Under consideration for publication in Formal Aspects of Computing

Specification by interface separation

I. J. Hayes[1] and J. W. Sanders[2]

[1]Department of Computer Science, University of Queensland, Brisbane, 4072, Australia
[2]Programming Research Group, O.U.C.L.,
Wolfson Building, Parks Rd, Oxford, OX1 3QD, England

Keywords: Formal specification; specification language Z

Abstract. In specifying an operation it is often advantageous to describe it with abstract inputs and outputs whose concrete representation is described separately. For example, it is often convenient to describe as a set, input which in practice occurs as a sequence.

The primary advantage of this approach is that one can initially concentrate on specifying an operation without the representations of its interface (that is, its inputs and outputs) obscuring the more important concerns of its abstract functional properties. Interface representations can be tackled independently, after the abstract functionality has been decided.

Such separation of an operation into an abstract core and its interface with its environment makes the task of specification simpler, aids clarity of the result, and encourages reuse of both the abstract operation and its interface descriptions.

1. Introduction

The specification of an operation (or program) often embodies details of its interface—that is, its input and output—that, if included directly in its specification, would obscure its function. Such details are typically imposed by an environment in which particular representations of the inputs and outputs are a consequence of the way the operation is to be used. Apart from compromising clarity of the specification they may make it difficult subsequently to reuse essentially the same operation in different environments. In such situations we propose separation of the specification into a core abstract description of the operation (written using suitably abstract representations of input and output) and descriptions of the way the actual inputs and outputs represent the abstract inputs and outputs. Not only can this lead to a clearer specification; it also leads to a specification that is less dependent on the particular environment. For a new environment only the interface representations need be updated. The result is separation of the representation of the interface from the functionality of the abstract core of the operation.

In such cases we propose that the specification of an operation be given in three parts:

- an operation specification (*aop*) using abstract inputs and outputs;

Correspondence and offprint requests to: I. J. Hayes, Department of Computer Science, University of Queensland, Brisbane, 4072, Australia.

- the concrete representation (*in*) of the inputs; and
- the concrete representation (*out*) of the outputs.

The specification of the complete operation is the composition of the three components:

$$op == in \gg aop \gg out$$

Each component can be regarded as a binary relation, and '$\gg$' as relational composition. The examples given in this paper are specified using Z schemas [Spi92], with schema piping playing the part of '$\gg$'. However the technique proposed is largely independent of notation.

In Section 2 we give an example of the use of separate interface representations to aid in structuring a specification. Section 3 discusses representation restrictions and errors. Section 4 discusses the use of relational composition in specification more generally. Section 5 contains a typical numerical example, whose interface representation is of a more general nature than the others considered in this paper. Finally Section 6 reviews the approach.

2. File update

To demonstrate our approach the first example is typical of specifications written using formal methods, like Z and VDM [Jon90]. The example is a sequential file update similar to that specified in [Hay93, p7]. A file is considered to be a mapping from keys to values ($Key \pfun Value$). Abstractly, a file-update operation takes as input a set $d?$ of keys to be deleted from the file and a partial mapping $r?$ to replace part of the file; it outputs the set $e!$ of keys that were to be deleted but were not in the file. In Z:

File_Update

$$f, f' : Key \pfun Value$$
$$d? : \mathbb{P}\, Key$$
$$r? : Key \pfun Value$$
$$e! : \mathbb{P}\, Key$$

$$d? \cap \operatorname{dom} r? = \{\} \land$$
$$f' = (d? \ndres f) \oplus r? \land$$
$$e! = d? \setminus \operatorname{dom} f$$

The keys to be deleted and the keys to be replaced must not overlap. The keys to be deleted are removed from the file and the replacements, $r?$, replace items in (or add items to) the file. Any keys in $d?$ that were not in the file are returned in the set, $e!$, of erroneous keys. In the above the state is described abstractly (as a mapping) and the inputs and outputs are described equally abstractly.

The concrete operation reflects its environment's lower-level nature: input and output are expressed as sequences. Moreover the inputs—delete or update—are represented by a single combined sequence. Each item in the sequence consists of a key/update pair, where each update is either a deletion or a new replacement value:

$$Update ::= delete \mid new\langle\langle Value\rangle\rangle$$

An update is either *delete*, indicating that the corresponding key is to be deleted, or of the form $new(v)$ indicating that the value associated with the key is to be replaced by v (or the key is added with value v if it was not present in the original file). The sequence is strictly ordered on keys.

Thought of as representing the elements they contain, the input and output sequences must contain the same information as the input and output sets they represent in the abstract schema. That would lead us, were we to ignore the possibility of interface separation, to the following rather opaque specification of the concrete operation.

*File_Update_Rep*1

$$
\begin{array}{l}
f, f' : Key \pfun Value \\
up? : \mathrm{seq}(Key \times Update) \\
se! : \mathrm{seq}\, Key \\
\hline
(\forall\, i, j : \mathrm{dom}\, up? \bullet i < j \Rightarrow first(up?(i)) < first(up?(j))) \land \\
\{k : Key \mid (k, delete) \in \mathrm{ran}\, up?\} \cap \\
\quad \mathrm{dom}\{k : Key;\ v : Value \mid (k, new(v)) \in \mathrm{ran}\, up?\} = \{\,\} \land \\
f' = (\{k : Key \mid (k, delete) \in \mathrm{ran}\, up?\} \domsub f) \oplus \\
\quad \{k : Key;\ v : Value \mid (k, new(v)) \in \mathrm{ran}\, up?\} \land \\
\mathrm{ran}\, se! = \{k : Key \mid (k, delete) \in \mathrm{ran}\, up?\} \setminus \mathrm{dom}\, f \land \\
(\forall\, i, j : \mathrm{dom}\, se! \bullet i < j \Rightarrow se!(i) < se!(j))
\end{array}
$$

The clarity of the abstract operation has been obscured by the complex nature of the input and output.

Let us now see how it can be simplified by interface separation. First we formalise the relationship between the abstract and concrete interfaces, then combine those representations with the abstract operation. The representation of the updates is specified by schema:

Update_Rep

$$
\begin{array}{l}
up? : \mathrm{seq}(Key \times Update) \\
d! : \mathbb{P}\, Key \\
r! : Key \pfun Value \\
\hline
(\forall\, i, j : \mathrm{dom}\, up? \bullet i < j \Rightarrow first(up?(i)) < first(up?(j))) \land \\
d! = \{k : Key \mid (k, delete) \in \mathrm{ran}\, up?\} \land \\
r! = \{k : Key;\ v : Value \mid (k, new(v)) \in \mathrm{ran}\, up?\}
\end{array}
$$

The output representation is treated in a similar manner. The erroneously deleted keys are represented by a strictly-ordered sequence of keys.

Error_Rep

$$
\begin{array}{l}
e? : \mathbb{P}\, Key \\
se! : \mathrm{seq}\, Key \\
\hline
(\forall\, i, j : \mathrm{dom}\, se! \bullet i < j \Rightarrow se!(i) < se!(j)) \land \\
\mathrm{ran}\, se! = e?
\end{array}
$$

The effect of the concrete operation is specified by the composition of the input representation, the abstract operation and the output representation:

$$File\_Update\_Rep == Update\_Rep \gg File\_Update \gg Error\_Rep$$

The piping operator '$\gg$' identifies outputs (variables with names ending in '!') of its first operand with inputs (ending in '?') to its second operand that have the same basename and type. Such variables then become local in the composition. *File_Update_Rep* can be expanded to give the equivalent operation specification in one larger schema.

*File_Update_Rep*2

$$
\begin{array}{l}
f, f' : Key \pfun Value \\
up? : \mathrm{seq}(Key \times Update) \\
se! : \mathrm{seq}\, Key \\
\hline
(\forall\, i, j : \mathrm{dom}\, up? \bullet i < j \Rightarrow first(up?(i)) < first(up?(j))) \land \\
(\exists\, d : \mathbb{P}\, Key;\ r : Key \pfun Value;\ e : \mathbb{P}\, Key \bullet \\
\quad d = \{k : Key \mid (k, delete) \in \mathrm{ran}\, up?\} \land \\
\quad r = \{k : Key;\ v : Value \mid (k, new(v)) \in \mathrm{ran}\, up?\} \land \\
\quad d \cap \mathrm{dom}\, r = \{\} \land \\
\quad f' = (d \rhd f) \oplus r \land \\
\quad e = d \setminus \mathrm{dom}\, f \land \\
\quad \mathrm{ran}\, se! = e) \land \\
(\forall\, i, j : \mathrm{dom}\, se! \bullet i < j \Rightarrow se!(i) < se!(j))
\end{array}
$$

The predicate can be simplified to eliminate d, r and e, in which case it becomes identical to the predicate of *File_Update_Rep*1.

While it is possible to give the expanded form as a specification of the operation, splitting the specification into parts dealing with representations and a part dealing with the abstract operation leads to a clearer separation of concerns:

- the abstract operation is separate from its interface, which is specific to a given environment;
- the abstract purpose of the operation is clearer (for both writer and reader) than in the combined specification;
- the concrete (efficient) purpose of the interface is clearer (for both writer and reader) than in the combined specification;
- the abstract operation can be reused, consistently, in different contexts; and
- the interface representation can be reused, consistently, with different operations.

3. Representation errors

Consider the operation *Double* that doubles a natural number.

Double

$$
\begin{array}{l}
x?, y! : \mathbb{N} \\
\hline
y! = 2 * x?
\end{array}
$$

The input is represented as a 32-bit unsigned quantity.

$$Bit32 == (0 \,..\, 4294967295)$$

The input-representation schema restricts the input to the range of values that can be represented in 32 bits.

In_Rep

$$
\begin{array}{l}
x? : Bit32 \\
x! : \mathbb{N} \\
\hline
x! = x?
\end{array}
$$

The output is also to be represented as a 32-bit number, but this is not always possible. For outputs that are out of the allowable range the Boolean variable *overflow*! is set to *True* (and the output left unconstrained).

Rep_Err

$$y? : \mathbb{N}$$
$$y! : Bit32$$
$$overflow! : \mathbb{B}$$

$$(y? \in Bit32 \Rightarrow y! = y? \land overflow! = False) \land$$
$$(y? \notin Bit32 \Rightarrow overflow! = True)$$

The range-restricted operation can be defined as follows.

$$Double\_Rep == In\_Rep \gg Double \gg Rep\_Err$$

An alternative approach is to disallow inputs that cause overflow. This can be specified by choosing the following output representation.

Out_Rep

$$y? : \mathbb{N}$$
$$y! : Bit32$$

$$y! = y?$$

The combination is defined as follows.

$$Double\_Rep2 == In\_Rep \gg Double \gg Out\_Rep$$

Because *Out_Rep* does not accept out-of-range results, that restriction is imposed, as a result of the conjunction and renaming in the definition of '$\gg$', on the input to the whole operation: the input must be such that its double is within range. Indeed the operator '$\gg$' acts like relational composition rather than sequential composition in an imperative programming language. This significant aspect of relational composition is examined in the next section.

Note that the easy distinction between the two versions of *Double* is another benefit of interface separation.

4. Composition of specifications

A specification in Z can be thought of as specifying a relation between inputs and outputs and hence can be characterised by a predicate.

The piping (relational) composition, $P \gg Q$, of schemas P and Q specifies a new schema. Any outputs of P (variables with names ending in '!') that match inputs to Q (variables with names ending in '?') when their respective decorations '!' and '?' are removed, are identified in the composition and become internal to it. The types of the corresponding variables must match for the composition to be well-defined. If we let $y!$ (of type T) stand for the outputs of P that match corresponding inputs $y?$ (also of type T) to Q, then the composition of P and Q is given by

$$P \gg Q == (\exists\, y : T \bullet P\,[y/y!] \land Q\,[y/y?])$$

where $P\,[y/y!]$ is the schema P with every free occurrence of the variable $y!$ replaced by y. (We have assumed here that the variable y does not already occur in P and Q; if it does we can choose some fresh variable not occurring in either P or Q instead of y.) An input $x?$ of P is related to an output $z!$ of Q by the composition $P \gg Q$ if there exists a y such that $x?$ is related to y by $P\,[y/y!]$ and y is related to $z!$ by $Q\,[y/y?]$.

When used for interface specification, composition of schemas acts like relational composition of binary relations. Relational composition should not be confused with sequential composition (';') in imperative programming languages.[1] The distinction between relational and sequential composition becomes apparent when the specification P is nondeterministic, that is, if there is more than one output value allowed for a particular input. Consider a simple example involving a nondeterministic operation P. For input a, P can

[1] Note that we are referring to sequential composition (';') in programming languages; this should not be confused with the Z schema composition operator '$\fatsemi$' which also acts as a form of relational composition similar to '$\gg$' but composing states rather than inputs and outputs.

produce an output of either b_1 or b_2; but Q terminates only on input b_2, producing output c. The relational composition $P \gg Q$ produces output c for input a. This behaviour resembles that of backtracking in logic programming languages such as Prolog.

On the other hand for the sequential composition, 'P; Q', of P and Q, the assumption is that P computes its result independently of whether or not Q terminates on inputting that result. If for a particular input a, P can produce either an input b_1 or b_2, but Q terminates only on input b_2, then the sequential composition 'P; Q' is not guaranteed to terminate on input a because P may choose output b_1. There is no possibility of backtracking.

All imperative programming languages support sequential composition but few support the backtracking facility of relational composition. In our present context we are concerned with relational composition as a specification operator; there, as we have seen, it is a powerful tool allowing separation of the representation of the interface from that of the abstract operation.

The laws for '$\gg$' when used for interface specification mimic those for relational composition. Indeed in that context the following condition holds.

> Every output of an input representation is matched by an input to the operation, and every input of an output representation is matched by an output of the operation. (In addition, input and output representations do not have any state components.) (1)

Fortunately condition (1) is sufficient to ensure that '$\gg$' behaves like relational composition. We assume it holds from here on. We now comment on three ways of exploiting interface separation, each based on a separate law.

The most important law is associativity. Firstly it enables parentheses to be omitted in $in \gg op \gg out$, demonstrating the unimportance of the order in which in and out are combined with op. Secondly associativity enables a complicated interface to be specified incrementally: from abstract interface to slightly less abstract, to even less abstract, and so on to concrete interface. The increments can then be piped together to give the desired complicated interface. That method constructs

$$in_n \gg (\ldots \gg (in_1 \gg op \gg out_1) \gg \ldots) \gg out_n$$

but, by associativity, that is equivalent to

$$(in_n \gg \ldots \gg in_1) \gg op \gg (out_1 \gg \ldots \gg out_n).$$

Disjunction distributes '$\gg$'. Frequently an operation is specified as $op1 \vee op2$, where $op1$ describes the ideal case and $op2$ the error case. Employing the technique of interface separation one specifies

$$in \gg (op1 \vee op2) \gg out,$$

but that is equivalent to

$$(in \gg op1 \gg out) \vee (in \gg op2 \gg out).$$

and correctness of that factorisation follows by distributivity, provided both $op1$ and $op2$ satisfy condition (1).

Examples like *Double_Rep* and *Double_Rep*2 of the previous section occur frequently in hardware design where *Double* is a typical abstract view of a combinational circuit. (In a hardware application the type *Bit*32 would typically be replaced by a type of bitstrings.) Interface separation permits such devices to be specified, combined and reasoned about abstractly, before interface representation. One of the reasons is that input and output representations are often the inverse of each other, as well as being bijections, so that $out \gg in$ acts as an identity. Thus adjacent interface representations cancel each other out when the abstract components are piped together. This time the law justifying such factorisation is

$$(in \gg op1 \gg out) \gg (in \gg op2 \gg out) = in \gg (op1 \gg op2) \gg out.$$

which follows from the associativity of '$\gg$' and the fact that $(out \gg in)$ acts as an identity.

Such properties of '$\gg$' play an important part in its utility for interface separation.

One property of composition requires care. In general it is not possible to refine a specification $P \gg Q$ by refining P and Q independently and then combining these refinements using composition. But this is not really of any consequence here: our goal has been to promote clear specification. The specification of a whole operation has been given by composition and it is the whole operation that must be refined. The choice of any internal representations of variables for the implementation may match the interface representation, or

alternatively the abstract representation; but it could equally well match a third representation that allows a more efficient implementation of the operation. The main point here is that the issues of specification and implementation should be kept separate. As usual the form of a specification is chosen for clarity and that of an implementation for efficiency.

5. A numerical example – sine

The final example is chosen to demonstrate another typical use of interface separation: numerical computation. Whilst previous interface representations were one-to-one (with the exception of *Rep_Err* when restricted to the complement of $Bit32$), the present example provides an interface representation which is one-to-many from abstract to concrete.

As an example of the application of separated interface representations in a numerical context, we consider an operation to calculate the sine of a real number. It is specified abstractly as follows.

Sine

$$x?, y! : \mathbb{R}$$

$$0 \leq x? < 2 * \pi \land$$
$$y! = sin(x?)$$

This specification is given in terms of real numbers which can only be approximated on a machine by, say, a floating-point representation: some finite subset of the reals that we represent here by the set *Float*. The actual input and output use floating-point representation. The input representation has the effect of restricting the input to be a *Float* rather than an arbitrary real number.

In

$$x? : Float$$
$$x! : \mathbb{R}$$

$$x! = x?$$

One could argue that the input for the original specification could just as easily have been a *Float* rather than a real, but the representation of *Float* may be different in different environments, such as different machines.

The separation becomes clearer for the output representation where we need to deal with representation restrictions. For the output, the floating point representation can only approximate the abstract output to within a given relative error bound *Rel_Error* and absolute error bound *Abs_Error*:

$$Rel\_Error : \mathbb{R}$$
$$Abs\_Error : \mathbb{R}$$

$$0 < Rel\_Error < 1 \land 0 < Abs\_Error$$

We assume here that the definition of the subset *Float* of $\mathbb{R}$ reflects its eventual representation using the conventional combination of a mantissa plus an exponent. The relative error allows for the fact that the mantissa has only finite accuracy and the absolute error for the fact that the exponent has a limited range – in this case we are concerned only with the smallest non-zero real that can be represented using the mantissa/exponent form.

Out

$$y? : \mathbb{R}$$
$$y! : Float$$

$$abs((y! - y?) \,/\, y!) < Rel\_Error \lor abs(y! - y?) < Abs\_Error$$

The absolute value of a real number z is denoted by $abs(z)$. Putting these together we get:

$$Sine\_Op == In \gg Sine \gg Out$$

or expanding we get the following.

Sine_Op

$x?, y! : Float$

$0 \leq x? < 2 * \pi \land$
$(abs((sin(x?) - y!) \,/\, y!) < Rel\_Error \lor$
$\quad abs(y! - sin(x?)) < Abs\_Error)$

An alternative approach to handling output-representation errors is explicitly to introduce an underflow indication similar to the overflow indication for the doubling operation. If the output cannot be represented because it is too small, an underflow indication is given instead.

All the benefits of interface separation apply to such numerical examples. The more involved the interface representation, (here one-to-many) the more is to be gained!

6. Discussion

When choosing representations for inputs and outputs, there are a number of approaches possible. The simplest approach is to represent each abstract input (or output) by a corresponding concrete input (or output). One needs to be aware that the choice of representation may restrict the range of inputs allowed, as well as having implications for possible implementation strategies.

Another approach is to consider a group of abstract input (output) variables and represent the group via one or more concrete input (output) variables. There may be different numbers of abstract and concrete variables. The input to the file update specification in Section 2 provides an example where two abstract inputs, the deletions and the replacements, have been represented by a single sequence of updates. This approach has the advantage that combinations of abstract inputs not allowed by the abstract operation can be implicitly excluded from the concrete representation, i.e., there is no way to represent such combinations. In the file update example, the fact that a deletion and a replacement are not allowed for the same key is translated into the requirement of the concrete representation that the sequence is strictly ordered.

The choice of interface representations for an operation (or program) involves concerns separate from the specification of the abstract functionality of the operation. One approach to specifying a system is initially to concentrate on its functionality and ignore the question of input/output representations. This allows one to get, more quickly, an overview of the system's functionality and resolve the more important questions of what it is to do, before considering interface representations. There is no point designing representations of the inputs and outputs to an operation when one has not yet settled on its functionality, or perhaps even on what abstract inputs and outputs it requires.

Once the abstract operation has been devised one can turn one's attention to the interface representation. But even here one may want to decouple the operation from the representation of its interface. Commonly the form of the input/output representations is not so much dictated by the requirements of the abstract operation but can be influenced by the particular programming environment (including such things as the programming language and the data structures it supports) or even the chosen implementation strategy. In this latter case there may be many possible representations that are equally suitable from the point of view of the user of the operation, but if the specifier is aware that one of them allows a more efficient implementation, then the specifier can chose that representation.

In practice it is not uncommon to leave the interface-representation details undecided until further into the implementation phase. Once the choice of interface representation can be made, however, one should be aware that although it is perhaps chosen to suit the implementation, it is really part of the specification and hence cannot be left completely up to the implementor. Agreement is required from the client that the chosen representation is suitable because

- the interface representation determines the external view of the operation;
- the choice of input representation may directly limit the range of inputs handled;
- the choice of output representation may indirectly limit the range inputs handled (as in the doubling example); and
- the choice of output representation may restrict the choices that a non-deterministic abstract operation can make for its corresponding output.

After considering the interface representation it is conceivable that one may wish to change (perhaps restrict)

the abstract operation to accommodate the representations more appropriately. This is part of the usual design cycle which may require compromise when separate specifications are combined to give a single specification.

Acknowledgements The technique described in this paper was reported by the authors at the Workshop on Refinement, University of York, January 1988. Ian Hayes acknowledges financial assistance provided by the Special Studies Program of the University of Queensland. We thank Ken Robinson and Lucy Chubb for useful discussions on this topic.

References

[Hay93] Ian Hayes, editor. *Specification Case Studies*. Prentice-Hall International, second edition, 1993.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, Englewood Cliffs, NJ, second edition, 1990.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, second edition, 1992.