

Formal specification and rapid prototyping: building in quality at the start

M.R. Poppleton

*School of Mathematical and Information Sciences,
Coventry University, Priory Street, Coventry,
CV1 5FB, UK*

ABSTRACT

Formal Methods for Software Engineering, although widely recognised in academe as of potential benefit to industry, have proved difficult to “sell”, both to industry and to students. Building on a novel teaching approach at Coventry University, we present an integrated approach to formal specification and rapid prototyping that offers a real contribution to building in quality at the front end of the development lifecycle.

The significance of formal specification and rapid prototyping for software development are discussed. An introduction is given to the functional programming language Miranda™, the vehicle for our approach. An extended type structure is presented for Miranda, providing an executable specification language based on simple set-theoretical structures. We are thus enabled to construct directly executable specifications which serve as rapid prototypes, useful for early requirements analysis and refinement.

An explanatory example is presented to show the utility of the approach in building quality systems. Z text is included to show the applicability of the approach to formal specifications expressed in languages such as Z and VDM. The example is further used to describe our HCI-builder tool, which enables the user to quickly produce a functional user interface to her prototype application.

1. INTRODUCTION

Key elements of the “Software Crisis” that have been identified are the requirements capture and validation problems. Hekmatpour [6] discusses the “significant cultural gap between the customer and the developer” and the customer’s “very vague requirements which could be interpreted arbitrarily by the developer”. The customer has

502 Software Quality Management

usually not defined her requirements exhaustively by analysing all completeness, consistency and other higher-level properties of the desired system. Communication between customer and requirements analyst is at best imperfect, and needs methodological and tool support to render it most effective. Furthermore, system specification documents are complex, voluminous, and dry in style: it is difficult for the customer to visualise the proposed system from such documents, hence validation activity is often compromised.

Formal specification (FS) and rapid prototyping (RP) have been proposed as key software engineering techniques to address these problems, and have been widely discussed in the literature. Hall [4] and Alexander [1] discuss the benefits of formalising specification activity using mathematical notations such as Z [10] or VDM [8]. Using mathematics for specification may require specialist skills, but it produces elegant, mathematically simple and unambiguous system descriptions. The vagueness and ambiguity inherent in narrative specification is removed: the formality and precision of modelling in mathematics forces the specifier to think very hard about the proposed system. The mathematics enables the specifier to *prove* that the specification has desired properties, such as non-ambiguity. It is widely accepted that the earliest validation in the lifecycle is the most cost-effective: test activity to satisfactorily demonstrate non-ambiguity would be expensive.

With formalism, then, the analyst makes the most of the requirements thus far elicited, and can feed back ambiguities, requests for further information, etc. back to the customer. However, earliest possible *visualisation* by the customer of the proposed functionality is essential. Hekmatpour [6] describes the utility of rapid prototyping for requirements validation with the customer. The prototype, produced cheaply and quickly, allows requirements specification to become an iterative “learning process” of the customer and requirements analyst. Rapid prototyping should clearly dovetail effectively with formal specification of requirements.

Our approach uses the notion of an “executable specification”. Such a thing is possible where the specification language, with its formal structure and logic, has an operational semantics. Considerable work has been done on executable specifications, which provide a natural integration of the two techniques [2], [3], [5], [6], [7], [9], [12]. Some approaches do not give truly “executable specifications”, but the lesser product of “executable quick-translations of formal specifications”: Baxter [2] and O'Neill [9] discuss translation of formal specifications in Z and VDM

respectively, into the Miranda (*Miranda* is a trademark of Research Software Ltd.) functional programming language [12], [13]. These approaches are informal in the sense that type and predicate structures are not preserved: the prototype is some model of the formal text, using Miranda types. The benefit of proofs of properties about the specification are lost. This approach will certainly give a prototype of the specification, but there is no guarantee of equivalence in other than an intuitive sense.

This paper presents a functional style of specification which provides guaranteed equivalence of specification meaning and prototype behaviour. Our specification “language” is a predicate calculus subset defined algebraically using the Miranda *abstype* notation to define abstract data types (ADTs) for *set* and *map*. The specifications are executable by virtue of the hidden ADT implementations, which can be proved to support the axioms of the ADTs.

This truer approach to “executable specification” is exemplified by Hekmatpour’s EPROS [6], a complete development environment based on compilation of VDM text into executable prototypes. Our approach assumes specification in the extended Miranda functional language, although we also discuss how Z specifications can be simply transcribed into our notation. Our approach is similar to that of Alexander’s *me too* tool [1], although we provide HCI-builder facilities missing from that tool.

The original motivation for developing our approach to FS/ RP was a perception of difficulties students were having with introductory topics in the Computer Science programme at Coventry University. Many students who are not specialist software engineers, require foundational Computing content in their programmes of study. We have found that many, if not most of these students have weak skills of abstract symbolic reasoning, and in general, poor confidence of their logical/ mathematical abilities.

Our approach was based on an intention to emphasise system modelling rather than pure formal notations, and to build on students’ intuitions. Most students arrive at university with some experience of imperative style programming. Introductory programming in Miranda is followed by development of the specification notation in Miranda. Discussion of the distinction between specification, prototyping, and subsequent system refinement arises naturally. Formal reasoning gently permeates the course.

The industrial relevance of the approach and associated tool became apparent during development, and is therefore the focus of this paper.

Section 2 briefly introduces the Miranda language. Section 3 describes the extended Miranda type structure in which our specifications are expressed. A simple system is specified and prototyped as a demonstration; this section includes sample Z text. The HCI-builder tool is then described, which enables the user to quickly produce a functional interface to her prototypic application.

Section 4 is the conclusion, in which current problems and future work are described.

2. A BRIEF INTRODUCTION TO FUNCTIONAL PROGRAMMING WITH MIRANDA

"Miranda is a purely functional language, with no imperative features of any kind. A program (script) is a collection of definitions, of functions and other data objects, written in the form of recursion equations"
Turner, [13]

Miranda is strongly typed, all types being constructed from the primitive types *bool* (values *True* and *False*), *num* (integers and real numbers), and *char* (single characters), using tuple, list, and user-defined type constructors. Functions are defined in "curried" style, i.e. arguments are not parenthesised.

Selection of alternative result values for a function, based on predicate (boolean-valued function) tests on the arguments, are performed using either "guard expressions", or "pattern-matching". "Higher-order" functions can be defined, i.e. functions that themselves take functions as arguments. For example, the expression *takewhile p xs* returns all elements from the list *xs* until its predicate argument *p* is negated. *dropwhile* is the inverse function that returns the remaining elements of the list. Note the type of *takewhile* in the following example Miranda script:

```
area::[char] -> num -> [char] (1)
```

```
area fig r = show (pi*r*r), if fig = "circle" (2)
```

```
          = show (r*r), if fig = "square"
```

```
          = error "Invalid figure", otherwise (3)
```

```
takewhile::(* -> bool) -> [*] -> [*] (4)
nextwd::[char] -> ([char], [char]) (5)
nextwd xs = (takewhile (~p) xs,dropwhile (~p) xs) (6)
  where (7)
    p x = x == ' '
```

- (1) `::` denotes a type definition, `[]` a list. This line is a “type signature” for *area*. Two arguments, list-of-characters (i.e. string), and number. Result: string.
- (2) `show` converts num to string. Guard expression denoted by `if`.
- (3) *otherwise* denotes negated disjunction of prior guards, i.e. “if anything else...”
- (4) *takewhile* is “polymorphic”, i.e. it will apply a suitable predicate (of type `* -> bool`) to each of a list of elements of any type `*`.
- (5) A function to split a string into a pair (2-tuple) of (first word, rest of string).
- (6) `~` denotes negation of a predicate. Take all chars until “is this char not equal to a space” becomes *False*, giving first component of result pair. *dropwhile* returns remainder of input string as second component.
- (7) *where* denotes subsidiary definitions of scope local to the current definition only. Here, predicate *p* tests whether its argument is a space or not. Note the overloaded use of “=”.

Here are example applications of these functions at the Miranda prompt:

```
Miranda> area "square" 2
4
Miranda> nextwd "one two three"
("one", " two three")
Miranda>
```

Miranda contains a number of mechanisms for definition of user-defined types. We use the *abstype*, or abstract data type (ADT) mechanism, to define polymorphic *set* and *map* types with associated operations. Using these types and operations, we are equipped to specify required systems in set-theoretic language familiar from notations such as Z [10] and VDM [8], albeit in a functional style. We have the further benefit of being able to execute such specifications.

The *abstype* mechanism incorporates the principle of “encapsulation”. That is, the *set* and *map* types and operations are implemented using Miranda types and algorithms of the implementer’s choice. This implementation is hidden from the

506 Software Quality Management

abstype user; values of the type may only be accessed via the defined interface of operations.

3. FORMAL SPECIFICATIONS WHICH ARE EXECUTABLE

Section 3.1 describes the *set* ADT. Section 3.2 gives a simple executable specification using *set*. Section 3.3 describes the *map* ADT and gives a brief example of the greater expressive power achieved. Section 3.4 develops the example to make the conceptual jump from deterministic to a more general class of specifications. Section 3.5 describes our HCI-builder tool for rapid interface prototyping.

Our specification language is simply a partial implementation of the predicate calculus in a functional language. It is hence structurally equivalent to (subsets of) formal specification languages such as Z and VDM. Since the extended Miranda has been the vehicle of teaching the course from which the approach derives, we use it here for specification. An alternative, more formal approach is to specify in Z or VDM and then transcribe to Miranda for execution/prototyping.

In this section, Miranda text will be followed by the equivalent specification text in the Z language. This will serve to demonstrate how a formal specification written in such a language can be transcribed to an executable specification in Miranda.

3.1 The Set Abstract Data Type

A polymorphic set type is introduced as an algebraic ADT definition (Miranda *abstype*). A key point is that the *set* implementation obeys its axioms; the onus is on the implementer to mathematically prove this using the logic of the functional language. That is, the ADT operations always adhere (for sets R,S,T) to the axiomatic properties such as:

$$\begin{aligned} S \cup T &= T \cup S & S \cap T &= T \cap S \\ R \cup (S \cap T) &= (R \cup S) \cap (R \cup T) \end{aligned}$$

set is a type which is polymorphic in one argument: the user specifies the element type. Hence we would declare a set of numbers to be of type *aset num*. We give some example *set* operations, explaining the non-obvious ones:

- *mkaset* is a set constructor: a set must be instantiated by taking elements from a suitable structure in the Miranda program environment: in this case, a polymorphic list, denoted [*].

- *union* is set union; *union a b* gives the set which is the union of *a* and *b*.
- *card* gives set cardinality.
- *exists* is the existential quantifier; *exists s p* is equivalent to:
$$\exists x:s \cdot p(x)$$
- *forall* is the universal quantifier; *forall s p* is equivalent to:
$$\forall x:s \cdot p(x)$$
- *comp* is a set comprehension using a single predicate; *comp s p* is equivalent to:
$$\{x:s | p(x)\}$$
- *choose* (nondeterministically) returns an element from the set.
- *setdiff* gives the difference between two sets, i.e. *setdiff a b* returns those elements in *a* not in *b*.

See appendix 1 for the Miranda *abstype* signature for *set*.

3.2 The First Specification

At this stage we have developed sufficient structure to introduce a model for “the software system”. The model used is based on that of the Z language [10], using central concepts of a (global) system state, and operations on that state. We use type *set* to specify a simple system for maintaining a set of items: adding, removing, listing elements, and testing whether an item belongs. Our example is a plane boarding system, representing the plane as a set of passenger names, and a plane capacity. The state is defined as a Miranda type, i.e. the set of all possible values of the following pair:

```
plane == (aset person, num)
```

person is some user-defined type that need not be elaborated here. We next introduce the concept of “state invariant” as a constraint which specifies legal values of this type. It is expressed as a predicate on the state which must always be true; as such, it provides a useful verification mechanism during execution of the specification. In this case, we have

```
invariant::plane -> bool
invariant (onboard, capacity)
  = card onboard <= capacity & natural capacity
```

[PERSON]

```
Plane
|
| onboard:P person
| capacity:N
|
| #onboard <= capacity
```

Note that the predicate is a compound of simpler predicates, joined with binary boolean operators. Bracketing is not required on the right hand side of the definition since function application binds tightest in the Miranda operator precedence hierarchy. This definition states that plane occupancy must not exceed its capacity, and that the capacity must be a natural number.

Intuitively, an initial state is required for the plane, which can be produced by the following operation definition:

```
init_plane::num -> plane
init_plane n = (emptyset, n)
```

emptyset is the constant, empty set (of any type) defined by the ADT. The concepts of operation and associated precondition definition are explained in the following example, that of boarding a person onto a plane. The operation is a function with 1) an input argument giving the new information for this operation, i.e. the person to be boarded, 2) a state argument for the input state, and 3) a result argument for the updated, output state:

```
board::person -> plane -> plane
board per (onboard, capacity)
  = (union onboard (mkaset [per]), capacity)
```

```
Board
|
| ΔPlane
| per?:PERSON
|
| onboard' = onboard U {per?}
| capacity' = capacity
```

Hence the operation model is:

```
op::<input args> -> state -> state
```

The precondition is introduced as a predicate that 1) tests the combination of input arguments and state for operation adherence to the invariant, and 2) establishes whether the operation “makes sense”, i.e. delimits the partiality (if any) of the operation. Our example precondition nicely exemplifies this definition since its first predicate (plane not full) matches (1) above, and its second predicate (new person not already on board) matches (2):

```
pre_board::person -> plane -> bool
pre_board person (onboard, capacity)
  =   card onboard < capacity &
      ~ belongs_to person onboard
```

In Z:

```
#onboard < capacity ∧ per? ∉ onboard
```

Hence the precondition model is:

```
pre::<input args> -> state -> bool
```

3.3 A Bigger Example

We construct further set-theoretic structure to provide more expressiveness in specification. The next stage is an ADT for a “function”, or “map” type (we use the latter term to avoid overloading the Miranda term *function*). Operations:

- *mkamap* is the constructor, instantiating a map (polymorphic in two types) from a list of pairs (denoted $[(*,*)]$)
- *dom* and *ran* return the domain and range of the given map, respectively.
- *drestrict* and *rrestrict* represent domain and range restriction, respectively.
- *apply* provides function application.
- *mapunion* provides a union facility for maps.
- *mapcard* - give cardinality of map; comments as for *mapunion*.

See appendix 1 for the Miranda *abstype* signature for *map*.

With *map*, we have more expressive power in specification. We will briefly look at a larger version of the previous example, an aeroplane seat booking system. An intuitive model is now (*bplane* for “better plane”):

```
bplane == (aset seat, amap seat person)
```

The set of seats enables us to cater for various plane types, with differing set numbering schemes. A mapping from seat to person is a natural way of both avoiding multiple bookings for one seat, and enabling block booking in one name.

The invariant shows use of the universal quantifier *forall*. It requires that 1) any booked seat must exist on the plane, and that 2) no person books more than 3 seats:

```
invariant::plane -> bool
invariant (seats, bookings)
  = subset (dom bookings) seats &                                (1)
    forall (ran bookings) (f bookings)                          (2)
  where
    f mp elt
      = mapcard (rrestrict (mkaset [elt]) mp) <= 3
```

[SEAT]

BPlane seats:P SEAT bookings:SEAT → PERSON dom bookings ⊆ seats ∀ p:ran bookings • #(bookings ▷ {p}) <= 3

Predicate (1) is straightforward; predicate (2) usefully exemplifies the curried style of function definition. The first argument of the universal quantifier operation *forall* gives the set it applies to, i.e. the set of persons who have booked seats. The second argument is the constraining predicate which each element must satisfy. This predicate function is locally defined (under the *where* clause) and needs two arguments: one from the top level (the *map* under consideration, i.e. *bookings*), and an element argument. Function *f* has type:

```
f::amap seat person -> person -> bool
```

and hence the expression *f bookings* above has type:

```
f bookings::person -> bool
```

An example operation is to book (*bpbook* for “better-plane-book”) a seat to a person. Any free seat can be associated with the

person; the pair can then be map-unioned (as a singleton map) into the bookings map. A free seat must be chosen (nondeterministically) from seats outside the domain of the bookings map.

```
bpbook::person -> plane -> plane
bpbook p (seats, bookings)
  = (seats, mapunion newmp bookings)
  where
    newmp
      = mkamap [(choose (setdiff seats (dom bookings)), p)]
```

<pre> BPBook ┌ ΔBPlane per?:PERSON └ ┌ seats' = seats ∃ s:(seats - (dom bookings)) • bookings' = bookings ∪ {s /-> per?} └ </pre>
--

The precondition has two predicates. Firstly, seats must exist which are not booked. The invariant guarantees that

```
subset (dom bookings) seats
```

hence we need only check non-equality of the two sets. Secondly, we must check that this person has booked less than three seats so far:

```
pre_bpbook::person -> plane -> bool
pre_bpbook p (seats, bookings)
  = ~ equals (dom bookings) seats &
    mapcard (rrestrict (mkaset [p]) bookings) < 3
```

In Z:

```
~ (dom bookings = seats) ∧ #(bookings ▷ {per?}) < 3
```

3.4 Nondeterminism in Operations

As discussed in section 1, the executable style of specification presented is purely functional. Operation *bpbook* above has nondeterminism, which is why the correspondence between the Miranda and Z versions breaks down. Z, using the existential quantifier, does not constrain the seat which is booked; it simply states that one of the set of free seats will be allocated. In Miranda, we must compute an answer, i.e. *choose* a seat. Note that that *set*

512 Software Quality Management

“operation” *choose* is non-functional: for a given set, the element chosen is implementation-dependent.

We need a Miranda notion of operation postcondition, a constraining relation between inputs, before- and after-state spaces, to address this. The approach is then to treat the precondition as an entry test, and the postcondition as an exit test, of the candidate operation. Any operation meeting the two conditions will suffice. We are thus expanding our approach beyond the class of deterministic specifications. This is an important step, not least because any significant system will have nondeterministic aspects, and specification should avoid any implementation bias if possible. Section 1 refers to Hayes’ [5] examples of this wider class.

We define a functional model of the postcondition as a predicate on inputs, after- and before-states:

```
post_op::<args> -> <state> -> <state> -> bool
```

Operation *bpbook* now has a postcondition which corresponds to the predicates of Z schema *BPBook*:

```
post_bpbook::person -> plane -> plane -> bool

post_bpbook p (s,b) (so,bo)
  = exists (setdiff so (dom bo)) f
    where
      f seat = eq s so &
        mapeq b (mapunion bo (mkamap [(seat,p)]))
```

Thus we now have a general model for nondeterministic operations. An improved *bpbook* could be obviously defined using the model:

```
user_op args state_o
  = (state, output), if pre_op args state_o &
                        post_op args state state_o
    where
      (state, output) = op args state_o
```

3.5 Interaction

Miranda, as a functional language, lacks a mechanism for stored variables. The technique used to preserve system state is one of preserving the changing state value as an argument to a recursively-invoked top-level function. We do not need to explain this issue further in this paper.

Also, built-in interaction support is minimal: a function application is made at the Miranda prompt, and the resulting value, possibly complex, is output directly.

Hence it is not possible for the user to interface directly with her executable specification/ prototype in any but the most cumbersome manner (For clarity, we will refer to the combination of the executable specification and the interface as the “prototype”). This was the motivation for our HCI-builder tool.

The HCI-builder is an interactive application written in Miranda. It addresses both of the above problems, and is simply and elegantly implemented, using a functional model of interaction suggested by Thompson [11]. It provides the user with the capability of quickly assembling an appropriate interface for her specification. It assumes a simple but flexible model of the user interface to the specification.

That is, the prototype will present an initial menu of options. Selection of an option leads to either 1) a subsidiary menu, 2) an operation-specific screen, or 3) a screen requesting more information, e.g. identity of state component to operate on. Screen (2) takes operation arguments, and returns output after operation execution and state update. Alternately, screen (3) takes further identifying information, and leads to an operation-specific screen.

The tool thus provides for user construction of a hierarchy of menus, data entry, and operation screens appropriate to the specification. Once complete, the interface definition is written to file in the form of Miranda constant definitions. The second job for the user is to provide standard conversion functions between interface input and output fields (type *[char]*), and operation arguments and results. Finally, the interface definition script and the script of run-time screen interface driver functions are linked into the user specification script. The latter two steps are of course guided by a suitable template and worked examples.

We have not discussed error handling. The tool provides for this. In the specification, errors are handled by extending operation definitions with suitable guard clauses. Each error guard should be the logical negation of some part of the pre- and post-condition predicates. An example error guard for *bpbook*:

`equals (dom bookings) seats`

The Z-literate will recognise that this approach corresponds with the schema calculus approach of disjoining the operation schema with suitable error-handling schemas.

4. CONCLUSION

Miranda was chosen as the vehicle for this development for historical reasons, but the language does impose some minor expressive limitations. The formally-minded reader will wonder why definitions *mapunion* and *mapcard* in section 3.3 are required - a map is simply a specialised set, and hence all set operations should be applicable. Since Miranda has no type hierarchy or subclassing mechanism, there is no way *abstype map* can inherit *set* operations as we might wish. Such a capability would require porting to languages such as Mark P. Jones' Gofer, or Haskell.

Further note the user predicate *natural* in section 3.2. This is required because Miranda's *num* type represents all reals (within implementation limits). We cannot describe sets such as the naturals or integers without cumbersome predicates. Gofer and Haskell also address this issue, by providing a suitable hierarchy of number types.

This development has been influenced by Hekmatpour's EPROS [6]. This system inputs and interprets VDM specifications, and addresses the issue of interface prototyping by providing an extended finite-state machine notation for interface specification.

Possible work for the future includes porting to a language such as Haskell, completing the modelling of predicate calculus and Z structures in Miranda, and enhancing the interface model, perhaps as per EPROS.

5. References

- 1 Alexander, H., Jones, V., *Software Design and Prototyping using me too*, Prentice Hall, 1990.
- 2 Baxter, S.P.L., *Executing Z Specifications*, British Telecom internal report RT31/009/88, 1988.
- 3 Fuchs, N.E., "Specifications are (preferably) executable", *Software Engineering Journal*, September 1992, pp. 323-334.
- 4 Hall, A., "Seven Myths of Formal Methods", *IEEE Software* (special issue on Formal Methods), September 1990, pp. 11-19.
- 5 Hayes, I.J., Jones, C.B., "Specifications are not (necessarily) executable", *Software Engineering Journal*, Vol 4 (6), November 1989, pp. 330-338.
- 6 Hekmatpour, S., Ince, D., *Software Prototyping, Formal Methods and VDM*, Addison-Wesley, 1988.
- 7 Henderson, P., "Functional Programming, Formal Specification, and Rapid Prototyping", *IEEE Transactions on Software Engineering*, Vol. SE-12 (2), 1986, pp. 241-249.
- 8 Jones, C.B., *Systematic Software Development using VDM*, Prentice Hall, 1990.
- 9 O'Neill, G., *Rapid Prototyping of Formal Specifications using Miranda*, National Physical Laboratory Report DITC 150/89, November 1989.
- 10 Spivey, J.M., *The Z Notation: A Reference Manual*, Prentice Hall, 1989.
- 11 Thompson, S., "Interactive functional programs: A method and a formal semantics", Chapter 10, *Research Topics in Functional Programming*, ed. Turner, D.A., pp. 249 - 286, Addison-Wesley, 1990.
- 12 Turner, D.A., "Functional Programs as Executable Specifications", *Mathematical Logic and Programming Languages*, ed. Hoare, C.A.R., Stephenson, J.C., Prentice Hall International, 1985.
- 13 Turner, D.A., "Miranda: A non-strict functional language with polymorphic types", *Lecture Notes in Computer Science Volume 201: Functional Programming Languages and Computer Architecture*, ed. Jouannaud, J.P., Goos, G., Hartmannis, J., Springer, 1985.

516 Software Quality Management

Appendix 1 - Abstract Type Signatures

A partial *abstype* signature for *set* is:

```
abstype aset *
with
  emptyset::aset *
  mkaset::[*] -> aset *
  eq::aset * -> aset * -> bool
  card::aset * -> num
  belongs_to::* -> aset * -> bool
  subset::aset * -> aset * -> bool
  union::aset * -> aset * -> aset *
  exists::aset * -> (* -> bool) -> bool
  forall::aset * -> (* -> bool) -> bool
  comp::aset * -> (* -> bool) -> aset *
  choose::aset * -> *
  setdiff::aset * -> aset * -> aset *
```

A partial *abstype* signature for *map* is:

```
abstype amap * **
with emptymap::amap * **
  mkamap::(*,**) -> amap * **
  dom::amap * ** -> aset *
  ran::amap * ** -> aset **
  rrestrict::aset ** -> amap * ** -> amap * **
  apply::amap * ** -> * -> **
  mapunion::amap * ** -> amap * ** -> amap * **
  mapcard::amap * ** -> num
```