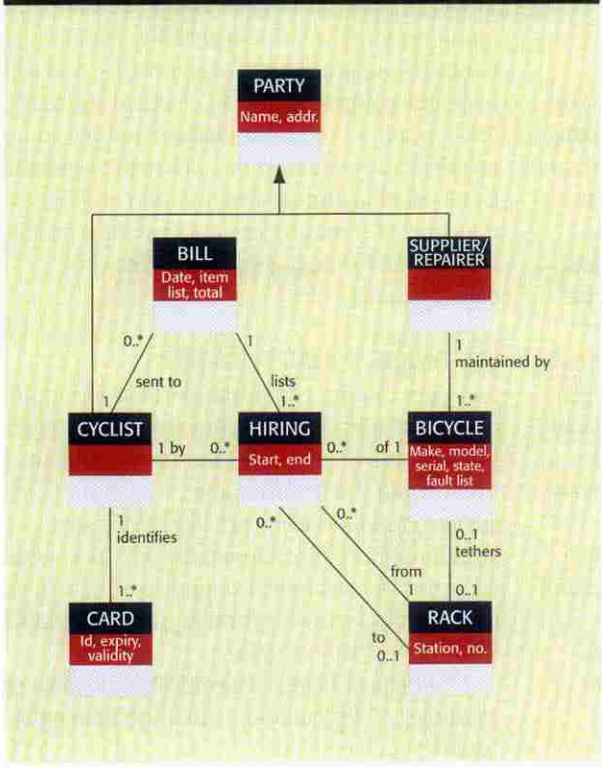
THE BEST OF BOTH WORLDS
INTEGRATING UML WITH Z FOR SOFTWARE SPECIFICATIONS

It is well known that the discipline of software engineering has been bedevilled by quality problems for decades, and that a substantial proportion of faults are subsequently traced back to deficiencies in specifications. A major cause of specification deficiencies is the use of natural language text, with its inherent scope for errors due to ambiguity, contradiction etc. In other branches of engineering this problem is greatly reduced by making extensive use of both diagrams and equations: in both cases there are common notations that are universally understood. Yet software engineering has largely avoided mathematical notations altogether, whilst producing a wide variety of competing diagram types with associated methodologies.

In recent years matters have been significantly improved by the growing popularity of the UML (Universal Modelling Language) notation, which is suitable for documenting both software specifications and designs. UML is fast becoming a *de facto* standard.

FIG. 1 CLASS DIAGRAM



For example, I recently read a telecommunications specification liberally scattered with UML diagrams, but without any mention that they were UML diagrams. It was assumed that the reader would recognise them. This development is very welcome but is not by itself sufficient, because there are important aspects of software specifications that cannot readily be expressed by diagrams.

By contrast the formal specification language Z, although longer established than UML, has only been employed in a small minority of software projects, usually where the customer has insisted on the use of formal methods because the application concerned is critical to safety.

After gleaning some practical experience of both UML and Z, the author evolved a method combining both notations that utilises the strengths of each to compensate for the weaknesses of the other, and so minimises the need for natural language text in software specifications. This method has been employed for specifying a variety of small extra features added to a large Unix based telecommunications system.

EQUIVALENCE EXAMPLE

To explain the equivalence between UML and Z it seems best to explore a small example. Imagine a software-controlled system for hiring out bicycles from railway stations. Typically a commuter would ride his own bicycle to his local station, then at his destination hire a bicycle from a rack to complete the journey to his

INTEGRATING THE UNIFIED MODELLING LANGUAGE WITH THE FORMAL SPECIFICATION LANGUAGE Z CAN PRODUCE SOFTWARE SPECIFICATIONS THAT EXHIBIT THE STRENGTHS OF BOTH NOTATIONS.

by Stephen Martin

office in town. (Normally, overcrowding on commuter services precludes carriage of bicycles on these trains.)

Each cyclist is given a swipe card to operate the locks that tether the bicycles to the station racks. The tethering chain incorporates an electronic tag to identify the bicycle. The details of these technologies are not relevant to the example, but it can be assumed that the combined effect is to track the hiring of bicycles and permit billing. A cyclist may only hire one bicycle at a time.

A possible class diagram is shown in Fig. 1, and the start of an equivalent Z specification in Fig. 2 (with line numbering to facilitate referencing from this text). The class diagram is the UML diagram that most overlaps with Z, and this should be reinforced by using matching names for classes and sub-schemas, associations and functions. The upper portion of the box for each class contains its name, the middle portion its attributes (internal data) and the lower portion its operations. Fig. 1 omits the class operations. This is common when a class diagram is first drawn. The operations are determined later (typically when creating the sequence diagrams) and then added to the class diagram. An asterisk annotating an association between classes stands for 'many'.

Observe that each class with non-trivial attributes has both a Z sub-schema and a finite set to represent its instances. For the 'rack' class these are at lines 26–28 and 35. Each association between classes is represented by a function. For one to many associations a function can either map from the many class to the one class (e.g. 'sent to') or from the one class to the many class (e.g. lists — note that the range elements in these case are finite sets).

- *Constraints and invariants:* In this example, there is clearly a restriction on the participation of instances of the 'bicycle' class in the associations with hirings and racks. A hired bicycle is not a bicycle that is tethered to a station rack and vice versa. This type of information is known in UML as a constraint and cannot be captured on a class diagram such as Fig. 1. Instead an OCL statement would be needed. In Z it is easily expressed with an invariant statement that the intersection of the hired and ready sets of bicycles must be empty; see line 45 of Fig. 3.

Another constraint arises from the bill–cyclist–hiring 'loop'. Each hiring is both recorded on a bill and associated with a specific cyclist. Yet not all possible pairings are permissible: a hiring must obviously only appear on a bill that is destined for the cyclist concerned! This constraint is expressed in Z with an invariant statement that for every bill, for every hiring on that bill, the hirer must be the cyclist that the bill is sent to (line 53 of Fig. 3).

- *Sequences and states:* Fig. 4 is a sequence diagram for the 'hire' use case where the outcome is successful. →

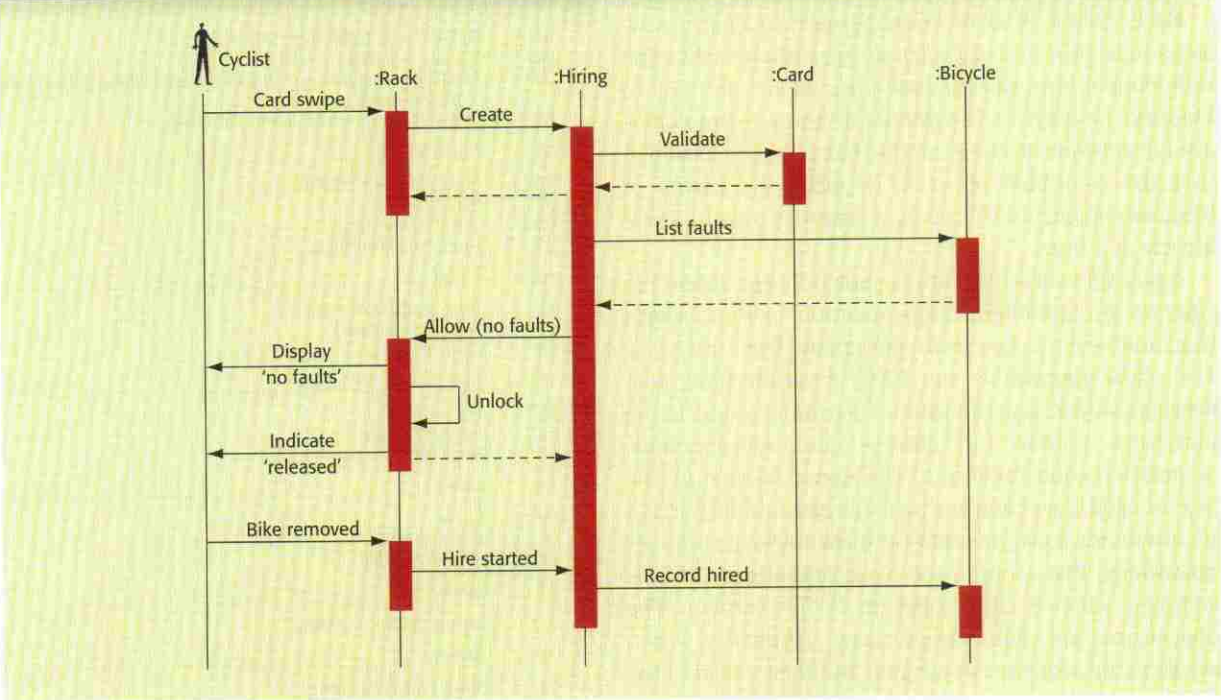
FIG. 2 Z SPECIFICATION FOR BICYCLE HIRE EXAMPLE (UPPER PORTION ONLY)

```
01  [DATE, TIME, DATETIME, STRING]
02  Validity ::= valid | invalid
03  FaultType ::= puncture | brakes | gears | wheel | chain | steering |
      pedal | other
04  CyclesState ::= ready | hired | in repair
05  ΔBicycle
06  Make, Model : STRING
07  Serial : N
08  State : CycleState
09  FaultList : F FaultType
10  ΔItem
11  Start, End : DATETIME
12  From, To : STRING
13  Cost : R
14  ΔBill
15  Date : DATE
16  Items : F ΔItem
17  Total : R
18  ΔCard
19  Id : N
20  Expiry : DATE
21  Status : Validity
22  ΔParty
23  Name, Address : STRING
24  ΔHiring
25  Start, End : DATETIME
26  ΔRack
27  Station : STRING
28  No : N
29  ΔBikeHire (main schema)
30  readyBikes, hiredBikes, bikesInRepair: F ΔBicycle
31  bills : F ΔBill
32  cards: F ΔCard
33  cyclists, supplierRepairers: F ΔParty
34  hirings: F ΔHiring
35  racks: F ΔRack
36  hireOf : ΔHiring → ΔBicycle
37  hireBy : ΔHiring → ΔParty
38  hireFrom, hireTo : ΔHiring → ΔRack
39  tethers : ΔRack → ΔBicycle
40  identifies : ΔCard → ΔParty
41  lists : ΔBill → F ΔHiring
42  maintainedBy : ΔBicycle → ΔParty
43  sentTo : ΔBill → ΔParty
```

FIG. 3 Z SPECIFICATION INVARIANTS FOR BICYCLE HIRE EXAMPLE (SELECTION) AND SPECIFICATION FOR 'RECORD HIRED' OPERATION

```
44  (invariants)
45  readyBikes ∩ hiredBikes = ∅
46  readyBikes ∩ bikesInRepair = ∅
47  hiredBikes ∩ bikesInRepair = ∅
48  dom hireOf = dom hireBy = dom hireFrom = hirings
49  dom hireTo ⊆ dom hireFrom
50  ran tethers = readyBikes
51  dom lists = dom sentTo = bills
52  ∀h : ran lists • h ⊆ hirings
53  ∀b : bills • ∀h : lists b • hireBy h = SentTo b
54  ∀c : cyclists • #{(dom (hireBy > c))\(dom hireTo)} <= 1
55  dom maintainedBy = readyBikes ∪ hiredBikes ∪ bikesInRepair
56  Record hired (operation)
57  readyBikes readyBikes'
58  hiredBikes hiredBikes'
59  bicycle? : ΔBicycle
60  (pre-conditions)
61  bicycle? ∈ readyBikes ∧ bicycle?.State = ready
62  (post-conditions)
63  bicycle?.State' = hired ∧ readyBikes' = readyBikes\{bicycle?} ∧
64  hiredBikes' = hiredBikes ∪ {bicycle?}
```

FIG. 4 SEQUENCE DIAGRAM FOR A SUCCESSFUL HIRE USE CASE



(Possible other outcomes of the hire use case include an invalid or expired card, a cyclist declining to hire a faulty bicycle and a cyclist not removing a bicycle in the time allowed before the rack re-locks it.) There is no direct Z equivalent. In drawing up sequence diagrams, operations are invented and can then be added to the class diagram. So in this example the 'card' class acquires a 'validate' operation, for instance.

Sequence diagrams are also valuable in discovering the lifecycles of classes, which can then be documented with state diagrams. In this example the 'bicycle' has a significant lifecycle and a possible state diagram for it is given in Fig. 5. This Figure illustrates many of the features of state diagrams. The 'maintenance' states have been separated as concurrent states because of a decision to allow bicycles with faults to be hired at the cyclist's discretion. The state diagram indicates that the fault list is cleared every time the 'fault-free' state is entered and that the list is updated whenever a fault is reported. The diagram also shows that the 'list faults' operation can be invoked in any state, but the 'record hired' operation will only be actioned if the bicycle is in the 'ready' state.

The outer box on Fig. 5 encloses the entire lifetime of an object instance. The solid dots within it specify that when a new bicycle object is created the parallel states initialise to the combination 'ready' and 'fault free'.

States can be handled in Z by listing them with an enumerate and including state attributes in sub-schemas. Alternatively there can be a subsets for the instances in each state. Both these approaches are illustrated in Fig. 2 for the bicycle class (see lines 4, 8 and 26). In this case, however, the 'fault free' condition always corresponds to an empty set fault list, so an explicit maintenance state attribute has not been declared. By contrast, transitions between states can only be indirectly specified in Z. This is done by recording the before and after states as pre and post-conditions of operations.

• *Actions and operations:* Although state diagrams identify where particular actions are required (within states, upon entry to or exit from states, or on transitions between states) they cannot specify what an action does. Likewise for operations identified on sequence diagrams. However, activity diagrams can be used to describe how actions or operations (or indeed whole use cases) should proceed.

The Z definition of an operation comprises its input and output parameters and its pre and post-conditions. The pre-conditions specify the validation required before allowing the operation to execute and the post-conditions specify the changed condition of the system after it has completed (see Fig. 3, lines 56–64, which specifies the 'record hired' operation of the bicycle class.) Lines 57 – 58 declare that the readyBikes and hiredBikes sets are changed by the operation. Line 59 declares that there is only one input to the operation — the bicycle in question. The pre-conditions state that the bicycle must be in the readyBikes set and that its state attribute must have the value 'ready' (thus agreeing with Fig. 5) for the operation to be allowed. The post-conditions state that the state attribute has been changed to the value 'hired' and that the bicycle has been removed from the readyBikes set and added to the hiredBikes set.

PICKING THE BEST AND PLUGGING THE GAPS

Although, as has been demonstrated, there is considerable overlap between UML and Z, there are aspects of software specification that only one notation covers, or for which one notation is clearly superior to the other. These cases are now summarised.

• *Constraints and invariants:* UML has had to be augmented with OCL to capture association constraints. As a type of pseudocode OCL is prone to the problems inherent in natural language specifications. The Z equivalent, the invariant statement, does not have this disadvantage and is usually more concise. Therefore invariants are to be preferred for documenting constraints.

• *States and sequences:* In Z, it is easy to declare a set of states, but hard to define the transitions between them. The state transitions can only be gleaned by studying a set of operations. Working out possible legal sequences of events is even harder. Clearly the UML state and sequence diagrams are essential for documenting these aspects of a software specification.

• *Action and operation specifications:* In UML, operations are identified on the class diagram and actions on the state diagram, but the only means of specifying either is with an activity diagram. The activity diagram can also define a whole use case (although some methods use structured English for this). Operations can also be defined in Z, but here the definition of 'operation' is broader and can encompass actions and use cases as well as true class operations.

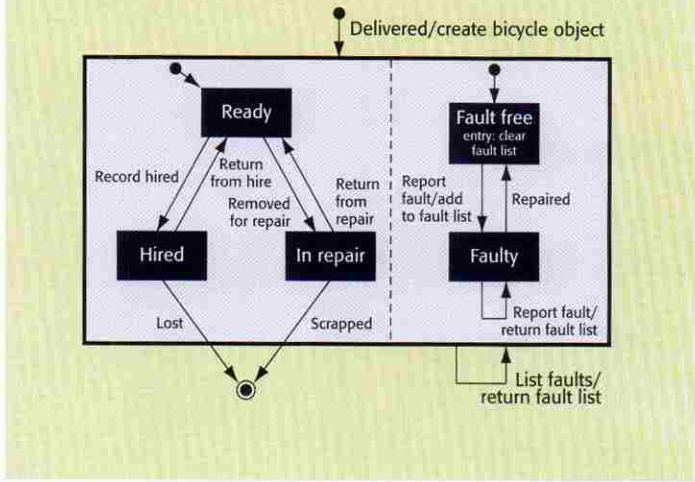
The difference between the activity diagram and the Z operation is this. In Z it is impossible to define how an operation should execute: only the pre-conditions that must be met for it to go ahead and the conditions that must pertain after it has completed. Hence this yields a pure specification, uncontaminated by any implementation detail. By contrast the UML activity diagram is not far removed from a flow chart. It really defines an algorithm, although it does have some novel features e.g. support for concurrent processing and an efficient means of representing iteration. Nevertheless it is very hard to draw an activity without constraining the implementation.

Hence activity diagrams should be used only where it is necessary to enforce a particular algorithm on the programmer. In normal circumstances a Z operation is to be preferred.

PUTTING IT ALL TOGETHER

In summary, work normally starts with the use case diagram. Next the components into which the system is partitioned are identified (in an object-oriented project these will be classes). From this point work proceeds in parallel to produce the UML class and sequence diagrams and the Z sub-schemas and data declaration part of the main schema. Thereafter careful consideration of the class diagram should identify both constraints (to be recorded as Z invariants) and 'live' classes (which will each require a state diagram). The actions on the state diagrams and the operations of classes are specified as Z operations. Optionally, activity diagrams are created to define the algorithms for actions, operations or whole use cases (but only where it is necessary to prescribe the implementation).

FIG. 5 STATE DIAGRAM FOR THE BICYCLE CLASS



The differing fortunes to date of UML and Z may be attributed to their separate origins in the disparate academic and commercial worlds. From the programmer's viewpoint the advent of UML promises a welcome standardisation of competing notations, many diagrams of which were of long standing and therefore quite familiar. By contrast, the mathematical notation of Z often appears daunting to the average programmer. Yet most people will have been taught basic set theory and Boolean algebra at school, whilst predicate calculus is usually included (in the form of 'assertions') in tertiary computer science and software courses. With these foundations, the task of learning Z is not intrinsically any more difficult than learning a programming language such as C.

Given its compatibility with Z (as demonstrated in this article) the growing popularity of UML presents an opportunity to pull through formal methods into mainstream commercial projects. This would be a step towards a more professional future for software engineering. Specifiers who combine UML with simple Z, by employing methods such as the one outlined in this article, should find that they get the best of both worlds! ■

Stephen Martin worked 20 years in telecommunications software for Plessey, OPT and Marconi.