

Chapter 2

Simple Refinement

We use the term *simple refinement* to describe refinements of operations (and collections of operations) where the state schema does not change. These rules apply to ADTs (as introduced in Chap. 1), and some even apply to “concrete” data types, which we will call *repertoires*.

We first present a semi-formal motivation of the simplest refinement relation of all, *viz. operation refinement*. Operation refinement can be applied to individual operations without reference to the other operations present in the ADT, and will be presented in detail in Sect. 2.2. The other simple refinement rules, *establishing* and *imposing invariants*, which we will discuss in Sect. 2.4, only apply in the context of a full ADT, and require abstraction, in the sense of the state not being observable.

2.1 What Is Refinement?

Through this book we hope to convey to our readers that refinement in a sufficiently flexible and expressive notation (like Z or Object-Z) forms a rich and interesting subject of study. Research into this area has certainly unearthed a variety of complex issues. The first definition of refinement in Z (Definition 2.5) contains some 10 different symbols, and quantifies (implicitly or explicitly) over some 5 variables, and these numbers are only increased by later definitions. All this seems to suggest that refinement is an inherently complex notion.

However, that is not really the case. The basic idea of refinement is fortunately a simple one—fortunately because any theory that starts from a set of complex definitions is likely to have a very tenuous intuitive foundation. The intuition behind refinement is just the following:

Principle of Substitutivity: it is acceptable to replace one program by another, *provided* it is impossible for a user of the programs to observe that the substitution has taken place. If a program can be acceptably substituted by another, then the second program is said to be a *refinement* of the first.

For example, replacing the current scheduling of trains in South-East England with one in which all trains run according to the timetable would be a valid refine-

ment. Of course some suspicion about a substitution having taken place would still arise, but a casual train traveller could never be entirely sure of that. This is due to the fact that all trains running on time is also an imaginable behaviour of the current system.¹

Any definition of refinement presented in this book should be validated. This tends to involve a proof that previous definition(s) are a special case of the new definition. Ideally, however, it should also include a demonstration that the refinement relation is based on a particular principle of substitutivity.

Note that, although it is as yet informal, the principle of substitutivity strongly suggests some properties of refinement. One way of guaranteeing that no difference can be observed is by substituting a program by itself. This implies that refinement will be *reflexive*. Two substitutions in a row, both of which are impossible to observe, would constitute another example of an unobservable substitution. So refinement will be *transitive* as well, allowing for “step-wise” refinement. It is not likely to be symmetric or anti-symmetric, though. Often we will present *refinement-equivalent* specifications, which may not be equal but which can be safely substituted for each other in both directions. Finally, it is desirable that refinement is a pre-congruence² or “compositional”: if a component on its own can be substituted by another, this substitution should also be acceptable when the component is part of a larger system. This allows for “piece-wise” refinement.

(So, if refinement is a simple notion in principle, where does the complexity of the refinement rules come from? In our view, there is a trade-off between abstract relational “point-free” characterisations, and notations like Z which tend to name every object being related. Relational characterisations can sometimes be very short and are then easier to manipulate, but certainly once they encode formulas containing multiple occurrences of the same variable, they quickly become less comprehensible. Conversely, the explicit naming and quantification in Z provide much “formal noise” for simple formulas, but make it easier to comprehend more complicated formulas piece by piece. For these reasons, Chap. 3 will define the central notions of data refinement in a relational setting, which will only get interpreted and derived in Z in Chap. 4.)

Of course the informal definition of refinement above is incomplete. It does not define programs, it does not define what a “user” is allowed to do, or what constitutes an “observation”. These are all questions with different possible answers, leading to different formal definitions of refinement later on. For the moment, we remain semi-formal:

- a *program* is a finite sequence of *operations*, interpreted as their *sequential composition*;

¹On the other hand, replacing the machine that performs the weekly lottery draw with one which returns the same set of numbers every week is probably not an acceptable refinement. This requires a more sophisticated explanation, however, concerning the difference between non-deterministic and probabilistic choice, which is outside the scope of this book—see [5] for a comprehensive treatment.

²A partial order \leq is a pre-congruence if for all contexts $C[\cdot]$, whenever $x \leq y$ also $C[x] \leq C[y]$.

- an *operation* is a binary relation over the state space of interest, taken from a fixed collection (indexed set) of such operations. If that state space is represented by the Z schema *State*, then operations are represented by schemas over $\Delta State$;
- an *observation* is a pair of states: the state before execution of a program, and a state after.

In this interpretation, the notion of *repertoire* (or *interface*, *signature*, or *alphabet*) of a system is evident.

Definition 2.1 (Repertoire) A repertoire is an indexed collection of operations $(Op_i)_{i \in I}$ over the same state. \square

Because the observations for these repertoires are pairs of states, we could also view repertoires as *concrete* data types. We did not introduce repertoires in the previous chapter, because their function in this book is mostly as an intermediate stage for the introduction and explanation of *abstract* data types.

Given two repertoires, we are not normally interested in replacing a program using the one with an *arbitrary* program using the other. Rather, we are interested in the relation between two programs of the *same structure*, but using the corresponding operations from each of the two repertoires in corresponding places. This is formalised in the following definitions, which presuppose a definition of refinement between programs, to be given later.

Definition 2.2 (Conformal repertoires) Two repertoires are conformal iff their operations are indexed by the same index set, and defined over the same state. \square

Definition 2.3 (Repertoire refinement) For conformal repertoires $A = (AOp_i)_{i \in I}$ and $C = (COp_i)_{i \in I}$, C is a repertoire refinement of A iff for every finite sequence $\langle i_1, i_2, \dots, i_n \rangle$ the program $\langle COp_{i_1}, COp_{i_2}, \dots, COp_{i_n} \rangle$ is a refinement of $\langle AOp_{i_1}, AOp_{i_2}, \dots, AOp_{i_n} \rangle$. \square

However, this is not a very practical definition, as it involves quantification over all finite sequences. The only plausible way of proving such a refinement would be by structural induction, which however requires an extra condition for it to work. The base case³ of such an induction could be found in the following definition.

Definition 2.4 (Operation-wise refinement) Given two repertoires $A = (AOp_i)_{i \in I}$ and $C = (COp_i)_{i \in I}$, both operating over the same state. C is an operation-wise refinement of A iff for all $i \in I$, the program $\langle COp_i \rangle$ is a refinement of $\langle AOp_i \rangle$. \square

The validity of the induction step depends on the condition in the following theorem.

³Minus the obvious case for the empty sequence.

Theorem 2.1 *If sequential composition is monotonic with respect to refinement, i.e., (using \circ for composition and \sqsubseteq for refinement)*

$$S_1 \sqsubseteq S_2 \wedge T_1 \sqsubseteq T_2 \Rightarrow S_1 \circ T_1 \sqsubseteq S_2 \circ T_2$$

then repertoire refinement and operation-wise refinement coincide.

Proof

- Repertoire refinement obviously implies operation-wise refinement, which considers only the programs of length 1.
- Operation-wise refinement implies repertoire refinement, by induction over the program sequences. Monotonicity allows the induction step. \square

Monotonicity of program constructors with respect to refinement has been listed before as a desirable property: it makes refinement a pre-congruence, and thus allows “piecewise” refinement.

However, the definition of refinement between programs is at this stage still missing, as we have not made precise what the possible observations are.

2.2 Operation Refinement

In this section we motivate the traditional definition of operation refinement in \mathcal{Z} by making precise what we mean by “observing” a program—in particular, a program consisting of a single operation.

Recall that operations and programs are binary relations over a state space, and observations are, for the moment, pairs of concrete states. The most obvious connection between those is to interpret the set of observations as a relation itself, which should be equal to the operation. This is perfectly acceptable when the operation is total. When it is not, however, it would be impossible to discuss observations starting from a state which is outside the operation’s domain—whereas the traditional \mathcal{Z} approach *does* allow such observations.

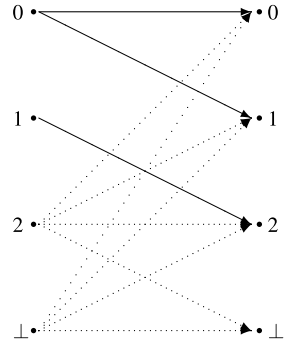
An operation allows two kinds of possible observations (which are illustrated in Fig. 2.1):

- a before-state and an after-state which are related by the operation;
- a before-state which is not in the domain of the operation, with an *arbitrary* after-state. For technical reasons, this after-state may also be the distinguished state \perp , representing a non-terminating computation; \perp as a before-state is also included in this case.

As a consequence, observing the original AOp and the substituting operation COp from a particular before-state s :

- if s was in the domain of AOp , then the after-state for COp should always be one of those possible according to AOp . This in turn means that s should be in the

Fig. 2.1 Observations of $Op = \{(0, 0), (0, 1), (1, 2)\}$ over the state $\{0, 1, 2\}$. The *dotted lines* indicate the observations from before-states outside the domain



domain of COp (or else \perp , which AOp does not allow, would be observable in COp);

- if s was not in the domain of AOp , any collection of possible observations for COp would be acceptable⁴—so s may be in the domain of COp , but it may not be.

For the definitions of refinement rules which follow now, we will assume that operations are represented by schemas in Z . When we take a more formal approach (starting from the next chapter), we will introduce a clear separation between operations as binary relations and operations as schemas, and indicate (in Chap. 4) precisely how they are related.

Definition 2.5 (Operation refinement) An operation COp is an operation refinement of an operation AOp over the same state space $State$ iff

Correctness

$$\forall State; State' \bullet \text{pre } AOp \wedge COp \Rightarrow AOp$$

Applicability

$$\forall State \bullet \text{pre } AOp \Rightarrow \text{pre } COp$$

□

At the level of binary relations, operation refinement coincides with inclusion of relations only when their domains are equal:

Lemma 2.1 If $\text{pre } AOp = \text{pre } COp$, then COp is an operation refinement of AOp iff

$$\forall State; State' \bullet COp \Rightarrow AOp$$

Proof Applicability is clearly satisfied under the given condition. The antecedent of correctness can be simplified using the tautology $Op \wedge \text{pre } Op = Op$. □

⁴Note, however, that given this definition of observations, it is impossible to define COp such that (x, \perp) is a possible observation but not (x, y) for some y .

Lemma 2.2 *If $\text{pre } COp = \text{State}$, i.e., COp is total on State , then AOp is an operation refinement of COp if*

$$\forall \text{State}; \text{State}' \bullet \text{pre } AOp \wedge COp \Rightarrow AOp$$

Proof Applicability holds trivially as its conclusion is true, so only correctness remains. \square

In some applications (cf. Chaps. 15 and 18) equality of domains is always required in operation refinement, due to a different interpretation of partial operations.

Theorem 2.2 *Operation refinement is a partial order on operations over the same state, i.e. it is transitive and reflexive.*

Proof Reflexivity is obvious from the conditions. Assume BOp is an operation refinement of AOp , and COp is an operation refinement of BOp . Now we need to prove that COp is an operation refinement of AOp . Applicability is trivial, as it follows from transitivity of \Rightarrow . For correctness we have:

$$\begin{aligned} & \text{pre } AOp \wedge COp \\ & \Rightarrow \{\text{applicability } AOp \text{ to } BOp\} \\ & \text{pre } AOp \wedge \text{pre } BOp \wedge COp \\ & \Rightarrow \{\text{correctness } BOp \text{ to } COp\} \\ & \text{pre } AOp \wedge BOp \\ & \Rightarrow \{\text{correctness } AOp \text{ to } BOp\} \\ & AOp \end{aligned}$$

\square

Operation refinements can be obtained through schema conjunction. This automatically guarantees correctness, and applicability amounts to the domain of the operation being preserved. This is expressed in the following lemma.

Lemma 2.3 *The operation $AOp \wedge X$, for X and AOp operations over ΔState , is an operation refinement of AOp iff*

$$\forall \text{State} \bullet \text{pre } AOp \Rightarrow \text{pre}(AOp \wedge X)$$

\square

So far we have only discussed operation refinement from the perspectives of repertoires, which form a sort of *concrete* data types (as their state is observable). Shortly, we will be focusing on ADTs, whose state space is *not* observable. Consequently, in order to make observations of ADT programs, we will need to introduce *inputs* and *outputs*. Operation refinement is also meaningful in the presence of inputs and outputs, which must form part of the observation. As a consequence of that, they cannot change in operation refinement, and must be simply added to the quantification.

Definition 2.6 (Operation refinement with inputs and outputs) An operation COp is an operation refinement of an operation AOp over the same state space $State$ and with the same inputs $?AOp$ and the same outputs $!AOp$ iff

Correctness

$$\forall State; State'; ?AOp; !AOp \bullet \text{pre } AOp \wedge COp \Rightarrow AOp$$

Applicability

$$\forall State; ?AOp \bullet \text{pre } AOp \Rightarrow \text{pre } COp$$

□

If we look back at Example 1.5 we can see that these definitions of operation refinement tackle two out of the three issues we highlighted, namely allowing preconditions to be weakened and non-determinism to be reduced. In particular, applicability requires a concrete operation to be defined everywhere the abstract operation was defined, however it also allows the concrete operation to be defined in states for which the precondition of the abstract operation was false. That is, the precondition of the operation can be weakened.

On the other hand, correctness requires that a concrete operation is consistent with the abstract whenever it is applied in a state where the abstract operation is defined. However, the outcome of the concrete operation only has to be consistent with the abstract, and not identical. Thus if the abstract allowed a number of options, the concrete operation is free to use any subset of these choices. In other words non-determinism can be resolved.

Example 2.1 In the league table example (Example 1.5) we can make a number of valid refinements even without changing the state space. For example, we might weaken the precondition of *Play* to describe the effect of this operation when two identical teams were used as parameters. Clearly a team cannot play against itself (despite the best efforts of some players) and we sensibly choose that *Play* should not change the state in these circumstances:

<i>Play</i> _D
$\Delta \text{LeagueTable}$
$\text{team?} : \text{CLUBS} \times \text{CLUBS}$
$\text{score?} : \mathbb{N} \times \mathbb{N}$
$\text{Play} \vee (\text{first team?} = \text{second team?} \wedge \exists \text{LeagueTable})$

We might also choose to reduce the non-determinism in the operation *Champion*, which currently outputs any team with maximum points, to one that also considers goal difference.

Champion <hr/> $\exists \text{LeagueTable}$ $\text{team!} : \text{CLUBS}$ <hr/> $\forall c : \text{CLUBS} \bullet p(\text{table}(c)) = 38$ $\text{pts}(\text{table}(c)) \leq \text{pts}(\text{table}(\text{team!}))$ $(f(\text{table}(c)) - a(\text{table}(c))) \leq (f(\text{table}(\text{team!})) - a(\text{table}(\text{team!})))$
--

Then

pre $\text{Play} = [\text{LeagueTable}; \text{team?} : \text{CLUBS} \times \text{CLUBS}; \text{score?} : \mathbb{N} \times \mathbb{N} \mid$
 $\text{first team?} \neq \text{second team?}]$
pre $\text{Play}_D = [\text{LeagueTable}; \text{team?} : \text{CLUBS} \times \text{CLUBS}; \text{score?} : \mathbb{N} \times \mathbb{N}]$

Applicability and correctness for Play easily follow. Similarly pre $\text{Champion} =$ pre Champion_D and $\forall \Delta \text{LeagueTable}; \text{team!} : \text{CLUBS} \bullet \text{Champion}_D \Rightarrow \text{Champion}$. The operations Play_D and Champion_D are thus operation refinements of their abstract counterparts. \square

Example 2.2 We model the payment of a given amount by means of a sequence of coins. The set of all different coins is modelled by a set of their values.

$\mid \text{coins} : \mathbb{F} \mathbb{N}_1$

For example, in the UK the normal collection of coins (amounts in pence) is

$\text{coins} = \{1, 2, 5, 10, 20, 50, 100, 200\}$

An operation to return a required amount in any combination of coins is then

GiveAmt <hr/> $\text{amt?} : \mathbb{N}$ $c! : \text{seq coins}$ <hr/> $\text{amt?} = +/c!$
--

Note that there is no state in this operation. If required, this can be taken as an abbreviation for using a trivial state space which has only one possible value.

The operation GiveAmt is clearly operation-refined by

GiveFewCoins <hr/> $\text{amt?} : \mathbb{N}$ $c! : \text{seq coins}$ <hr/> $\text{amt?} = +/c!$ $\forall c : \text{seq coins} \bullet +/c = \text{amt?} \Rightarrow \#c \geq \#c!$
--

as, if a sequence of coins exists, so does a shortest sequence. Correctness follows from Lemma 2.3.

In many Eurozone countries, prices are expressed in cents, but supermarkets actually round amounts to the nearest 5 cents, as if the 1 and 2 cent coins did not exist and the smallest coin was worth 5 cents. In such a situation, *GiveAmt* is partial, and we might need to refine it to

<i>GiveRounded</i>
$amt? : \mathbb{N}$
$c! : \text{seq } coins$
$\forall c : \text{seq } coins \bullet +/c - amt? \geq +/c! - amt? $

GiveRounded implies *GiveAmt* in those cases where the exact amount can be paid out. For applicability, observe that *GiveRounded* is total. (It is even defined when $coins = \emptyset$.) \square

2.3 From Concrete to Abstract Data Types

In the setting of repertoires, operation refinement is the only *simple* refinement possible, due to the actual states being observable. The canonical interpretation of an *abstract* data type in Z has as its observations only the outputs generated by a particular sequence of inputs. In addition, through a construction similar to the one for state-observations in the previous section, partiality of operations may also be “observed”.

Not only has the notion of observation to change, but also the notion of program. This is due to ADTs (unlike repertoires) having an initialisation. Programs are still characterised by sequences of indices, and the evolution of the state is governed by sequential composition of (initialisation and) operations. Observations are now pairs of sequences: the sequence of outputs produced by a particular sequence of inputs. However, the formalisation of this is postponed to Chap. 3, which gives a relational interpretation of Z ADTs.

We will, however, present more possibilities for “simple” refinement, which do not change the state space, although they do require consideration of the full ADT for their verification.

2.4 Establishing and Imposing Invariants

In Chap. 1 we discussed how invariant relations between the various components would be described in a state schema. However, in first drafts of specifications quite often you will not have thought of all the invariants that are required to hold in

the system so, as a consequence, intuitively correct properties cannot be verified. Sometimes this means a return to the drawing board, but sometimes refinement will actually allow the introduction of invariants. How this can be done will be described in this section.

Formally, one could view the introduction of invariants either as a change to the state schema, or as a change to all operations. In the former view, introduction of invariants is a special (degenerate) case of data refinement, which we will describe in Chap. 4. In this chapter, however, we will concentrate on the latter approach, with the formal justification deriving from the data refinement conditions to be presented later.

We describe two ways of introducing invariants: *establishing* invariants is a refinement step in both directions, and *imposing* invariants is, in general, a refinement in only one direction. What the two have in common, is the paradoxical effect that preconditions can be *strengthened* in refinement—though only if the strengthening is an acceptable invariant of the original system already.

2.4.1 Establishing Invariants

Establishing invariants concerns the elimination of unreachable parts of the state space. This may be a useful and necessary step if certain safety properties of the system appear hard to prove from the specification.

Example 2.3 Consider the following (artificial) example.

<div style="border-bottom: 1px solid black; margin-bottom: 5px;"> <i>State</i> </div> <div style="border-bottom: 1px solid black; margin-bottom: 5px;"> $x : \text{seq } \mathbb{N}$ </div> <div style="border-bottom: 1px solid black; margin-bottom: 5px;"> <i>Obs</i> </div> <div style="border-bottom: 1px solid black; margin-bottom: 5px;"> $\exists \text{State}$ </div> <div style="border-bottom: 1px solid black; margin-bottom: 5px;"> $y! : \mathbb{N}$ </div> <div style="border-bottom: 1px solid black; margin-bottom: 5px;"> $y! = \text{head } x$ </div>	<div style="border-bottom: 1px solid black; margin-bottom: 5px;"> <i>Init</i> </div> <div style="border-bottom: 1px solid black; margin-bottom: 5px;"> State' </div> <div style="border-bottom: 1px solid black; margin-bottom: 5px;"> $x' = \langle 1 \rangle$ </div> <div style="border-bottom: 1px solid black; margin-bottom: 5px;"> <i>Inc</i> </div> <div style="border-bottom: 1px solid black; margin-bottom: 5px;"> ΔState </div> <div style="border-bottom: 1px solid black; margin-bottom: 5px;"> $x' = \langle 1 \rangle \cap x$ </div>
---	---

The operation *Obs* is always going to be applicable, as it is “obvious” from the rest of the specification that x will never be empty. □

Another example is that which occurs in low-level refinement steps, where one often requires that the data structures to be implemented are finite. This clashes with the common habit of Z specifiers using \mathbb{P} where \mathbb{F} would have worked equally well.

Such desirable invariants can in many cases be added to an ADT at the cost of a relatively simple proof. If the invariant holds for the ADT as given, adding it results in a refinement-equivalent specification.

Theorem 2.3 *Given an ADT $(State, Init, \{Op_i\}_{i \in I})$, the schema Inv on the state $State$ is an invariant of the ADT iff*

1. *it is guaranteed to be established by the initialisation:*

$$\forall Init \bullet Inv'$$

2. *it is preserved by every operation:*

$$\forall i : I \bullet \forall \Delta State \bullet Inv \wedge Op_i \Rightarrow Inv'$$

In that case both $(State \wedge Inv, Init, \{Op_i\}_{i \in I})$ and $(State, Init, \{Op_i \wedge Inv \wedge Inv'\}_{i \in I})$ are ADTs which are refinement equivalent to the given ADT.

Proof The original ADT's being a refinement of either of the modified ones follows from operation refinement on each individual operation (widening the preconditions to include the possibility of $\neg Inv$; condition 2 ensures that the postcondition is not weakened).

The modified ADTs' being a refinement of the original follows from Theorem 2.4. □

The strongest invariant that can be established is by definition the strongest property satisfying the above conditions, which can easily be seen to be *reachability*: the property that characterises that the state can be reached from some initial state through a sequence of operations.

Example 2.4 Continuing from Example 2.3, the property that $x \neq \langle \rangle$ is an invariant; the proof conditions reduce to $\langle 1 \rangle \neq \langle \rangle$ and $x \neq \langle \rangle \Rightarrow x \frown \langle 1 \rangle \neq \langle \rangle$, both of which obviously hold. Thus, this invariant can safely be added to the state (or to every operation). □

Similarly, finiteness can be proved invariant for any set in a state schema which is initialised to a finite set and changed using only the “standard” binary set operators with finite arguments. In that case, \mathbb{F} can be used rather than \mathbb{P} .

For example, *squad*, in Example 1.1, is declared as having the type $\mathbb{P}PLAYER$. However, it is initially empty and always remains finite. So states with infinite squads are unreachable and could be eliminated from the state space.

2.4.2 Imposing Invariants

Even when the required invariant does not follow directly from the specification, it may well be the case that it can be added by resolving non-determinism in the specification in a suitable way.

Theorem 2.4 *Given an ADT $(State, Init, \{Op_i\}_{i \in I})$, the schema Inv on the state $State$ can be imposed as an invariant on the ADT iff*

1. *it is allowed by the initialisation:*

$$\exists Init \bullet Inv'$$

2. *it can be imposed on every operation:*

$$\forall i : I \bullet \forall State \bullet Inv \wedge \text{pre } Op_i \Rightarrow \text{pre}(Op_i \wedge Inv')$$

In that case both $(State \wedge Inv, Init, \{Op_i\}_{i \in I})$ and $(State, Init \wedge Inv', \{Op_i \wedge Inv \wedge Inv'\}_{i \in I})$ are refinements of the original ADT.

Proof This follows from a degenerate case of data refinement, a full proof will be given after Definition 4.3. The concrete state is a subset of the abstract state, with the invariant as the retrieve relation. The correctness condition follows from Lemma 2.3; applicability simplifies to condition 2. \square

2.5 Example: London Underground

This example describes people travelling on the London Underground in a very abstract sense—just how abstract will become clear when we manage to prove refinements that display rather unexpected (if not unacceptable!) behaviour.

We assume types of stations and lines:

$$[STATION, LINE]$$

A network consists of some (names of) lines with the lists of the stations that are on them. There is always at least 1 station on every line (i.e., even the shortest possible line has a starting and ending station).

$$NET == LINE \rightarrow \text{seq}_1 STATION$$

We will need to describe graph-like aspects of such a network. The relation *sameline* describes a graph linking every two stations on the same line; *connected* then represents the reachability relation on that graph. It is a transitive relation because it is a transitive closure, and reflexive and symmetric because *sameline* is reflexive and symmetric (this is obvious from the form of its definition). We also introduce *lines* and *stations* to make the specifications that follow more readable. (Understanding Z should not be about deciphering expressions involving multiple instances of “ran” and “dom”.)

$connected : NET \rightarrow (STATION \leftrightarrow STATION)$ $sameline : NET \rightarrow (STATION \leftrightarrow STATION)$ $lines : NET \rightarrow \mathbb{P} LINE$ $stations : NET \rightarrow \mathbb{P} STATION$
$\forall x, y : STATION; n : NET \bullet$ $(x, y) \in sameline\ n \Leftrightarrow \exists l : lines\ n \bullet x \in \text{ran}(n\ l) \wedge y \in \text{ran}(n\ l)$ $connected\ n = (sameline\ n)^+$ $lines\ n = \text{dom}\ n$ $stations\ n = \text{ran}(\bigcup \text{ran}\ n)$

Rather than describing the entire London Underground map, we will record only some crucial properties that will be used later.

$lu_0 : NET$ $victoria, mc : STATION$
$victoria \in \text{dom}((connected\ lu_0) \setminus \text{id})$ $mc \notin \text{stations}\ lu_0$

The map represented by lu_0 connects at least two *different* stations, Victoria being one of these. Mornington Crescent is also special—for not being part of the map. Historically, this was the case from 1992 to 1998. Together these properties imply the existence of at least three stations.

Finally, we present our rather abstract ADT.

LU $lu : NET$ $here : STATION$ <hr/> $here \in \text{stations}\ lu$	$Travel$ ΔLU $h! : STATION$ <hr/> $lu' = lu$ $here \mapsto here' \in \text{connected}\ lu$ $h! = here'$
$Init$ LU' <hr/> $lu' = lu_0 \wedge here' = victoria$	

The *Travel* operation represents a trip between two stations, possibly with changes.

We can now illustrate a number of simple refinements, of the various types described above. First, an operation refinement of *Travel* is given by

$DontTravel$ $\exists LU$ $h! : STATION$ <hr/> $h! = here$

Its precondition is equal to the precondition of *Travel*, which is *true* in any possible state. The postcondition can be strengthened to $here' = here$ because we know $here \in stations$ and *connected* *lu* is reflexive.

The rules for operation refinement do not allow us to refine *Travel* to

$$TrueTravel == Travel \wedge \neg DontTravel$$

This operation, which requires the starting station and end station to be *different*, is a realistic mode of travel for anyone but kids, trainspotters, and the terminally forgetful. Moreover, we know that in the above specification, starting from Victoria, we will always end up in a station connected to some different station, which could be Victoria itself when we are not there already. (In graph terms: we always remain in the component containing Victoria, and this component has at least 2 nodes due to the restriction on *lu*.) So this condition should always be satisfiable.

However, imagine a station in *stations* *lu*₀ which only occurs as the single station on a single line—let's call it *mdome*. (In graph theoretic terms, *mdome* is an isolated node in *connected* *lu*₀.) The precondition of *Travel* holds when $here = mdome$, but not the precondition of *TrueTravel*. Thus, operation refinement fails (for some possible values of *lu*₀).

The solution to this is to establish the knowledge that we will always be in the component of the graph to which Victoria belongs as an invariant. This also requires the information that the map is as it was initially.

<i>InVicComp</i>	_____
<i>LU</i>	

$(here, victoria) \in connected\ lu$	
$lu = lu_0$	

The conditions for establishing an invariant are proved using the properties of *connected* *lu*₀. Reflexivity of *connected* ensures that the invariant is established initially $((victoria, victoria) \in connected\ lu_0)$. Transitivity and symmetry of *connected* ensure that the invariant is preserved by *Travel*.

After this invariant has been added to *Travel* (or to the state—technically this is a data refinement step as argued above) *TrueTravel* is provably an operation refinement of *Travel*.

Another invariant that could be established is $here \neq mc$. This would also require $lu = lu_0$ or some weaker condition which would prevent Mornington Crescent being reconnected.

Finally, we will look at imposing an invariant. Our travellers will like this even less than the previous refinements.⁵

⁵In other words, by having the single operation *Travel* as its interface, our ADT does not express the level of external choice that would be necessary from the customer's point of view.

Consider the operation

$NearVic$ ΔLU $h! : STATION$
$lu' = lu$ $here \mapsto here' \in connected\ lu$ $(here', victoria) \in sameline\ lu$ $h! = here'$

That is, you can travel anywhere provided there is a direct train back to Victoria. This is not an operation refinement of the initial *Travel*, because the postcondition is not satisfiable from any station *here* which is at least 2 changes away from Victoria⁶ (nor is it satisfiable from any station in a different component of the graph such as *mdome*, of course). Moreover, we cannot establish the invariant $(here, victoria) \in sameline\ lu$, because it is *not* guaranteed to be preserved by *Travel*.

However, this invariant *can be imposed* on the specification. Initially it is certainly true. Then, in *Travel*, whenever we start out in a station on one line with Victoria, we can always find a connected station that is also on one line with Victoria – for example, by staying where we are, or travelling to Victoria itself! In fact, even the invariant $here = victoria$ can be imposed, but that (in combination with $lu = lu_0$) should certainly be the strongest possible one.

2.6 Bibliographical Notes

The principle of substitutivity is so intuitive that it has often been proposed under various other names. It is called “semantic implementation correctness” in [3], attributing the term to Gardiner and Morgan [4]. Nipkow [6] gives a similar intuitive definition of “implementation”. Related fundamental definitions follow from formalising different methods of testing a specification and comparing their observations, see the work by De Nicola [2] and the extensive hierarchy of refinement relations explored by Van Glabbeek [9, 10], and also more recently work by Reeves and Streader [7]. For other references to early work on refinement we refer to the monograph by de Roever and Engelhardt [3].

Typically, Z textbooks do cover operation refinement separately from data refinement, as we do here. However, the other forms of simple refinement are not normally covered independently, although the Z community has been aware of them [8]. Some textbooks even require the abstract and concrete state to be always different in data refinement, to avoid name capture.

⁶At the time of writing, there were actually very few such stations.

References

1. de Bakker, J. W., de Roever, W.-P., & Rozenberg, G. (Eds.) (1990). *REX Workshop on Step-wise Refinement of Distributed Systems*, Nijmegen, 1989. *Lecture Notes in Computer Science: Vol. 430*. Berlin: Springer.
2. de Nicola, R. (1987). Extensional equivalences for transition systems. *Acta Informatica*, 24(2), 211–237.
3. de Roever, W.-P., & Engelhardt, K. (1998). *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge: Cambridge University Press.
4. Gardiner, P. H. B., & Morgan, C. C. (1993). A single complete rule for data refinement. *Formal Aspects of Computing*, 5, 367–382.
5. McIver, A., & Morgan, C. C. (2004). *Abstraction, Refinement and Proof for Probabilistic Systems*. Berlin: Springer.
6. Nipkow, T. (1990) Formal verification of data type refinement—theory and practice. In de Bakker et al. [1] (pp. 561–591).
7. Reeves, S., & Streader, D. (2011). Contexts, refinement and determinism. *Science of Computer Programming*, 76(9), 774–791.
8. Strulo, B. (1995). Email communication.
9. van Glabbeek, R. J. (1993). The linear time–branching time spectrum II. The semantics of sequential systems with silent moves (extended abstract). In E. Best (Ed.), *CONCUR'93, 4th International Conference on Concurrency Theory. Lecture Notes in Computer Science: Vol. 715* (pp. 66–81). Berlin: Springer.
10. van Glabbeek, R. J. (2001). The linear time–branching time spectrum I. The semantics of concrete sequential processes. In J. A. Bergstra, A. Ponse, & S. A. Smolka (Eds.), *Handbook of Process Algebra* (pp. 3–99). Amsterdam: North-Holland.

Refinement in Z and Object-Z
Foundations and Advanced Applications
Derrick, J.; Boiten, E.A.
2014, XVIII, 492 p., Hardcover
ISBN: 978-1-4471-5354-2