$

Contrary to what was suggested by the previous answer, in moving from $AddBirthday$ to$AddBirthday \wedge Success$, "conjunction" is really irrelevant, being only a consequence of Z's syntactical conventions. Consider the following tautologies on sets (in mathematics, not in Z):

$$A = \{x \mid x \in A\} \qquad\qquad B = \{y \mid y \in B\}$$

Now combine these set comprehensions in a particular syntactic way, by concatenating their declarations and taking the conjunction of their predicates:

$$\{x, y \mid x \in A \wedge y \in B\}$$

Now would we call the resulting set $A \wedge B$? Of course not – what is being constructed is a *product* set. The same goes for $AddBirthday \wedge Success$ – maybe we should say that it is *unfortunate* that product schemas in Z appear as conjunctions[6], because products as a concept are too important to be lumped in with conjunctions.

Once we have established that $AddBirthday \wedge Success$ denotes a product schema, let us consider its factors. $Success$ is peculiar, in that it has only one possible inhabitant. Viewed as a set of bindings, it is a singleton set. One notion of equivalence in a theory of products is that of *isomorphism*, and the product of a set A with a singleton set is isomorphic to A. (This formalises the argument given in the "informal" answer.) There is another pair of isomorphic types involved in these schemas. Consider the type of all outputs of $AddBirthday$ – there are none. One way of modelling a value from an empty product type is by defining it to be the anonymous unique value * of the predefined type $\mathbb{1}$. The output signature of $Success$ is $Success$ itself, and the product of these two output signatures is isomorphic (if not equal!) to $Success$, and isomorphic to $\mathbb{1}$.

Isomorphisms between state schemas in Z induce data refinements in both directions. Isn't it strange that isomorphisms between input- or output types do not?

As a response to this question we expect to hear "inputs and outputs are observed, whereas states are not". This is an acceptable answer within certain limits, as will be explained in Section 3.1. However, taking a different view on this introduces no inconsistencies or new anomalies.

"Isomorphism" again suggests IO-refinement in both directions. It is clear that less stringent conditions will suffice for having IO-refinement in one direction only. Such conditions will emerge from the derivation given in the next section.

Finally, we would like to point out the similarity between this issue and the *generalisation* or *embedding* strategy in program transformation [15] (or indeed in proofs in general): input types can be extended by generalising constants to variables; output types can be generalised as long as the originally required output can be reconstructed from it.

2.6 A Methodical Answer

We can derive it, so it must be okay.

Woodcock and Davies [18] motivate the Z data refinement conditions by deriving them from a characterisation of simulations between abstract data types, modelled as binary relations. Using exactly the same set up, by generalising identity functions to more general functions in their derivation, we managed to obtain conditions for a generalised type of refinement, which we call IO-refinement. This involves only a very marginal relaxation in the "rules of the game", very similar to the implications of having initialisations observable.

Of course, a formal derivation, however reassuring, on its own means nothing. However, the generalisation we choose is the obvious one given Woodcock and Davies' derivation (replacing explicit occurrences of identity functions by more general functions), and the resulting refinement relation encompasses all the relaxations suggested by the answers above, and possibly quite a few more.

Stepney, Cooper and Woodcock in recent work [17] have also observed that more powerful rules than the standard rules for Z can be derived from the binary relation characterisation.

3 A Derivation of the Conditions for IO-refinement

In this section we will derive conditions for IO-refinement in a derivation that generalises the derivation of refinement from simulation in [18, pages 254-255]. In order to appreciate the subtleties of IO-refinement, we first need to discuss the issue of *unwinding* – transforming a system where all input occurs at initialisation and all output occurs at finalisation into one where every operation can have input or output.

3.1 Unwinding: the Relation between Initialisations and Input

The canonical theory of simulations on which the notion of refinement in Z is based [12, 13] is one of binary relations. A "run" of the system consists of three parts: an initialisation, a sequence of operations, and a finalisation. If the state

[6] There is a strong analogy here with products in relational (database) algebra appearing as natural joins.

space of the environment is *G* and the state space of the system is *S*, then initialisation is a relation between *G* and *S*; every operation is a relation on *S*; and the finalisation is a relation between *S* and *G*. As in [18], let us concentrate on a system which has only one operation – multiple operations, each with their own input/output types, would make everything that follows unnecessarily complicated. This simplification has the additional consequence that, apart from the initialisation condition for data refinement (which is unaffected by our modified derivation), IO-refinement is a condition on a single pair of operations. We will sometimes say that an *operation* IO-refines another one when strictly speaking we are talking about the *systems*, each consisting of one operation and the implied state, refining each other.

The first part of the derivation in [18] ("relaxing") results in conditions for data refinement of a system as described above, i.e. one with no individual inputs or outputs for every operation. These conditions are the relational versions of the correctness and applicability conditions given in Section 1.1. For a relation co to refine a relation ao given abstraction relation r:

$$(\mathrm{dom}\, ao) \lhd (r \fatsemi co) \subseteq ao \fatsemi r \tag{2}$$

$$\mathrm{ran}(\mathrm{dom}\, ao \lhd r) \subseteq \mathrm{dom}\, co \tag{3}$$

A particular class of such systems is interpreted as modelling a system with inputs and outputs for every operation, namely ones where the following is the case. The state contains 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 (the) operation is executed, the first value is removed from the input sequence, and 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.

Example 4 The system (call it A)

State

$x{:}S$

Init

$State'$

$x' = a$

Op

$\Delta State$

$y?{:}A$

---

p

(it has no outputs, to simplify the presentation) is modelled by the system (let us call it A^*) where all inputs are provided at initialisation time, which is

State

$x{:}S$

$inps : \mathrm{seq}\, A$

Init

$State'$

$x' = a$

Op

$\Delta State$

$p[hd\, inps\, /\, y?]$

$inps' = tl\, inps$

□

The description at the level of binary relations of this construction uses a number of auxiliary functions defined below, which will also be used in our modified derivation. $split$ removes one input from the input sequence; $\|$ is a kind of parallel composition; $merge$ adds an output to the output sequence.

[A,B,C]

$split : A \times (\mathrm{seq}_1\, B \times \mathrm{seq}\, C) \rightarrow (A \times B) \times (\mathrm{seq}\, B \times \mathrm{seq}\, C)$

$\forall s{:}A;\ is : \mathrm{seq}_1\, B;\ os : \mathrm{seq}\, C \bullet$
$split\,(s,(is,os)) = ((s,head\, is),(tail\, is,os))$

[W,X,Y,Z]

$\_ \| \_ : (W \leftrightarrow Y) \times (X \leftrightarrow Z) \rightarrow W \times X \leftrightarrow Y \times Z$

$\forall \rho : W \leftrightarrow Y;\ \sigma : X \leftrightarrow Z;\ w{:}W;\ x{:}X;\ y{:}Y;\ z{:}Z \bullet$
$(w,x) \mapsto (y,z) \in \rho \| \sigma \Leftrightarrow w \mapsto y \in \rho \land x \mapsto z \in \sigma$

$$[A,B,C]$$
$$merge : (A \times C) \times (\mathrm{seq}\, B \times \mathrm{seq}\, C) \rightarrow A \times (\mathrm{seq}\, B \times \mathrm{seq}\, C)$$
$$\forall s{:}A ;\ o{:}C ;\ is : \mathrm{seq}\, B ;\ os : \mathrm{seq}\, C \bullet$$
$$merge\,((s,o),(is,os)) = (s,(is, os \frown \langle o \rangle))$$

Using these operations, the relation between an operation op in a system with IO and its counterpart op_s in a system without IO is

$$op_s = split \fatsemi (op \| id) \fatsemi merge$$

where the identity relation id ensures that the rest of the input and output sequence are unaffected in the current operation.

The second part of the derivation then translates the refinement conditions between ao_s and co_s (substituting these for ao and co in (2) and (3)) into conditions between ao and co, using as a retrieve relation on the extended state

$$r_s = r \,\|\, id[\mathrm{seq}\, Inp] \,\|\, id[\mathrm{seq}\, Outp] \tag{5}$$

where Inp and $Outp$ are the types of the input and output of op. The conditions that result, when translated to conditions on Z schemas, are those given in Section 1.1.

Note, however, that there is a small problem with representing a system with input in the way described above. (This is not observed in [18], but solved by the more general rules in [17].) It is best illustrated by means of an example.

Example 6 Continuing from Example 4, assume that *A*, the set from which the input parameter $y?$ is taken is properly contained in another set *B*. Clearly the first system can be refined by replacing the declaration of $y?$ in Op by $y? : B$, and let us call this system B. Now B^* is equal to A^*, except for the declaration of $inps$, which is now of type $\mathrm{seq}\, B$. However, B^* *fails to be a forward simulation refinement of* A^*! Consider the condition

$$\forall CState' \bullet CInit \Rightarrow \exists Astate' \bullet AInit \wedge Abs'$$

which in this particular case (abstraction relation is injection of abstract state space into the concrete one) translates to $\forall x' : S ;\ inps' : \mathrm{seq}\, B \bullet x' = a \ \Rightarrow\ inps' \in \mathrm{seq}\, A$ which is false for *A* being a proper subset of *B*. □

Conclusion: after "unwinding", refinements are allowed that were not allowed before.

The relevance of this problem is as follows. When using input-refinement, the initialisation condition for "all input at initialisation" translates to: for every concrete input, there is an abstract input which is linked to it by the input-transformer. This implies that new inputs may freely be added, but that no input may have its type extended. This is clearly undesirable given our motivations, but the issue needs to be resolved for "standard" Z refinement as well.

Of course the problem is in viewing the initialisations as independent. For the "input as initialisation" notion, it is important to realise that the inputs are provided by the environment. In the original relational refinement set-up, initialisation and finalisation are functions from- and to the environment, respectively. In other words, a fair model of A^* would have as its initialisation:

$$Init$$
$$State'$$
$$inps? : \mathrm{seq}\, A$$
$$x' = a$$
$$inps' = inps?$$

such that, with appropriately modified refinement rules for initialisations with inputs, A^* would still be refined by B^*. This construction solves the paradox inherent in [18] – in the following derivation we assume that a similar construction can also be given to discharge the undesirable condition for IO-refinement deriving from the initialisation condition. [17] presents a solution which is essentially similar, by always keeping a reference to the global environment *G* in the refinement rules.

3.2 The Derivation

In this section we paraphrase the second part of the derivation in [18], allowing for an easy comparison between the two. The crucial step lies in generalising the identity functions in (5) to arbitrary maps. We will use the following relational generalisation of "map":

$[A,B]$

$$\_^* : (A \leftrightarrow B) \rightarrow (\mathrm{seq}\, A \leftrightarrow \mathrm{seq}\, B)$$

$$\forall \rho : A \leftrightarrow B ;\ as : \mathrm{seq}\, A ;\ bs : \mathrm{seq}\, B \bullet$$
$$(as, bs) \in \rho^* \Leftrightarrow \#as = \#bs \wedge \forall i : \mathrm{dom}\, as \bullet (as\, i, bs\, i) \in \rho$$

The starting point of our derivation is the assumption that we have refinement rules for a particular relation between concrete and abstract states for a set of operations that have no inputs or outputs. If we want to derive similar conditions for operations that do have inputs and outputs, we need to apply the standard method: the state contains extra components representing the sequence of inputs still to be dealt with and the sequence of outputs already computed.

Assume we wish to compare operations ao and co which consume input and produce output. Their equivalent operations that expect input and output sequences are given by

$$ao_s = split \fatsemi (ao \| id) \fatsemi merge$$

and similarly for co_s. 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. If r is a relation of type

$$AState \leftrightarrow CState$$

then we require a relation r_s of type

$$AState \times (\mathrm{seq}\, AInp \times \mathrm{seq}\, AOutp) \leftrightarrow CState \times (\mathrm{seq}\, CInp \times \mathrm{seq}\, COutp)$$

in order to compare ao_s and co_s. For the comparison to make sense, the two operations should have lists of outputs and inputs that are equally long. Moreover, the relation between input and output sequences should be between elements in comparable positions only. Not to assume that would imply the system cheated in some way – by making use of "future" input or "past" output, for example. If the relation between individual input elements is it and that between individual output elements is ot, the relation r_s between enhanced states is then defined by:

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

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. (This condition is formally a little too strict – it rules out input and output transformers which reduce the input and output types to the values that can actually occur. However, this can still be done as operation refinement, using just predicates.)

The rules for the correctness of a forwards simulation require that

$$(\mathrm{dom}\, ao_s) \lhd (r_s \fatsemi co_s) \subseteq ao_s \fatsemi r_s$$

which requirement is equivalent to

$$(\mathrm{dom}\, ao) \lhd ((r \| it) \fatsemi co) \subseteq ao \fatsemi (r \| ot)$$

The other requirement, that co_s is defined everywhere that ao_s is defined, leads to a second constraint[7]:

$$\mathrm{ran}((\mathrm{dom}\, ao) \lhd (r \| it)) \subseteq \mathrm{dom}\, co$$

[7] [18] has $id[Output]$ in the corresponding condition – it should be $id[Input]$.

The first derivation in [18] also contains an initialisation condition that needs to hold between the initialisations of the two systems involved, which is claimed to be independent from the "unwinding" construction and thus does not change for the second derivation. This is not strictly true, as demonstrated above. Similarly, this condition would have its effect on our modified version of unwinding. The resulting condition would be that for every concrete input, an abstract input exists. (Compare with the situation in Example 6.)

In addition, a *finalisation* condition exists for a simulation to hold between two systems with no inputs or outputs. This condition becomes moot in Woodcock and Davies' unwinding, but in our modified approach it has an important consequence:

$$(ot \fatsemi ot^{\sim}) \subseteq id[AOutp]$$

which is to say, ot needs to be injective. This condition guarantees that different abstract ("original") outputs can be distinguished in the concrete cases because their concrete representations will be different as well.

The next section will define the analogues of the retrieve relation for changing inputs and outputs in refinement steps, called *IO-transformers*, and some other notations useful for expressing the IO-refinement rules at the Z schema level.

4 IO-transformers

For convenience, we first define the notion of a signature.

Definition 7 (Signatures) For a schema S, its signature ΣS and its input and output signatures $?S$ and $!S$ are given by

$$\Sigma S = S \lor \neg S$$

$$!S = \Sigma\ (\exists \text{ "}\textit{all components of S whose names don't end in !}\text{"} \bullet S)$$

$$?S = \Sigma\ (\exists \text{ "}\textit{all components of S whose names don't end in ?}\text{"} \bullet S)$$ □

By definition of schema negation, a signature schema has all components declared as being from their *type* (rather than a subset of it) and an everywhere true predicate (which we usually forget about). If S has no inputs, $?S$ turns out to be $[\text{true}]$, the schema whose only inhabitant is the empty binding (similarly for $!S$ if S has no outputs.) This schema plays the role of the unit type $\mathbb{1}$ mentioned in Section 2.5.

The following operations on names, which extend componentwise to signatures (similarly to the standard convention for decorations) are defined.

Definition 8 (Decorations for input and output) For all x, $\overline{x?} = x!$; $\overline{x!} = x?$. □

(The overline operator is chosen in analogy with CCS - in piping communication when they are both present in the right direction, x and $\overline{x}$ become hidden.)

We will need to change inputs and outputs of operation schemas without (directly) affecting their changes on the state. For that purpose we will use schemas which contain inputs and outputs *only*, which will be connected to the operations using *piping* $\gg$. Hayes and Sanders [11] 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".

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.

Definition 9 (Input and output transformer) Schema S is an input transformer for schema T iff

$$\Sigma S = (!S \wedge ?S) \ \wedge \ \overline{!S} = ?T$$

and it is an output transformer for T iff

$$\Sigma S = (!S \wedge ?S) \ \wedge \ \overline{?S} = !T$$

□

An IO-transformer will normally have a predicate relating its inputs and outputs. For every schema, input- and output transformers can be defined that act as identities with piping:

Definition 10 (Input and output identity) For a schema S its input identity is defined by

$$\mathsf{Iid}\ S = [?S; \ \overline{?S} \mid \textbf{all}\ x\ \textbf{in}\ ?S \bullet x = \overline{x}]$$

and its output identity by

$$\mathsf{Oid}\ S = [!S; \ \overline{!S} \mid \textbf{all}\ x\ \textbf{in}\ !S \bullet x = \overline{x}]$$

where **all** x **in** $S \bullet P[x]$ denotes universal quantification over all component names in S, i.e. $P[x] \wedge P[y] \wedge P[z]$ if the components of S are x, y and z. □

Clearly $\mathsf{Iid}\ S \gg S = S$ and $S \gg \mathsf{Oid}\ S = S$. Another special kind of IO-transformers are terminators and generators, IO-transformers that only consume input or only produce output.

Definition 11 (Terminator and generator) For IO-transformer S:

- S is a *terminator* iff $\Sigma S = ?S$
- S is a *generator* iff $\Sigma S = !S$ □

Example 12 Consider schemas

$State$

$x : \mathbb{N}$

Op

$\Delta State$
$y? : \mathbb{N}$
$x' = x + y?$

Gen

$y! : \mathbb{N}$
$y! = 17$

Then Gen is a generator, since $\Sigma Gen = !Gen = [y! : \mathbb{Z}]$. A generator used as the first argument of a piping fixes an input value, e.g. $Gen \gg Op$ is (no renaming needed):

$$\exists y? : \mathbb{Z}; \ y! : \mathbb{Z} \bullet [Gen; \ Op \mid y? = y!]$$

which simplifies to

$\Delta State$
$x' = x + 17$

□

It is most obvious to have generators as input transformers, and terminator as output transformers. However, the reverse is possible as well for operations that have no inputs or no outputs.

Example 13 Continuing the birthday book example from Section 1.2, $Success$ is a generator. It is not an input transformer for $AddBirthday$ however (it has no component $name!$), but it *is* an output transformer for $AddBirthday$ (because it has no inputs). Its use in $RAddBirthday$ is as an output transformer indeed, observe that $AddBirthday \gg Success = AddBirthday \wedge Success$. □

5 Conditions of IO-refinement

Now we have defined the equivalent of schema composition for input and output parameters, we can rephrase the conditions of IO-refinement as conditions on Z schemas.

Definition 14 (IO-refinement) Given abstract data types A=(A,$\{AOp\}$,$AInit$) andC=(C,$\{COp\}$,$CInit$), then C is an IO-refinement of A if there exists an abstraction schema

$$\begin{array}{|l} \hline R \\ A;\ C \\ \hline Pred \\ \hline \end{array}$$

such that the following conditions hold:

initialisation $\forall C \bullet CInit \Rightarrow (\exists A \bullet AInit \wedge R)$

and schemas IT and OT exist such that

transformers IT is a total input transformer for AOp; OT is a total and injective output transformer for AOp.

applicability COp is defined on all representatives of A and $?COp$ on which AOp is defined (modulo retrieve relation and inverse input transformation):

$$\forall A;\ C;\ ?COp \bullet \text{pre}(\overline{IT} \gg AOp) \wedge R \ \Rightarrow\ \text{pre}\, COp$$

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:

$$\forall A;\ C;\ ?AOp;\ C';\ !COp \bullet \text{pre}\, AOp \wedge R \wedge (IT \gg COp) \ \Rightarrow\ \exists A' \bullet R' \wedge (AOp \gg OT)$$

□

This definition easily extends to multiple operations.

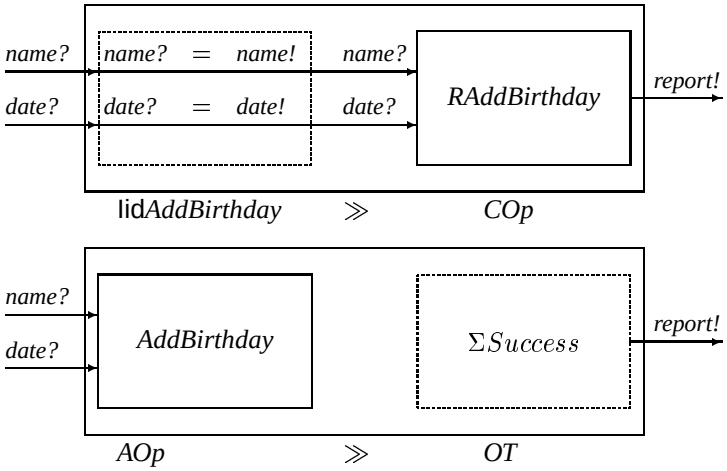
A very important point about IO-refinement is that, when we have done a refinement step using transformers *IT* and *OT*, we can *not* forget about them (unlike the abstraction schema). We will need to administrate all IO-transformers used, so they can be "plugged in" in the final result of our refinement derivation in order to get inputs and outputs as they were originally specified.

Observe that we have chosen for both IO-transformers to have abstract inputs and concrete outputs, so that they are typically used in opposite directions. Strangely enough, the correctness condition looks like the corresponding condition for normal data refinement between ($IT \gg COp$) and ($AOp \gg OT$), whereas the applicability condition looks like the corresponding condition between COp and $\overline{IT} \gg AOp$.

Example 15 For the birthday book example, in order for $RAddBirthday$ to be an IO-refinement of $AddBirthday$, we need (instantiations R=A=C=$BirthdayBook$; $OT = !RAddBirthday = \Sigma Success$; $?AddBirthday = ?RAddBirthday = [name? : NAME]$; $!AddBirthday = [\text{true}]$; $IT = \mathsf{Id}\ AddBirthday = [name?, name! : NAME;\ date?, date! : DATE \mid name? = name! \wedge date? = date!]$) for correctness:

$$\begin{array}{l} \forall BirthdayBook, BirthdayBook', ?AddBirthday, !RAddBirthday \bullet \\ \quad name? \notin known \wedge RAddBirthday \ \Rightarrow\ \exists BirthdayBook' \bullet AddBirthday \gg \Sigma Success \end{array}$$

or, as observed above, "correctness" of operation refinement between the following two operations:



and for applicability:

$$\forall BirthdayBook, ?RAddBirthday \bullet name? \notin known \Rightarrow \text{pre}\, RAddBirthday$$

which both obviously hold.

This implicitly also proves that $AddBirthday \gg Success$ refines $AddBirthday$. The reverse can also be proved, making these two IO-refinement equivalent. □

An obvious observation is that IO-refinement is a proper generalisation of data refinement (of course it already follows from the derivation being a generalisation).

Theorem 16 IO-refinement generalises data refinement.

Proof For IT substitute lid COp, for OT substitute Oid AOp, this reduces the first condition to the traditional one. For the second condition, observe that lid COp = lid AOp and that $\overline{\text{lid } S}$ = lid S for any S, then it also reduces to the traditional condition. □

The next thing to be proved is that, compared to data refinement, it has equally nice properties. The previous theorem ensures one of these already:

Corollary 17 IO-refinement is reflexive. □

The following theorem does not follow directly.

Theorem 18 IO-refinement is transitive.

Proof The proof is completely analogous to a proof that data refinement is transitive. The witnessing abstraction relation for the two-step refinement is a composition of the two abstraction relations (conjunction of both, hiding components of the intermediate state), the input and output transformers for the two-step refinement are constructed by piping the transformers of the individual steps together.

Proofs like these are best done in the set up of binary relations as in Section 3, rather than in the schema formulation. □

Now if the conditions for data refinement were already complicated, the ones for IO-refinement add an extra level of complication to them. Even if the relevant input and output transformers are fixed, the formulas are quantified over two more variables than the data refinement ones. The complexity of the rules for data refinement has been observed before, and a solution has been proposed which separates out the "most general data refinement", which is completely determined by the abstract type and the abstraction relation. The operation refinement implicit in the data refinement

rules is then carried out in a separate step. This strategy is originally described in [13], and more explicitly for Z in [5, section 4]; the chapter on calculating data refinements in [18] describes a special case of it.

We will make IO-refinement more tractable by proposing a similar strategy: first do an IO-refinement step in which no data refinement occurs (often resulting in $\Leftrightarrow$ holding where the conditions require $\Rightarrow$), and then do any data refinement. Unlike with the solution for most general data refinement described above, we will make no claims about completeness of this strategy. In other words, not all IO-refinements allowed by the general rules can actually be described as a restricted IO-refinement followed by data refinement. Further investigation may provide a complete rule that is nevertheless easy to use.

The essence of our suggestion for restricted IO-refinement lies in the following theorems, which give *sufficient* conditions for IO-refinement.

First, the result of applying bijective IO-transformers to an operation will always be a refinement, i.e. inputs and outputs can be replaced by "isomorphic" values:

Theorem 19 (Isomorphic IO) If IT is a total injective functional input transformer for Op and OT is a total injective output transformer for Op, then $\overline{IT} \gg Op \gg OT$ IO-refines Op.

Proof Considering the conditions of IO-refinement as in Definition 14, abstract state *A*, concrete state *C* and abstraction relation are all identical. Taking $COp = \overline{IT} \gg AOp \gg OT$, observe that for injective functional IT, $IT \gg \overline{IT} = \mathsf{lid}\ AOp$, and for total OT, $\mathrm{pre}(\overline{IT} \gg AOp \gg OT) = \mathrm{pre}(\overline{IT} \gg AOp)$. □

Also, we can add inputs and outputs to an operation which previously did not have any.

Theorem 20 (Inventing inputs and outputs) If $AOp = Gen \gg COp \gg Term$, Gen is a *fixed* generator which is an input transformer for COp, $Term$ is a terminator and output transformer for COp, then COp IO-refines AOp.

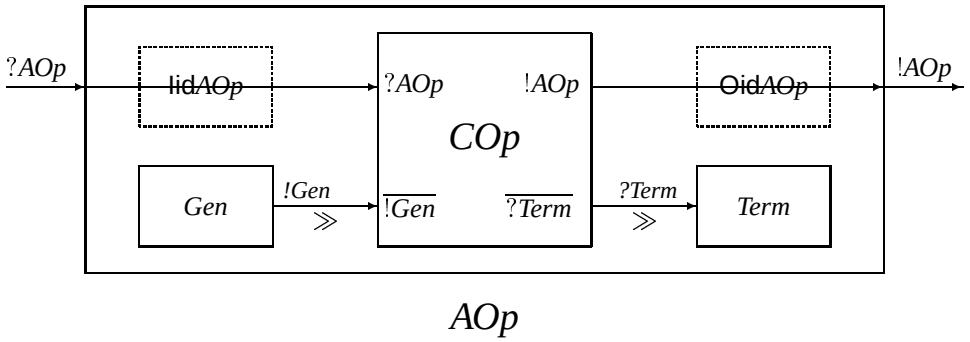
Proof Let $IT = Gen$, and $OT = \overline{Term}$ (which is injective). Observe that, for applicability, $\overline{Gen} \gg Gen = \mathsf{lid}\ Gen \Rightarrow \mathsf{lid}\ COp$ and for correctness, $Term \gg \overline{Term} = \Sigma(\mathsf{Oid}\ COp) \Leftarrow \mathsf{Oid}\ COp$. □

Moreover, inputs and outputs can also be added to an operation which did have some already.

Theorem 21 If, for a fixed generator Gen such that $\Sigma\,\overline{Gen} \cap ?COp = \varnothing$ and a terminator $Term$ such that $\Sigma\,\overline{Term} \cap !COp = \varnothing$ it is the case that $AOp = Gen \gg COp \gg Term$, then COp IO-refines AOp.

Proof Note that Gen and $Term$ are not necessarily input and output transformer for COp, so the IO-refinement rule cannot directly be used. This can be fixed by taking the equivalent $(Gen \wedge \mathsf{lid}\ AOp) \gg COp \gg (Term \wedge \mathsf{Oid}\ AOp)$. Then for $IT = (Gen \wedge \mathsf{lid}\ AOp)$ and $OT = \overline{Term \wedge \mathsf{Oid}\ AOp}$ the conditions of IO-refinement are satisfied, analogous to the proof of Theorem 20. □

A pictorial representation of this theorem illustrates how by IO-refinement we can change the boundaries of a system while we are developing it:



In the above diagram, COp IO-refines AOp, using a fixed generator Gen and a terminator $Term$.

Theorems 19 and 21 can also be viewed in the context of [11]: they give a kind of soundness conditions for separating out the user interface from the functionality of an operation.

In summary, using IO-refinement we can

- add inputs;
- add outputs;
- change inputs and outputs by applying injective functions to them;

and of course all possible combinations of the above, resulting in a quite powerful refinement calculus.

6 Examples

6.1 Coffee machine

A very simple and, in our view, convincing example of the usefulness of IO-refinement is that of coffee machines which have more buttons and outputs than "strictly needed". Consider the (stateless) machine for black coffee

Black

$$\begin{array}{l} coin? : COIN \\ drink! : \{BlackCoffee\} \\ \hline coin? \geq 10p \end{array}$$

and the advanced machine

Coffee

$$\begin{array}{l} coin? : COIN \\ milk?, sugar? : Boolean \\ drink! : COFFEE \\ receipt! : PAPER \\ \hline coin? \geq 10p \\ milk? \land sugar? \Rightarrow drink! = WhiteCoffeeWith \\ milk? \land \neg sugar? \Rightarrow drink! = WhiteCoffee \\ \neg milk? \land \neg sugar? \Rightarrow drink! = BlackCoffee \\ \neg milk? \land sugar? \Rightarrow drink! = BlackCoffeeWith \end{array}$$

These two are indeed related by IO-refinement: define *Term* and *Gen* by

Gen

$$\begin{array}{l} milk!, sugar! : Boolean \\ \hline \neg milk! \land \neg sugar! \end{array}$$

Term

$$receipt? : PAPER$$

then $Black = Gen \gg Coffee \gg Term$ and thus by Theorem 21, $Coffee$ is an IO-refinement of $Black$.

6.2 Sequences

A datatype of sequences of natural numbers could be defined by a type Seq with initialisation $InitSeq$ and sample operation Add:

$$Seq \triangleq [\, s : \mathrm{seq}\, \mathbb{N} \,]$$

$$InitSeq \triangleq [\, Seq' \mid s' = \langle\rangle \,]$$

$$Add \triangleq [\, x? : \mathbb{N};\ \Delta Seq \mid s' = s \frown \langle x? \rangle \,]$$

Now a very general sorting operation, which takes a binary relation as a parameter, is defined by:

$$Permute \triangleq [\, \Delta Seq \mid items\, s = items\, s' \,]$$

$$Sort \triangleq [\, Permute;\ leq? : \mathbb{P}(\mathbb{N} \times \mathbb{N}) \mid \forall i,j : \mathrm{dom}\, s' \bullet i < j \Rightarrow (s'[i], s'[j]) \in leq? \,]$$

How does the normal sorting operation on sequences of natural numbers relate to this operation, in terms of IO-refinement? This can be answered using a generator:

$$FixLeq \triangleq [\, leq! : \mathbb{P}(\mathbb{N} \times \mathbb{N}) \mid \forall x,y : \mathbb{N} \bullet (x,y) \in leq! \Leftrightarrow x \leq y \,]$$

Now $FixLeq \gg Sort$ is the obvious unparameterised sorting on sequences of natural numbers. By Theorem 20, $Sort$ is thus an IO-refinement of that.

Continuing this example, clearly not all binary relations are sensible as inputs to the sorting operation. We will add some error handling to this, using a new type for output reports called msg:

$$msg ::= wrong \mid ok$$

The error case, and a more robust version of the sorting operation are now described by:

$$CantSort \triangleq [\, leq? : \mathbb{P}(\mathbb{N} \times \mathbb{N});\ \Xi Seq;\ report! : msg \mid report! = wrong;\ \neg\, (leq? \fatsemi leq? \subseteq leq? \wedge \mathbb{N} \times \mathbb{N} \subseteq leq?) \,]$$

$$Success \triangleq [\, report! : msg \mid report! = ok \,]$$

$$RSort \mathrel{\widehat=} (Sort \wedge Success) \vee CantSort$$

$RSort$ may appear to be an IO-refinement of $Sort$. However, it is not, because $RSort$ is not a data refinement of $Sort \wedge Success$. (The preconditions of $Sort$ and $CantSort$ overlap.) Nevertheless, $RSort$ is an IO-refinement of $FixLeq \gg Sort$ because $FixLeq \gg CantSort$ is empty, and $FixLeq \gg RSort \gg Shred = FixLeq \gg Sort$ for the following terminator:

$$Shred \triangleq [\, report? : msg \,]$$

7 Conclusions and Related Work

We have presented a generalisation of Z refinement which is derived from the same theoretical framework that Woodcock and Davies used for deriving the conditions of Z refinement, in a similar way. The conditions that emerge are fairly complex, but they appear to be reasonable. We have given a few theorems that simplify the conditions for IO-refinement in particular cases. More examples need to be studied in order to decide whether IO-refinement is suitable for all uses suggested in Section 2, and to investigate whether the simplifying theorems cover enough "practical" cases. The example in the previous section at least suggests that IO-refinement opens the road for formally allowing implementations in terms of standard components with pre-derived implementations. It also gives a semantic interpretation for the standard technique of adding "diagnostic" outputs to Z operations.

Input and output transformers play a role which is slightly different to that of abstraction relations in data refinement. This is due to the fact that in the final specification a way must be found to convert abstract inputs into inputs for the concrete system, and a way to convert concrete outputs into abstract ones. Thus, in a derivation by IO-refinement, all input and output transformers used need to be administrated in order to create these conversions for the final system.

These combined transformers are very similar to the representation schemas Hayes and Sanders [11] use in order to separate user interface from functionality. Like in our approach, these will have to be combined with the "core" specification using schema piping. Their work concentrates on specification issues and ours concentrates on development through refinement, and as such they are complementary.

Woodcock and Davies [18, Example 17.6] include one example of a refinement in Z in which input parameters are different for the concrete and abstract specifications. However, this appears to be possible only because their refinement conditions do not quantify over inputs, and because the operations involved do not actually use the inputs. We do not think that our rules for IO-refinement justify their claim of refinement in that example, one probably needs a complete action refinement for that.

Recent work by Stepney, Cooper and Woodcock [17] also addresses generalisation of the standard data refinement rules for Z, deriving from the binary characterisation of refinement. Their rules (which were given without a derivation) also allow refinement of inputs and outputs, by adding what we would call output transformers to non-trivial finalisations (and presumably also input transformers to initialisations). This amounts to the same construction as we have presented in Section 3, which may be confirmed by an article cited as "in preparation". The paper [17] does not have simplified refinement theorems as presented in this paper.

Independently of our work, Mikhajlova and Sekerinski [14] have recently investigated class refinement and interface refinement for object-oriented languages. Their results have a few very reassuring similarities to ours. First, their "interface refinement" condition generalizes their "class refinement" condition in much the same way as our IO-refinement generalises data refinement. Also, they conclude that sensible input-transformers should be surjective and output-transformers functional, which is exactly what we required (their transformers operate in the opposite direction). Also, the need to administrate the transformers that have been used appears in their notion of "wrappers".

References

[1] Abrial J-R. The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996

[2] Ainsworth M, Riddle S, Wallis PJL. Formal validation of viewpoint specifications. Software Engineering Journal 1996; 11(1):58-66

[3] Boiten EA, Bowman H, Derrick J, Steen MWA. Viewpoint consistency in Z and LOTOS: A case study. In FME'97 [10], pp 644-664

[4] Boiten EA, Derrick J, Bowman H, Steen MWA. Consistency and refinement for partial specification in Z. In: Gaudel M-C, Woodcock J (eds) FME'96: Industrial Benefit of Formal Methods, Third International Symposium of Formal Methods Europe. Springer-Verlag, 1996, pp 287-306 (Lecture Notes in Computer Science No. 1051)

[5] Boiten EA, Derrick J, Bowman H, Steen MWA. Coupling schemas: data refinement and view(point) composition. In: Duke DJ, Evans AS (eds) Northern Formal Methods Workshop, Springer-Verlag, 1997 (Electronic Workshops In Computing)

[6] Bowen JP, Hinchey MG, Till D (eds). ZUM '97: The Z Formal Specification Notation. Springer-Verlag, 1997 (Lecture Notes in Computer Science No. 1212)

[7] Bowman H, Boiten EA, Derrick J, Steen MWAS. Viewpoint consistency in ODP, a general interpretation. In: Najm E, Stefani J-B (eds) First IFIP International workshop on Formal Methods for Open Object-based Distributed Systems, Chapman & Hall, 1996, pp 189-204

[8] Derrick J, Boiten EA, Bowman H, Steen MWAS. Translating LOTOS to Object-Z. In: Duke DJ, Evans AS (eds) Northern Formal Methods Workshop, Springer-Verlag, 1997 (Electronic Workshops In Computing)

[9] Derrick J, Boiten EA, Bowman H, Steen MWAS. Weak refinement in Z. In: ZUM'97 [6], pp 369-388

[10] Fitzgerald J, Jones CB, Lucas P (eds). FME'97: Industrial Application and Strengthened Foundations of Formal Methods. Springer-Verlag, 1997 (Lecture Notes in Computer Science No. 1313)

[11] Hayes IJ, Sanders JW. Specification by interface separation. Formal Aspects of Computing 1995; 7(4):430-439

[12] He Jifeng, Hoare CAR, Sanders JW. Data refinement refined. In: Robinet B, Wilhelm R (eds) Proc. ESOP 86, Springer-Verlag, 1986, pp 187-196 (Lecture Notes in Computer Science No. 213)

[13] He Jifeng, Hoare CAR. Prespecification and data refinement. In: Data Refinement in a Categorical Setting. Oxford University Computing Laboratory, 1990 (Technical Monograph PRG-90)

[14] Mikhajlova A, Sekerinski E. Class refinement and interface refinement in object-oriented programs. In FME'97 [10], pp 82-101

[15] Partsch H. Specification and Transformation of Programs - a Formal Approach to Software Development. Springer-Verlag, Berlin, 1990

[16] Spivey JM. The Z notation: A reference manual. Prentice Hall, 1989

[17] Stepney S, Cooper D, Woodcock J. More powerful Z data refinement. In: Bowen J, Fett A, Hinchey M (eds) ZUM'98: The Z Formal Specification Notation. Springer-Verlag, 1998, pp 284-307 (Lecture Notes in Computer Science No. 1493)

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