Refinement of objects and operations in Object-Z

John Derrick and Eerke Boiten
Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Phone: + 44 1227 764000, Email: J.Derrick@ukc.ac.uk.)

Abstract

In this paper we describe how we can refine both objects and operations in an Object-Z specification. In particular, we will be concerned with changes of granularity of both objects and operations. Objects in that we wish to change the structure of objects in a specification. Operations in that we wish to provide explicit support for *action refinement* in this language. There are clear advantages in being able to change such levels of granularity when performing a refinement. In this paper we discuss the issues surrounding such refinements and derive general rules to support their use. We illustrate our ideas by looking at a specification of a cash point machine at a bank.

Keywords: Object-Z; Refinement; Action refinement; Object structure.

1 Introduction

In this paper we illustrate, through a worked example, how we can refine the granularity of both objects and operations in Object-Z.

It could be argued that one of the principal problems of designing a system formally is the need to fix an appropriate level of abstraction when initially describing the system. In particular, designers need to choose the granularity of the operations or events in the initial model and there is a balance to be found between too few events arising from a very abstract view or too many events which can clutter the understanding (and indeed verification) of the system. Furthermore, in an OO notation the granularity of objects must also be fixed in the initial specification, and again there is a tension between a suitably abstract view vs the actual division of objects appropriate in an implementation. For example, an implementation may require objects to be distributed across several nodes, whereas in the initial specification such detail is neither relevant nor helpful.

The problem arises because there is little support for changing the granularity of specifications when they are developed. Development in the context of formal specification means the use of refinement, implementation or testing relations which define the acceptable implementations of a specification, usually on the basis that the implementation exhibits behaviour that was feasible in the original specification. Examples of development relations include data refinement in Z [8, 15], the failures-divergences preorder in CSP [9], and reduction in LOTOS [4].

However, the majority of these relations leave the granularity of components (e.g. operations and objects) unchanged. For example, refinement of events is usually atomic, an event cannot be broken down into its constituent parts under a refinement. It is this that forces the developer to retrofit the level of granularity in the initial specification. Clearly there would be many benefits in being able to change the granularity of both objects and operations in a refinement, and the purpose of this paper is to describe a solution to this problem in the specification language Object-Z.

Object-Z [7] is an object-oriented extension of the Z [12] notation, and there has been a certain amount of interest in using Object-Z to specify distributed systems, particularly within the ODP initiative. An Object-Z specification consists of a number of interacting classes and objects. Each class consists of a state space, an initialisation together with a collection of operations which change the state and thus define the behaviour of the class.

In Object-Z and Z the implementation relation is called data refinement [14]. The principle of data refinement is that the more concrete specification can reduce any non-determinism present in the abstract specification. To verify a refinement a retrieve relation is used which relates the concrete to abstract states and allows the comparison between the data types to be made on a step by step basis by comparing an abstract operation with its concrete counterpart.

The main work on refinement in state-based specification languages has been in a non-object-oriented setting, however there has been some related work on refinement for object-oriented formal methods. For example, a definition of refinement for Object-Z has been given in [11], however, a restricted subset of Object-Z was used where classes could not contain objects as state variables. In this paper we relax that restriction and show how we can use the definition of refinement to change the structure of objects in a specification.

In addition to objects the other aspect we consider is that of operations. The step by step comparison of operations that refinements make is possible because the specifications are assumed to be *conformal* [8], 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. Providing such an action refinement will help solve the problem mentioned above, namely it will allow a natural change of granularity to be expressed as part of the development process. This could also enable interleavings of concrete operations which might be necessary for efficiency reasons.

Action refinements arise naturally in a number of settings. The particular example we will look at will be a cash point machine with an operation that requires as input a sequence of digits representing the p.i.n. of the user. 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 input digits one by one.

Such action refinements have been extensively discussed in the context of process algebras, usually under the name of action refinement [2, 10]. The difficulty in a process algebra is due to the interleaving semantics and in particular the law that $a \| b = a;\ b \ [] \ b;\ a$. Incorporation of action refinement and the requirement that semantic equivalences should be congruences with respect to refinement means that the natural interleaving semantics has to be abandoned.

However, in Object-Z and Z there are no such constraints because there are no global behavioural constructors such as $\|$ in a process algebra. The $\|$ primitive in Object-Z is a schema calculus operator which builds a single new operation rather than defining a temporal constraint over existing behaviours. So the situation which causes the problem in a process algebra never occurs in a state-based language (i.e. the standard semantics can be used and equivalences all still hold). There are some simple examples of action refinements in state-based languages, e.g. protocol refinements in B [1], in Z [14] and buffers in B [5]. These approaches introduce a *skip* operation (i.e. stuttering step) in the abstract specification. Such an operation produces no change in the abstract state, and the concrete system is constructed so that one of the concrete operations refines the abstract operation whilst the other refines *skip*.

However, although some action refinements can be verified in such a manner we wish to go further and consider refinements where we split a collection of inputs or outputs across several concrete operations as in our cash point example. Because we are transforming the inputs/outputs in this fashion the concrete operations don't refine *skip*, and such a refinement cannot in general be verified using abstract stuttering steps. In this paper we develop machinery to verify action refinements where the inputs/outputs can be split and combined across the operations in a concrete decomposition.

The structure of this paper is as follows. In Section 2 we discuss the standard notion of refinement in Object-Z. In Section 3 we illustrate how we can refine the class structure of a specification and thus change the granularity of objects during development. Section 4 goes on to discuss the problem of decomposing operations in a refinement, and Section 5 looks at the solution.

2 Refinement in Object-Z

In this section we discuss how to refine an Object-Z specification. The definition of refinement in Object-Z is a simple adaption of refinement in Z. That is, refinement of a class A by a class C requires that we relate the two classes by a retrieve relation linking the state spaces in the abstract and concrete classes. Then the initial states of the classes must be related in the standard fashion and each abstract operation in class A must be matched by a concrete counterpart in C such that the preconditions are identical modulo the retrieve relation (but not weakened because of the meaning of preconditions in Object-Z), and the effect of the concrete is consistent with the behaviour of the abstract.

This assumes the specifications are conformal, i.e. for each abstract operation AOp there is exactly one concrete operation COp. It also assumes that the inputs and outputs of the concrete operation are identical to those of the abstract operation. For the moment we will assume this, but in Section 4 we will relax this condition when we decompose individual operations.

We can summarise the requirements of refinement in Object-Z in the following definition, where $A.\textsc{State}$ denotes the state schema in the class A etc, and $A.\textsc{Init}$ denotes the initialisation.

Definition 1 *Object-Z refinement*
An Object-Z class C is a refinement of the class A if there is a retrieve relation R such that every visible abstract operation AOp is recast into a visible concrete operation COp and the following hold.

1 $\forall C.\textsc{State} \bullet C.\textsc{Init} \Rightarrow (\exists A.\textsc{State} \bullet A.\textsc{Init} \land R)$

2 $\forall A.\textsc{State};\ C.\textsc{State} \bullet \mathrm{pre}\, AOp \land R \iff \mathrm{pre}\, COp$

3 $\forall A.\textsc{State};\ C.\textsc{State};\ C.\textsc{State}' \bullet R \land COp \Longrightarrow \exists A.\textsc{State}' \bullet R' \land AOp$

One difference between refinement in Z and refinement in Object-Z is in the treatment of preconditions of operations. Refinement in Object-Z requires that preconditions are preserved (condition 2 above), whereas in Z preconditions can be weakened under refinement. This arises due to different interpretations of a partial operation in Z and Object-Z. In Z the meaning of an operation specified as a partial relation is that it behaves as specified when used within its precondition, and outside its precondition, anything may happen. However, in Object-Z the operation is not enabled outside its precondition. Therefore guards cannot be modelled in Z since all operations are always enabled, whereas in Object-Z outside the precondition an operation will be refused.

An example of refinement in Object-Z using this definition appears in [11] where a restriction is made that classes do not contain object instantiations. In fact the above definition can serve as a definition of refinement between arbitrary Object-Z specifications (not just two classes) because each Object-Z specification has a main class through which the behaviour of the whole specification is viewed. One Object-Z specification is then a valid refinement of another if the main class of the first is a refinement of the main class of the second.

We look at two aspects of refinement in this paper. The first will be to see how we can refine the class structure of the specification by introducing a number of communicating objects in a refinement. Secondly we will relax the assumption of operation conformity made above. The starting point for this 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* [3] which allows inputs and outputs to change under a refinement in a controlled manner.

Notation
The schema calculus composition ($\fatsemi$) and piping ($\gg$) operators are used frequently. The composition operator acts as composition on the state spaces of two operations, so $COp_1 \fatsemi COp_2$ is a new operation formed by considering the operations as relations on the state space. In such a composition the declarations of COp_1 and COp_2 are merged. The piping operator acts as composition on the inputs and outputs of two operations, and thus can be seen as a one way communication of the outputs of the first into the inputs of the second operation. We also use the parallel composition operator, $\|$, which defines two way communication between two operations.

3 Decomposing objects in a refinement

We now show how we can use the definition of refinement given above to verify refinements where a class is split into a collection of interacting classes. By developing refinements such as these we can support a change of granularity of the objects in the system. Initial designs can then contain few objects, representing an abstract view of the system's functionality, and refinements can change this granularity by introducing perhaps many objects which reflect the structure of a final implementation.

In our example the initial specification consists of a single class ATM_0 which describes a cash point machine. The behaviour of the class is defined by operations *Insert_card*, *Passwd*, *Withdraw* etc. A user of the machine inserts a card which is modelled by reading the account number *account*?. A four-digit p.i.n. is then given, and if this matches the account then the user is able to *proceed* and *Withdraw* money.

The bank accounts are modelled as a partial function *accts* from account numbers to amounts. The p.i.n. for a given account m is given by $pins(m)$. The card currently inside the machine is represented by *card*, and we use a boolean to determine whether a transaction is allowed to *proceed*.

$$
\begin{array}{|l}
\hline
\textit{ATM}_0 \\
\hline
\begin{array}{|l}
\hline
accts : \mathbb{N} \pfun \mathbb{N} \\
pins : \mathbb{N} \pfun \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
card : \mathbb{N} \\
proceed : \mathbb{B} \\
\hline
\mathrm{dom}\ accts = \mathrm{dom}\ pins \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\hline
\textit{INIT} \\
\hline
card = 0 \land \neg proceed \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\hline
\textit{Insert\_card} \\
\hline
\Delta(card, proceed) \\
account? : \mathbb{N} \\
\hline
card = 0 \land account? \in \mathrm{dom}\ pins \\
\neg proceed' \\
card' = account? \\
\hline
\end{array}
\quad
\begin{array}{|l}
\hline
\textit{Passwd} \\
\hline
\Delta(proceed) \\
pin? : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
\hline
card \neq 0 \\
pins(card) = pin? \Rightarrow proceed' \\
pins(card) \neq pin? \Rightarrow \neg proceed' \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\hline
\textit{Withdraw} \\
\hline
\Delta(accts) \\
money! : \mathbb{N} \\
\hline
proceed \\
accts' = accts \oplus \{card \mapsto accts(card) - money!\} \\
\hline
\end{array} \\
\hline
\end{array}
$$

In an implementation the branch of a user is not necessarily co-located with the cash point being used, and we therefore want to split this description into two separate classes: *CashPoint* and *Bank*. These are given as follows.

$$
\begin{array}{|l}
\hline
\textit{CashPoint} \\
\hline
\begin{array}{|l}
\hline
pins : \mathbb{N} \pfun \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
card : \mathbb{N} \\
proceed : \mathbb{B} \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\hline
\textit{INIT} \\
\hline
card = 0 \land \neg proceed \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\hline
\textit{Insert\_card} \\
\hline
\Delta(card, proceed) \\
account? : \mathbb{N} \\
\hline
card = 0 \land account? \in \mathrm{dom}\ pins \\
\neg proceed' \\
card' = account? \\
\hline
\end{array}
\quad
\begin{array}{|l}
\hline
\textit{Passwd} \\
\hline
\Delta(proceed) \\
pin? : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
\hline
card \neq 0 \\
pins(card) = pin? \Rightarrow proceed' \\
pins(card) \neq pin? \Rightarrow \neg proceed' \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\hline
\textit{Proceed} \\
\hline
proceed! : \mathbb{B} \\
acct! : \mathbb{N} \\
\hline
proceed! = proceed \\
acct! = card \\
\hline
\end{array} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Bank} \\
\quad \begin{array}{|l}
\hline
accts : \mathbb{N} \nrightarrow \mathbb{N} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{Withdraw} \\
\Delta(accts) \\
money! : \mathbb{N} \\
acct? : \mathbb{N} \\
\hline
accts' = accts \oplus \{acct? \mapsto accts(acct?) - money!\} \\
\hline
\end{array} \\
\hline
\end{array}
$$

The complete behaviour, including communication between the two components, is given by the main class ATM_1. This includes a bank and a cash point, and it promotes operations from these objects to the overall class. A customer can then *Withdraw* from the bank when the cash point gives permission to proceed and has communicated which $acct!$ permission is being granted for. The communication $\|$ then identifies $acct!$ with $acct?$ in *Withdraw* and the correct account is debited.

$$
\begin{array}{|l}
\hline
ATM_1 \\
\quad \begin{array}{|l}
\hline
c : CashPoint \\
b : Bank \\
\hline
\mathrm{dom}\; b.accts = \mathrm{dom}\; c.pins \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{INIT} \\
c.\textit{INIT} \\
\hline
\end{array} \\
Insert\_card \mathrel{\widehat=} c.Insert\_card \\
Passwd \mathrel{\widehat=} c.Passwd \\
Withdraw \mathrel{\widehat=} (b.Withdraw \| c.Proceed \bullet [proceed! = true]) \setminus \{proceed!\} \\
\hline
\end{array}
$$

Then we claim that ATM_1 is a refinement of the class ATM_0. To show this formally we need to define a retrieve relation between the two main classes, and we use the following, where we have prefixed each variable name by the name of the class to keep the variable names distinct.

$$
\begin{array}{|l}
\hline
R \\
ATM_0.\textit{STATE} \\
ATM_1.\textit{STATE} \\
\hline
ATM_0.accts = ATM_1.b.accts \\
ATM_0.pins = ATM_1.c.pins \\
ATM_0.proceed = ATM_1.c.proceed \\
ATM_0.card = ATM_1.c.card \\
\hline
\end{array}
$$

We then have to show that the conditions in Definition 1 hold. We concentrate on the conditions for the operations here, and for every operation we have to verify correctness and applicability. For example, for the *Insert_card* operation we need to verify that

$$
\begin{array}{l}
\mathrm{pre}\; ATM_0.Insert\_card \land R \iff \mathrm{pre}\; ATM_1.Insert\_card \\
R \land ATM_1.Insert\_card \implies \exists (ATM_0.\textit{STATE})' \bullet R' \land ATM_0.Insert\_card
\end{array}
$$

In order to do this we note that the use of object instantiation (e.g. banks and cash points) means that we need a way of interpreting the preconditions, the state spaces and the retrieve relation in their presence. Once we have done that the verification should be straightforward.

For the state space we define $(ATM_0.\textit{STATE})'$ to be $ATM_0.(\textit{STATE})'$, with the process being applied recursively if there are further object instantiations. This allows the refinement conditions to access the state variables of an object used in a refinement, and this is necessary since it is the state variables that alter when an operation is invoked upon that object.

Thus $ATM_0.\textit{STATE}'$ and $ATM_1.\textit{STATE}'$ will be

$$
\begin{array}{|l}
\hline
ATM_0.\textit{STATE}' \\
\hline
accts' : \mathbb{N} \pfun \mathbb{N} \\
pins' : \mathbb{N} \pfun \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
card' : \mathbb{N} \\
proceed' : \mathbb{B} \\
\hline
\mathrm{dom}\, accts' = \mathrm{dom}\, pins' \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
ATM_1.\textit{STATE}' \\
\hline
c.\textit{STATE}' \\
b.\textit{STATE}' \\
\hline
\mathrm{dom}\, b.accts' = \mathrm{dom}\, c.pins' \\
\hline
\end{array}
$$

The latter being:

$$
\begin{array}{|l}
\hline
b.accts' : \mathbb{N} \pfun \mathbb{N} \\
c.pins' : \mathbb{N} \pfun \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
c.card' : \mathbb{N} \\
c.proceed' : \mathbb{B} \\
\hline
\mathrm{dom}\, b.accts' = \mathrm{dom}\, c.pins' \\
\hline
\end{array}
$$

This interpretation is what we require. Without it $(ATM_1.\textit{STATE})'$ would be

$$
\begin{array}{|l}
\hline
c' : CashPoint \\
b' : Bank \\
\hline
\mathrm{dom}\, b'.accts = \mathrm{dom}\, c'.pins \\
\hline
\end{array}
$$

However, this is incorrect because when the object b evolves its internal state changes but the references to it is still denoted by b and not by b'. So $(ATM_1.\textit{STATE})'$ must contain primed references to the components actually altered by the operations, i.e. $b.accts'$ and not b' itself.

This definition then allows us to interpret the state spaces in the schema R' so that it can be used in the refinement conditions as desired. We also need to calculate preconditions such as $ATM_1.Insert\_card$, that is to be able to calculate $\mathrm{pre}\, ATM_1.c.Insert\_card$. In order to do this we need to define $\mathrm{pre}\, a.Op$ for an object a with operation Op. $\mathrm{pre}\, a.Op$ should be a schema representing those states where it is possible to apply Op on a. Since this depends on the current state of a, we define $\mathrm{pre}\, a.Op$ to be $a.\mathrm{pre}\, Op$, as the latter defines a schema representing states in a when Op is applicable.

Definition 2 *Let A be an Object-Z class containing an operation Op. Suppose that the state declaration $a : A$ appears in another class, then in that class $\mathrm{pre}\, a.Op$ is defined to be $a.\mathrm{pre}\, Op$.*

With these definitions in place we can verify the refinement conditions above. For example, applicability for $Insert\_card$ now requires that

$$
\begin{array}{l}
(card = 0 \land account? \in \mathrm{dom}\, pins) \land (pins = c.pins \land card = c.card) \iff \\
\qquad c.card = 0 \land account? \in \mathrm{dom}\, c.pins
\end{array}
$$

Similarly correctness requires that

$$
\begin{array}{l}
(pins = c.pins \wedge card = c.card \wedge \ldots) \wedge \\
(c.card = 0 \wedge account? \in \operatorname{dom} c.pins \wedge \\
\neg c.proceed' \wedge \\
c.card' = account?) \\
\implies \exists\, ATM_0.\mathit{STATE}' \bullet (pins' = c.pins' \wedge card' = c.card' \wedge \ldots) \wedge \\
\qquad\qquad (card = 0 \wedge account? \in \operatorname{dom} pins \wedge \\
\qquad\qquad \neg proceed' \wedge \\
\qquad\qquad card' = account?)
\end{array}
$$

which are easily seen to be true. The conditions for the other operations are similar.

To summarise, what we have done here is to use the basic refinement conditions to verify refinements where we change the granularity of the objects in the specification. We have done this by using the standard conditions, but interpreting them appropriately in the presence of object instantiation.

4 Decomposing operations in a refinement

Having decomposed our initial single object into a number of communicating objects we will now look at the issue of decomposing individual operations, i.e. look at action refinement in Object-Z. The situation we wish to provide support for is an abstract specification containing an operation AOp, and a subsequent refinement where AOp is implemented as a sequence of concrete operations: COp_1 followed by COp_2.

The assumption of conformity in the standard refinement rules means that there is one concrete operation for each abstract operation. If we are to decompose AOp into $COp_1 \mathbin{⨾} COp_2$, then one choice we could make would be for COp_1 to refine AOp, and for COp_2 to refine a stuttering step $skip$, which makes no change in the abstract state. This preserves the conformity of the two specifications and so simple action refinements can be verified as two sets of rules, one for refining AOp into COp_1 and the other for refining $skip$ into COp_2.

The requirements on COp_1 refining AOp are the standard ones and the requirements on refining $skip$ to COp_2 are:

$$
\begin{array}{l}
\forall\, A.\mathit{STATE};\ C.\mathit{STATE} \bullet R \Leftrightarrow \operatorname{pre} COp_2 \\
\forall\, A.\mathit{STATE};\ C.\mathit{STATE};\ C.\mathit{STATE}' \bullet R \wedge COp_2 \Rightarrow \exists\, A.\mathit{STATE}' \bullet \Xi A.\mathit{STATE} \wedge R'
\end{array}
$$

It is in this context that the action refinements given in [14, 5] are verified.

However, we wish to go further than this and we want to be able to split the input and output across the concrete decomposition. If we do so then the operations in the concrete decomposition will not correspond to abstract $skips$. An example will make it clear why this is the case.

Consider the *CashPoint*. The *Passwd* operation accepted the pin in one go, but in an implementation we wish to accept the 4 digits one at a time, with each digit being entered by a separate operation. We are therefore going to split this single operation into a sequence consisting of 4 operations: *First*, *Second*, *Third* and *Fourth*. Note that, as usual, the denial to proceed is only flagged after all the digits have been inserted. (The definitions of *Insert_card* and *Proceed* are as before and are elided.)

$$\boxed{\begin{array}{l}
\textit{CashPoint}_1 \\[4pt]
\begin{array}{|l}
pins : \mathbb{N} \pfun \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
card : \mathbb{N} \\
proceed : \mathbb{B} \\
temp : \mathrm{seq}_4\, \mathbb{N}
\end{array} \\[6pt]
\textit{INIT} \\
\quad card = 0 \land temp = \langle\,\rangle \\[6pt]
Insert_card \mathrel{\widehat=} \ldots \\
Proceed \mathrel{\widehat=} \ldots \\[6pt]
\begin{array}{ll}
\begin{array}{|l}
\multicolumn{1}{l}{First} \\
\Delta(temp) \\
d? : \mathbb{N} \\
\hline
card \neq 0 \\
temp = \langle\,\rangle \land temp' = \langle d? \rangle
\end{array}
&
\begin{array}{|l}
\multicolumn{1}{l}{Second} \\
\Delta(temp) \\
d? : \mathbb{N} \\
\hline
\#temp = 1 \land temp' = temp \frown \langle d? \rangle
\end{array}
\\[6pt]
\begin{array}{|l}
\multicolumn{1}{l}{Third} \\
\Delta(temp) \\
d? : \mathbb{N} \\
\hline
\#temp = 2 \land temp' = temp \frown \langle d? \rangle
\end{array}
&
\begin{array}{|l}
\multicolumn{1}{l}{Fourth} \\
\Delta(proceed) \\
d? : \mathbb{N} \\
\hline
\#temp = 3 \\
pins(card) = temp \frown \langle d? \rangle \Rightarrow proceed' \\
pins(card) \neq temp \frown \langle d? \rangle \Rightarrow \neg proceed'
\end{array}
\end{array}
\end{array}}$$

We would like to view this as a refinement of *CashPoint*. However, the refinement cannot be verified by making of one of the concrete operations refine *Passwd* whilst the others refine *skip*.

The problem with the refinement is the following. Clearly the retrieve relation has to be the identity on the state spaces, and therefore abstract *skip* operations can be refined by concrete operations which only change *temp*. Therefore *First*, for example, looks a suitable candidate to refine *skip*. However, *First* (and all the other operations) consumes input, therefore it alters the state (as a result of changing the environment) and thus does not correspond to *skip*. The problem is that the inputs of *Passwd* are distributed throughout the concrete operations. This means that the rules as they stand can't be used to verify a refinement in general. The rest of this paper is devoted to developing the necessary machinery to solve this problem by adapting action refinement rules developed for Z [6].

5 Action refinement

In this section we will consider refinements where we decompose an abstract operation into a sequence of concrete operations without requiring that any of the concrete operations refine an abstract *skip* operation. This will solve the problems identified in the example above.

To do so we will apply and extend the technique of IO-refinement to address the issue of decomposing inputs and outputs across a sequence of concrete operations. Our example illustrates what we wish to achieve: *Passwd* takes an input $pin? : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ and breaks it down into single inputs $d? : \mathbb{N}$ provided a number of times in the concrete operations.

To solve this problem we drop the condition that when we decompose AOp into $COp_1 \mathbin{;} COp_2$ one of the concrete operations refines AOp and the other refines *skip*. This means that AOp can be replaced by COp_1 followed by COp_2, however, if the concrete object performs a single instance of COp_1 (not followed by COp_2) then this won't necessarily simulate anything at the abstract level. Indeed this may be viewed as exhibiting a deficiency of the abstract cash machine specification: it does not represent the (realistic) situation where a user only ever enters one of the digits of their p.i.n..

This is the situation in our example. *Passwd* is refined by the sequence $First \fatsemi Second \fatsemi Third \fatsemi Fourth$, however, a single instance of *First* does not correspond on its own to any abstract level operation, but only a bit of *Passwd*'s functionality.

With the single requirement that AOp is refined by $COp_1 \fatsemi COp_2$ we can now describe the conditions necessary for a refinement to hold. Without considering any input and output transformations at this stage the formulation is as follows (we omit consideration of the initialisation condition as that is unaltered).

Definition 3 *Action refinement without IO transformations*
The retrieve relation R defines an action refinement between two classes (where the abstract operation AOp has been decomposed into a sequence of concrete operations $COp_1 \fatsemi COp_2$) if the following hold.

$$\forall A.\mathit{STATE};\ C.\mathit{STATE};\ C.\mathit{STATE}' \bullet (COp_1 \fatsemi COp_2) \land R \Rightarrow \exists A.\mathit{STATE}' \bullet R' \land AOp$$
$$\forall A.\mathit{STATE};\ C.\mathit{STATE} \bullet \mathit{pre}\, AOp \land R \Leftrightarrow \mathit{pre}\, COp_1$$
$$\forall A.\mathit{STATE};\ C.\mathit{STATE} \bullet \mathit{pre}\, AOp \land R \land COp_1 \Rightarrow \mathit{pre}\, COp_2$$

These conditions generalise to an action refinement with an arbitrary number of concrete operations in the obvious manner.

The conditions in Definition 3 express the following requirements. The first says that the effect of $COp_1 \fatsemi COp_2$ is consistent with that of AOp (but can of course reduce any non-determinism in AOp). The second says that COp_1 can be invoked precisely when AOp can be, and the third says that when (and only when) COp_1 has been completed COp_2 can be invoked. Informally these are clearly the correct conditions for a refinement of AOp into $COp_1 \fatsemi COp_2$. Formally they can be derived from the relational basis of refinement in the same way that [6] gives the relational basis for action refinement in Z.

This definition is sufficient to verify the applicability conditions for the example above, but for correctness we need some input transformations. To see this note that if we calculate ($First \fatsemi Second \fatsemi Third \fatsemi Fourth$) we find that we have lost the differentiation between the inputs in the concrete operations, i.e. this schema composition results in:

$$\begin{array}{|l}
\Delta(\mathit{proceed}) \\
d? : \mathbb{N} \\
\hline
\mathit{card} \neq 0 \\
\mathit{temp} = \langle\, \rangle \\
\mathit{pins}(\mathit{card}) = \langle d?, d?, d?, d? \rangle \Rightarrow \mathit{proceed}' \\
\mathit{pins}(\mathit{card}) \neq \langle d?, d?, d?, d? \rangle \Rightarrow \neg \mathit{proceed}'
\end{array}$$

which cover only very limited range of 4-digit p.i.n. codes, and is certainly not the same as *Passwd*.

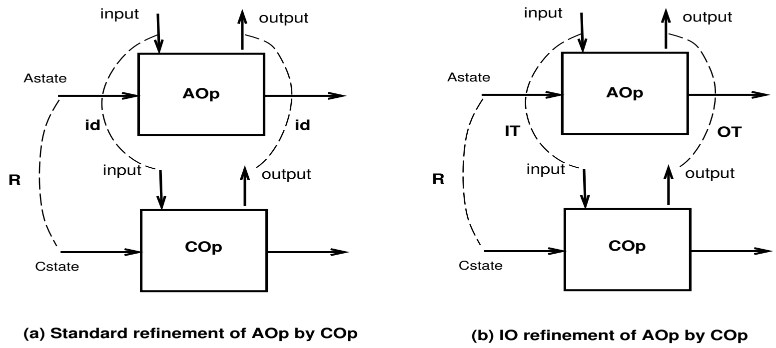


Figure 1: Standard and IO refinement of operations

5.1 Altering the Input and Output

We now consider the transformations of input and output that are needed to support action refinements. To do so we will use IO refinement in our action refinements to produce a set of conditions that allow inputs and outputs to be distributed throughout a concrete decomposition.

IO refinement [3, 13] is a generalisation of the standard Z refinement rules, which are normally given in terms of a retrieve relation R together with two identities id between the inputs and outputs (see Figure 1(a)). In order to allow the types of inputs and outputs to change, IO refinement replaces the identities id 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 (see Figure 1(b)).

It is necessary to impose some conditions on IT and OT. The first is that we require that IT and OT are total on the abstract input and output types. This ensures that every abstract input can be used in the concrete operation. The second is that OT must be injective. This means that different abstract ("original") outputs can be distinguished in the concrete cases because their concrete representations will be different as well.

IT and OT are written as schemas and 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 ($\gg$). To do so 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.

For an IO refinement of one abstract operation into one concrete operation the following definition is used.

Definition 4 *IO refinement for a single operation*
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 (initialisation is as before):

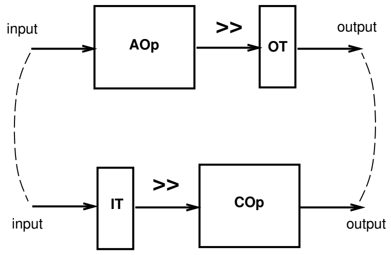
Applicability: $\forall A.\mathit{STATE};\ C.\mathit{STATE} \bullet pre(\overline{IT} \gg AOp) \wedge R \Leftrightarrow pre\, COp$

Correctness:

$$\forall A.\mathit{State};\ C.\mathit{State};\ C.\mathit{State}' \bullet R \wedge (IT \gg COp) \Rightarrow \exists A.\mathit{State}' \bullet R' \wedge (AOp \gg OT)$$

The correctness criteria requires that COp with the input transformation should produce a result related by R and the output transformation to one that AOp could have produced.

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 act as wrappers to a concrete operation, converting abstract inputs to concrete ones and similarly for the output.



5.2 Applying IO transformations to action refinement

IO refinement is a mechanism to refine the inputs/outputs of one abstract operation into one concrete operation. To apply it fully to action refinements we need to be able to spread the inputs/outputs throughout a sequence of concrete operations. Because of this there will not necessarily be a 1-1 mapping between the number of abstract inputs/outputs and the number of concrete inputs/outputs.

The principal hurdle to overcome is how to spread the abstract inputs/outputs throughout a sequence of concrete operations. To solve this problem we will generalise IO refinement slightly along the following lines. IO refinement is defined as a condition between one abstract and one concrete operation, because of that we used a simple transformer IT. In order to decompose one abstract operation into a sequence of concrete operations we need to use a mapping between an abstract input and a sequence of concrete inputs representing the inputs needed in the decomposition. Therefore we will generalise our transformers IT and OT to produce a sequence of inputs and outputs instead of just one.

For example, for the *Passwd* operation and its decomposition, the input transformer we use is the following.

IT

$pin? : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$
$d! : \mathrm{seq}_4\, \mathbb{N}$

$pin? = (d!.1, d!.2, d!.3, d!.4)$

This takes in a single input $pin?$ for the abstract operation *Passwd* and maps it to a sequence for use by the concrete operations. The concrete operations then use this sequence one by one. Therefore the abstract input $pin?$ has been decomposed and spread throughout the concrete sequence as desired. The input and output transformer machinery is necessary in order that we can express the intuitive requirements formally in the schema calculus. There are no outputs in this particular example so the output transformer is the identity. We then apply the ideas from IO refinement to the definition of action refinement, this results in correctness criteria such as

$$\begin{array}{l}(IT \gg (First[d!.1/d?] \mathbin{;} Second[d!.2/d?] \mathbin{;} Third[d!.3/d?] \mathbin{;} Fourth[d!.4/d?])) \wedge R \implies \\ \qquad\qquad \exists\, CashPoint.\mathit{STATE}' \bullet R' \wedge Passwd\end{array}$$

The substitutions, e.g. $[d!.1/d?]$, are used in order to restore the distinction between the individual inputs. Such substitutions allow us to describe the process of consuming the inputs one by one explicitly in the operations. This correctness condition thus asks that with the input wrapper IT the concrete sequence is a refinement of $Passwd$.

We now consider the general case of a decomposition of AOp into $COp_1 \mathbin{;} COp_2$. The general formalisation combines the three conditions needed for a action refinement of AOp into $COp_1 \mathbin{;} COp_2$ with the use of input and output transformers from IO refinement. The refinement will be defined by three parts: a retrieve relation between the state spaces together with transformers IT and OT which map abstract inputs/outputs to a sequence of concrete inputs/outputs. We can now combine action refinement (definition 3) with IO refinement (definition 4) to produce the general conditions for action refinement with IO transformations.

The following definition expresses the refinement of AOp into a fixed sequence $COp_1 \mathbin{;} COp_2$. In fact explicit substitutions (as in $First$ above) are only necessary when the decomposition of AOp involves more than one occurrence of the same input or output parameter names in the concrete operations. If COp_1 and COp_2 are distinct operations with distinct parameter names then the formalisation is simplified by the omission of the substitutions.

Definition 5 *Non-atomic refinement with IO transformations*
Let IT be an input transformer for $COp_1 \mathbin{;} COp_2$ which is total on the abstract inputs. Let OT be a total injective output transformer for AOp. The retrieve relation R defines a action IO refinement if:

$$\begin{array}{l}\forall\, A.\mathit{STATE};\ C.\mathit{STATE};\ C.\mathit{STATE}' \bullet (IT \gg COp_1 \mathbin{;} COp_2) \wedge R \Rightarrow \exists\, A.\mathit{STATE}' \bullet R' \wedge (AOp \gg OT) \\ \forall\, A.\mathit{STATE};\ C.\mathit{STATE} \bullet pre(\overline{IT} \gg AOp) \wedge R \Leftrightarrow pre\, COp_1 \\ \forall\, A.\mathit{STATE};\ C.\mathit{STATE} \bullet pre(\overline{IT} \gg AOp) \wedge R \wedge (IT \gg COp_1) \Rightarrow pre\, COp_2\end{array}$$

The conditions in full for the $Passwd$ operation then become:

$$\begin{array}{l}(IT \gg (First[d!.1/d?] \mathbin{;} Second[d!.2/d?] \mathbin{;} Third[d!.3/d?] \mathbin{;} Fourth[d!.4/d?])) \wedge R \implies \\ \qquad\qquad \exists\, CashPoint.\mathit{STATE}' \bullet R' \wedge Passwd \\ \mathrm{pre}(\overline{IT} \gg Passwd) \wedge R \Leftrightarrow \mathrm{pre}\, First \\ \mathrm{pre}(\overline{IT} \gg Passwd) \wedge R \wedge (IT \gg First[d!.1/d?]) \Rightarrow \mathrm{pre}\, Second \\ \mathrm{pre}(\overline{IT} \gg Passwd) \wedge R \wedge (IT \gg First[d!.1/d?] \mathbin{;} Second[d!.2/d?]) \Rightarrow \mathrm{pre}\, Third \\ \mathrm{pre}(\overline{IT} \gg Passwd) \wedge R \wedge (IT \gg First[d!.1/d?] \mathbin{;} Second[d!.2/d?] \mathbin{;} Third[d!.3/d?]) \Rightarrow \mathrm{pre}\, Fourth\end{array}$$

which are easily verified. Therefore we have done what we set out to achieve. That is, define a set of action refinement rules which are a generalisation of standard refinement but allow the full decomposition of operations including inputs and outputs.

6 Conclusions

In this paper we have looked at how we can change the granularity of both objects and operations in an Object-Z refinement step. To change the granularity of objects we view refinement as refining the main class of the specification under consideration. By interpreting the state spaces and retrieve relation appropriately in the presence of object instantiation we can refine a single class into a number of communicating classes.

To change the granularity of operations we dispensed with the requirement that every concrete operation corresponds to an abstract one. This enabled quite general refinements to be derived. 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.

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

[5] 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.

[6] J. Derrick and E.A. Boiten. Non-atomic refinement in Z. In J.M. Wing, J.C.P. Woodcock, and J. Davies, editors, *FM'99 World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *LNCS*, pages 1477–1496, Berlin, 1999. Springer.

[7] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, September 1995.

[8] 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.

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

[10] Arend Rensink and Roberto Gorrieri. Action refinement as an implementation relation. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[11] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M. Hinchey and Shaoying Liu, editors, *First IEEE International Conference on Formal Engineering Methods (ICFEM '97)*, pages 293–302, Hiroshima, Japan, November 1997. IEEE Computer Society.

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

[13] 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.

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

[15] J. C. P. Woodcock and C. C. Morgan. Refinement of state-based concurrent systems. In D. Bjorner, C. A. R. Hoare, and H. Langmaack, editors, *VDM '90 VDM and Z - Formal Methods in Software Development*, LNCS 428, pages 340–351, Kiel, FRG, April 1990. Springer-Verlag.