Z and Abstract Machine Notation: A Comparison

Antoni Diller and Rosemary Docherty

Abstract

In this paper we compare the formal specification languages Z and Abstract Machine Notation (AMN); the latter of which is due to Abrial. The strategy adopted is that of presenting the same specification both in Z and AMN and of commenting on salient differences as they arise. The specification chosen is a slightly revised version of the specification of an Internal Telephone Number Database found in chapter 4 of [4]. At the end of the paper some general conclusions are drawn.

Antoni Diller and Rosemary Docherty, "Z and Abstract Machine Notation: A Comparison" in Jonathan P. Bowen and J. Anthony Hall (eds.), *Z User Workshop: Cambridge 1994*, [London, Springer-Verlag, 1994], pp. 250-263.

Z and Abstract Machine Notation: A Comparison

Antoni Diller and Rosemary Docherty*

January 18, 1994

Abstract

In this paper we compare the formal specification languages Z and Abstract Machine Notation (AMN); the latter of which is due to Abrial. The strategy adopted is that of presenting the same specification both in Z and AMN and of commenting on salient differences as they arise. The specification chosen is a slightly revised version of the specification of an Internal Telephone Number Database found in chapter 4 of [4]. At the end of the paper some general conclusions are drawn.

1 Introduction

Z is a well-established, formal specification language. It was developed about 15 years ago, has had many books and articles written about it and there is a sizable community of users both in academia and also in industry. Abrial's Abstract Machine Notation (AMN) is, by contrast, a recent innovation with only a handful of users and little has been published about it. One of the aims of this paper is to introduce the person familiar with Z to AMN and to compare these two specification languages. In such a short paper it is impossible to mention every interesting feature of AMN, but we hope to convey the "feel" of a specification written in AMN.

Our goal in writing this paper is not primarily to try and convince Z users to convert to being users of AMN. However, in addition to introducing AMN to a wider audience we also hope to add to the current debate in the Z community about whether or not object-oriented features should be incorporated into Z and if they should how this ought to be done. A study of AMN can help someone appreciate the issues involved as AMN is built around the equivalent in the realm of specification of what in the realm of programming language theory is known as a *module* or a *class* or a *package*.

The approach taken in this paper is to present a specification of a small system both in Z and in AMN and to comment on some of the important

*We would like to express our gratitude both to the SERC and to BP International Limited for financial support in the form of a CASE studentship awarded to Rosemary Docherty. BP International Limited also provided much of the software needed for Rosemary Docherty's research and for that we are very grateful.

differences between them. The system chosen for this purpose is that of an Internal Telephone Number Database found in chapter 4 of [4], though slight changes have been made to the specification found there.

2 The Internal Telephone Number Database

In this section we give an informal description in English of the functionality of the system to be specified:[1]

> An organisation wants to computerise its internal telephone directory. The database must keep a record of all the people who are currently members of the organisation (as only they can have telephone extensions). The database must cope with the possibility that one person may be reached at several extensions and also with the possibility that several people might have to share an extension. Six operations are to be provided, namely:
>
> (1) Adding a person to the database's record of who is currently a member of the organisation.
>
> (2) Removing a person from the database.
>
> (3) Adding an entry to the database, where an *entry* is an association between a person and a telephone extension.
>
> (4) Removing an entry from the database.
>
> (5) Interrogating the database by person.
>
> (6) Interrogating the database by extension number.

3 Specification Prologue and Initialisation

In this paper the *prologue* of a specification refers to that part of it which excludes the operations. The prologue, so to speak, sets the scene for the specification of the operations that we are interested in. First, the prologue will be given in Z and then it will be presented using Abrial's AMN.[2]

The specification of the telephone directory makes use of two basic types, which in Z are introduced by means of a basic type definition as follows:

$$[PEOPLE, PHONES].$$

$PEOPLE$ is the type of all people and $PHONES$ is the type of all possible telephone numbers. We also need a set of messages. This is called $REPORTS$

[1]This is based on the informal account that is found near the beginning of chapter 4 of [4].

[2]In [1], pp. 3–6, what we call the prologue is referred to as the *statics*. Furthermore, Abrial calls the definition of the operations in an abstract machine the *dynamics*. This terminology is not used of Z specifications, so we decided to use the neutral term *prologue*.

and it can be defined by means of a free type definition like this:

$$\begin{aligned} REPORTS ::= {} & Okay \\ \mid {} & Already\_Member \\ \mid {} & Not\_Member \\ \mid {} & Entry\_Already\_Exists \\ \mid {} & Unknown\_Entry \\ \mid {} & Unknown\_Name \\ \mid {} & Unknown\_Number. \end{aligned}$$

The state of the internal telephone number database is given by means of the schema *PhoneDB*:

$$\begin{array}{|l} \textit{PhoneDB} \\ \hline \textit{members} : \mathbb{P}(PEOPLE) \\ \textit{telephones} : PEOPLE \leftrightarrow PHONES \\ \hline \operatorname{dom}(\textit{telephones}) \subseteq \textit{members} \\ \hline \end{array}$$

The set *members* consists of all the members of the organisation and the relation *telephones* records the information of who can be reached at which extension. An element of the relation *telephones* will be called an *entry* and an example of such an entry is the ordered pair or maplet $diller \mapsto 4794$, which tells us that *diller* can be reached at the extension whose number is 4794. The state invariant of the schema *PhoneDB* is the single formula:

$$\operatorname{dom}(\textit{telephones}) \subseteq \textit{members}.$$

This tells us that everyone who has a telephone extension must be a member of the organisation.

Although not part of the prologue, it is convenient to mention the schemas $\Delta PhoneDB$, $\Xi PhoneDB$ and $InitPhoneDB'$ here. The schema $\Delta PhoneDB$ is a convenient way of referring to all the variables needed to specify an operation. It is defined as follows:

$$\Delta PhoneDB \mathrel{\widehat{=}} PhoneDB \wedge PhoneDB'.$$

The schema $\Xi PhoneDB$ is used in the specification of operations that do not change the state of the database:

$$\begin{aligned} \Xi PhoneDB \mathrel{\widehat{=}} [\Delta PhoneDB \mid {} & members' = members \wedge \\ & telephones' = telephones]. \end{aligned}$$

The initial state of the system is $InitPhoneDB'$, where $InitPhoneDB$ is defined in this way:

$$\begin{array}{|l} \textit{InitPhoneDB} \\ \hline \textit{PhoneDB} \\ \hline \textit{members} = \emptyset \\ \textit{telephones} = \emptyset \\ \hline \end{array}$$

A Z specification can be thought of as defining an abstract data type. However, as [10], p. 129, puts it, 'For a specification to describe a genuine abstract data type, there must be at least one possible initial state.' In the case of the telephone number database, the requirement that there be at least one possible initial state is satisfied if the following can be proved to hold:

$$\exists PhoneDB' \bullet InitPhoneDB'.$$

Written out in full this formula looks like this:

$$\begin{aligned}&\exists members' : \mathbf{P}(PEOPLE); telephones' : PEOPLE \leftrightarrow PHONES |\\ &\quad \mathrm{dom}(telephones') \subseteq members' \bullet members' = \emptyset \wedge telephones' = \emptyset.\end{aligned}$$

This is true as it simplifies to $\mathrm{dom}(\emptyset) \subseteq \emptyset$.

The specification prologue in AMN should be readily understood by anyone familiar with Z. The specification is as follows:

machine

$\quad Phone$

sets

$\quad PEOPLE; PHONES; REPORTS = \{\ Okay, \ldots\ \}$

variables

$\quad members; telephones$

invariant

$\quad members \subseteq PEOPLE \wedge$

$\quad telephones \in PEOPLE \leftrightarrow PHONES \wedge$

$\quad \mathrm{dom}(telephones) \subseteq members$

initialisation

$\quad members, telephones := \emptyset, \emptyset$

operations

$\quad\quad \vdots$

end

One of the major differences between Z and AMN is that in AMN the entire specification of a single software system is packaged up into an abstract machine—which may well make use of other machines—that is a well-defined entity in its own right. In the above (incomplete) machine, the prologue consists of everything apart from what occurs in the **initialisation** and **operations** components. What goes into the **operations** component in order to complete the machine will be explained in section 6 below. According to [1], p. 1, the notion of an abstract machine is one of the central features of AMN and it is the analogue in the context of specification of what—in the context of programming language description—is known as 'a *class* (SIMULA), or a *module* (MODULA), or a *package* (ADA)'.[3] A minor difference between Z and AMN can be discerned here and it is that AMN does not use the typed predicate calculus which Z uses.

[3]'Ada' is a registered trademark of the US Government (Ada Joint Program Office).

The name of the abstract machine displayed above is *Phone*. In the **sets** component of the machine are found both the basic types of the corresponding Z specification, namely $PEOPLE$ and $PHONES$. The collection of messages, namely $REPORTS$, is also placed there. What occurs in the **variables** component of the machine corresponds to the variables declared in the state schema *PhoneDB*. The state invariant from that schema is found in the **invariant** component of the machine, where any other information, such as type statements, constraining the values of the variables *members* and *telephones* also occurs. It should be noted that the symbol := that occurs in the **initialisation** component of the machine is not assignment but substitution and $members, telephones := \emptyset, \emptyset$ refers to the simultaneous substitution of $\emptyset$ for both the variables *members* and *telephones*. The substitution is performed on a description of the state of the system being specified. Whereas Z is a model-based specification language, AMN is built on the idea of textual substitution.

When using the B-method to write a specification in AMN the specifier needs to discharge certain proof obligations concerning the specification that he or she is writing.[4] One of these relates to the **initialisation** component of the abstract machine and the formula that has to be proved is $[T]\ I$, where T is the initialisation and I is the invariant.[5] In the specification being discussed here this amounts to the following formula:

$$\begin{array}{l} [members, telephones := \emptyset, \emptyset] \\ \quad members \subseteq PEOPLE \wedge \\ \quad telephones \in PEOPLE \leftrightarrow PHONES \wedge \\ \quad \mathrm{dom}(telephones) \subseteq members, \end{array}$$

which simplifies to the formula

$$\emptyset \subseteq PEOPLE \wedge \emptyset \in PEOPLE \leftrightarrow PHONES \wedge \mathrm{dom}(\emptyset) \subseteq \emptyset,$$

which is easily proved to be a theorem of set theory.

Another proof obligation relates to the invariant. It has to be shown that the invariant is not trivially false. That is to say, it has to be shown that $\exists x \cdot I$, where x is the list of variables of the abstract machine and I is the invariant.[5] In the case of the current specification this is the following formula:

$$\begin{array}{l} \exists members, telephones \cdot \\ \quad members \subseteq PEOPLE \wedge \\ \quad telephones \in PEOPLE \leftrightarrow PHONES \wedge \\ \quad \mathrm{dom}(telephones) \subseteq members, \end{array}$$

which is easily seen not to be trivially false.

4 The Operations

The operation of successfully adding a member of the organisation to the database is specified in Z by means of the schema *AddMember*:

[4]Note that whereas AMN is a *language* that is used to write specifications of software systems, the expression 'the B-method' is used to refer to the *way* in which—according to Abrial—AMN should be used to write such specifications.

[5]In the general case this proof obligation is more complicated because an abstract machine may contain a **constraints** and a **properties** component. See [1] for more details.

```
─ $AddMember$ ──────────────────────────
│ $\Delta PhoneDB$
│ $name? : PEOPLE$
├─────────
│ $name? \notin members$
│ $members' = members \cup \{name?\}$
│ $telephones' = telephones$
└───────────────────────────────────────
```

Here, *name*? is the person who we are trying to add to the database. It is possible in Z to obtain a precondition schema from any schema describing an operation which makes use of the convention that before variables are unprimed and after variables are primed. This is done by hiding or existentially quantifying over the after and output variables. Doing this here—and simplifying what is obtained—results in the schema *PreAddMember*:

```
─ $PreAddMember$ ───────────────────────
│ $PhoneDB$
│ $name? : PEOPLE$
├─────────
│ $name? \notin members$
└───────────────────────────────────────
```

There is no requirement when producing a specification written in Z to work out what the precondition schema is.

In AMN the operation of making someone known to the database is specified like this:

```
$AddMember\ (name\_in) \mathrel{\widehat=}$
    pre
        $name\_in \in PEOPLE \wedge$
        $name\_in \notin members$
    then
        $members := members \cup \{name\_in\}$
    end
```

A minor difference between Z and AMN is that AMN does not have the convention that input variables are decorated with a question mark. The operation *AddMember* (*name_in*) is defined to be a *pre-conditioned substitution*, that is to say, it is a substitution that is only carried out if the formulas that occur between the key words **pre** and **then** are satisfied. Note that if these formulas are not satisfied, then anything might happen.

When the B-method is being used to write AMN specifications, then whenever an operation is specified there is a proof obligation to be carried out. This involves proving that the invariant is preserved by the operation, that is to say, the formula $I \wedge Q \Rightarrow [S]I$ has to be established, where I is the invariant and the operation is **pre** Q **then** S **end**.[5] In this case this is the formula:

$$\begin{array}{l}
(\mathit{members} \subseteq \mathit{PEOPLE} \wedge \\
\mathit{telephones} \in \mathit{PEOPLE} \leftrightarrow \mathit{PHONES} \wedge \\
\mathrm{dom}(\mathit{telephones}) \subseteq \mathit{members} \wedge \\
\mathit{name\_in} \in \mathit{PEOPLE} \wedge \\
\mathit{name\_in} \notin \mathit{members}) \Rightarrow \\
([\mathit{members} := \mathit{members} \cup \{\mathit{name\_in}\}] \\
\quad \mathit{members} \subseteq \mathit{PEOPLE} \wedge \\
\quad \mathit{telephones} \in \mathit{PEOPLE} \leftrightarrow \mathit{PHONES} \wedge \\
\quad \mathrm{dom}(\mathit{telephones}) \subseteq \mathit{members}).
\end{array}$$

The proof of this is straightforward—if a little tedious.

The specification of the operation labelled (2) in the informal description of the system being specified—given in section 2 above—is straightforward and introduces no new ideas.

As already mentioned, when using the B-method a proof obligation has to be discharged whenever an operation is specified in AMN. When using Z, however, there is no such obligation. Ensuring that the state invariant is preserved is left entirely at the intuitive level. Thus, it would be possible to include in a Z specification the following schema as an attempt to specify operation (3):

```
┌─ WrongAddEntry ─────────────────────────────
│ ΔPhoneDB
│ name? : PEOPLE
│ number? : PHONES
├──────────
│ name? ↦ number? ∉ telephones
│ telephones′ = telephones ∪ {name? ↦ number?}
│ members′ = members
└─────────────────────────────────────────────
```

This is perfectly acceptable to *f*UZZ, as *f*UZZ is only a syntax and type checker and so cannot be expected to catch this kind of error. However, if *name*? is not an element of *members*, then the predicate-part of this schema is equivalent to the constant formula $false$, because the state invariant $\mathrm{dom}(\mathit{telephones}) \subseteq \mathit{members}$ is not preserved: $\mathrm{dom}(\mathit{telephones}')$ contains *name*? which is absent from $\mathit{members}'$. In such a simple case it is not too difficult to see that something is not correct about *WrongAddEntry*, but in a large and complicated specification it is easy to overlook such faults. The following is a correct version of this operation:

```
┌─ AddEntry ──────────────────────────────────
│ ΔPhoneDB
│ name?: Person
│ newnumber?: Phone
├──────────
│ name? ∈ members
│ name? ↦ newnumber? ∉ telephones
│ telephones′ = telephones ∪ {name? ↦ newnumber?}
│ members′ = members
└─────────────────────────────────────────────
```

It would be possible to specify an operation in the AMN *language* that is incorrect in the same way that $WrongAddEntry$ is incorrect, but by using the B-*method* to develop specifications such an incorrect specification would be quickly discovered because the relevant proof obligation could not be discharged.

The specification of (4) is straightforward and introduces no new ideas. The two operations (5) and (6) do, however, introduce some new ideas when specified in AMN. Operation (5) is specified in Z by means of the schema $FindPhones$:

$$\begin{array}{|l}
\multicolumn{1}{l}{FindPhones} \\ \hline
\Xi PhoneDB \\
name? : PEOPLE \\
numbers! : \mathbb{P}(PHONES) \\ \hline
name? \in \mathrm{dom}(telephones) \\
numbers! = telephones(\!|\{name?\}|\!) \\ \hline
\end{array}$$

In AMN this looks as follows:

```
$numbers\_out \longleftarrow FindPhones\ (name\_in) \mathrel{\widehat{=}}$
    **pre**
        $name\_in \in \mathrm{dom}(telephones)$
    **then**
        $numbers\_out := telephones[\{name\_in\}]$
    **end**
```

The novel feature introduced here is the way in which outputs are indicated in AMN. They occur first, followed by a left-pointing arrow and then comes the name of the operation being defined. Thus, the variable $numbers\_out$ here is the output of the operation $FindPhones$ for the input $name\_in$. Two minor differences between Z and AMN to notice here are that AMN does not have the Z convention of indicating output variables by decorating them with an exclamation mark and, secondly, that the syntax for the relational image of a set through a relation is slightly different in the two notations.

The inclusion of the schema $\Xi PhoneDB$ in $FindPhones$ tells us that that operation does not involve any changes being made to the values of the state variables. In the case of the AMN specification, this means that no substitutions are made on the state variables.

Operation (6) is specified analogously to operation (5).

5 Completing the Specification in Z

The schema $AddMember$ tells us what happens only if the precondition is satisfied. In order to complete the specification so that something sensible happens even when the precondition is not satisfied we need the schema $AlreadyMember$, which is defined like this:

```
┌─ AlreadyMember ─────────────────────────
│ ΞPhoneDB
│ name? : PEOPLE
│ rep! : REPORTS
├──────────
│ name? ∈ members
│ rep! = Already_Member
└──────────────────────────────────────────
```

If we try to add a name to the set *members* which is already present, then a suitable error message is output. In order to provide some sort of indication that the operation of adding a name to *members* has been successful we define a schema *Success*:

```
┌─ Success ───────────────────────────────
│ rep! : REPORTS
├──────────
│ rep! = Okay
└──────────────────────────────────────────
```

It is now possible to specify the total operation *DoAddMember* like this:

$$DoAddMember \mathrel{\widehat=} AddMember \land Success \lor AlreadyMember.$$

DoAddMember, unlike *AddMember*, is defined for all conceivable inputs. The specification of the other total operations is similar, although further error schemas need to be defined. The actual schemas involved in all this are uncomplicated and are summarised in the following table:

operation	total specification	successful outcome	error schemas
(1)	*DoAddMember*	*AddMember*	*AlreadyMember*
(2)	*DoRemoveMember*	*RemoveMember*	*NotMember*
(3)	*DoAddEntry*	*AddEntry*	*NotMember* *EntryAlreadyExists*
(4)	*DoRemoveEntry*	*RemoveEntry*	*UnknownEntry*
(5)	*DoFindPhones*	*FindPhones*	*UnknownName*
(6)	*DoFindNames*	*FindNames*	*UnknownNumber*

6 Completing the Specification in AMN

It is not surprising that *AddMember* (*name_in*), *FindPhones* (*name_in*), and the other AMN operations which describe successful operations are all placed in the **operations** component of the abstract machine *Phone*. We also need to decide where we are going to put the specification of errors. We have decided to put them into *Phone*. It may seem strange at first sight to include operations that output error messages in the abstract machine *Phone*, but the reason for doing so is that these operations have the same invariant as do the other operations, like *AddMember* (*name_in*) and *FindPhones* (*name_in*). Schematically, therefore, the machine *Phone* looks like this:

machine
  Phone
sets
  $PEOPLE; PHONES; REPORTS = \{\ Okay, \ldots\ \}$
variables
  $members; telephones$
invariant
  $members \subseteq PEOPLE\ \wedge telephones \in PEOPLE \leftrightarrow PHONES\ \wedge$
  $\mathrm{dom}(telephones) \subseteq members$
initialisation
  $members, telephones := \emptyset, \emptyset$
operations
  $AddMember\ (name\_in) \mathrel{\widehat{=}} \ldots\ ;$
  $RemoveMember\ (name\_in) \mathrel{\widehat{=}} \ldots\ ;$
  $AddEntry\ (name\_in, number\_in) \mathrel{\widehat{=}} \ldots\ ;$
  $RemoveEntry\ (name\_in, number\_in) \mathrel{\widehat{=}} \ldots\ ;$
  $numbers\_out \longleftarrow FindPhones\ (name\_in) \mathrel{\widehat{=}} \ldots\ ;$
  $names\_out \longleftarrow FindNames\ (number\_in) \mathrel{\widehat{=}} \ldots\ ;$
  $rep\_out \longleftarrow AlreadyMember\ (name\_in) \mathrel{\widehat{=}} \ldots\ ;$
  $rep\_out \longleftarrow NotMember\ (name\_in) \mathrel{\widehat{=}} \ldots\ ;$
  $rep\_out \longleftarrow EntryAlreadyExists\ (name\_in, number\_in) \mathrel{\widehat{=}} \ldots\ ;$
  $rep\_out \longleftarrow UnknownEntry\ (name\_in, number\_in) \mathrel{\widehat{=}} \ldots\ ;$
  $rep\_out \longleftarrow UnknownName\ (name\_in) \mathrel{\widehat{=}} \ldots\ ;$
  $rep\_out \longleftarrow UnknownNumber\ (number\_in) \mathrel{\widehat{=}} \ldots$
end

The operation which just outputs a message saying that a successful outcome has been achieved is placed in a machine called *PhoneCtx*. It is a different kind of operation from the others encountered so far because it is stateless and that, indeed, is the reason for putting it in a separate machine. One of the most important parts of a machine is the description of the state, since the operations describe substitutions to be performed on this, therefore it seems sensible to distinguish between operations on this state and others, and to place these two different kinds of operation in separate machines.

machine
  PhoneCtx
uses
  Phone
operations
  $rep\_out \longleftarrow Success \mathrel{\widehat{=}}$ **begin** $rep\_out := okay$ **end**
end

Note the presence here of a **uses** component in the machine *PhoneCtx*. This is one of several ways in which one machine can make use of another one. It means that the machine *PhoneCtx* can access the variables, sets and constants introduced in the machine *Phone*, but it cannot alter the values of any of the variables introduced in *Phone*. The constant *Okay* that is used in *PhoneCtx* is introduced in the machine *Phone*.

In order to complete the specification of the Internal Telephone Number Database in AMN we define the machine *DoPhone* as follows:

```
machine
    $DoPhone$
includes
    $Phone, PhoneCtx$
operations
    $rep\_out \longleftarrow DoAddMember\ (name\_in) \mathrel{\widehat{=}} \ldots\ ;$
    $rep\_out \longleftarrow DoRemoveMember\ (name\_in) \mathrel{\widehat{=}} \ldots\ ;$
    $rep\_out \longleftarrow DoAddEntry\ (name\_in, number\_in) \mathrel{\widehat{=}} \ldots\ ;$
    $rep\_out \longleftarrow DoRemoveEntry\ (name\_in, number\_in) \mathrel{\widehat{=}} \ldots\ ;$
    $rep\_out, numers\_out \longleftarrow DoFindPhones\ (name\_in) \mathrel{\widehat{=}} \ldots\ ;$
    $rep\_out, names\_out \longleftarrow DoFindNames\ (number\_in) \mathrel{\widehat{=}} \ldots$
end
```

Note the presence of an **includes** component in the machine *DoPhone*. This means that that machine can access the variables, sets and constants introduced in the machine *Phone* and *PhoneCtx* and, furthermore, it can also alter the values of any variables introduced in those machines, but it can only do this by calling the operations introduced in those machines. The reason for this restriction is that it allows us to ensure that the invariants of the included machines are maintained by any operations defined in the including machine.

DoAddMember (*name_in*), *DoRemoveMember* (*name_in*) and the other total operations are all built up out of the corresponding partial operations and appropriate error operations as you would expect. A single example should suffice to explain how this is done:

```
$rep\_out \longleftarrow DoAddMember\ (name\_in) \mathrel{\widehat{=}}$
    pre
        $name\_in \in PEOPLE$
    then
        if $name\_in \notin members$
        then
            $AddMember\ (name\_in) || rep\_out \longleftarrow Success$
        else
            $rep\_out \longleftarrow AlreadyMember\ (name\_in)$
        end
    end
```

The symbol || that occurs in this specification represents simultaneous substitution. Note also the presence of the *conditional substitution*

if P **then** S **else** T **end**,

which has the effect that the substitution S is performed if P is true, but T is carried out if P is false.

7 Conclusion

In this paper we have compared some aspects of the formal specification languages AMN and Z and the main differences that we have found between them will be summarised here:[6]

(1) This first point relates to the large-scale organisation of specifications. (As refinement has not been mentioned above the following remarks do not apply to it.) There is a precise description of the syntax of Z—found, for example, in [10], pp. 142–146—and according to this a Specification is a well-defined unit. There are no ways, however, of combining two specifications—except that of concatenating them.[7] (In doing this it may also be necessary to rearrange the order of some of the specification components and to remove duplicate items.) In AMN, by contrast, an abstract machine is a well-defined unit and there are several carefully thought-out ways of combining machines. Two of these have been illustrated in this paper. We have shown, for example, the way in which one machine **uses** another and also the way in which one machine **includes** other machines. There are still further ways of relating machines to one another.

(2) Z is based on first-order logic and set theory and many people who advocate the use of Z say that one of the advantages of using a mathematical notation is that it is possible to prove things about a formal specification, but not much guidance is given to the specifier to help him or her decide which properties of a specification are worth proving and which are not. In the B-method, however, certain properties of a specification are singled out as proof obligations and if these are carried out, then the specifier knows that the specification he or she has written has a number of desirable properties. In this paper, for example, we have mentioned the following proof obligations:

   (a) that the substitution in the **initialisation** component of an abstract machine establishes the invariant;

   (b) that the invariant is not trivially false; and

   (c) that each operation in an abstract machine preserves the invariant.[8]

[6]Further comparisons between Z and AMN—some of which overlap with those contained here—can be found in [5].

[7]This comment refers to standard Z as defined in Spivey's book. Much research is currently being done to see how object-oriented features can be added to Z. See, for example, [6] and [2].

[8]When using Z there is no need to prove that the invariant is maintained and—as shown above—this may cause problems.

Further proof obligations need to be discharged when more complicated things are being done. This is especially useful, for example, in a non-academic environment where there may be various pressures at work and in such circumstances it may be useful to know that all necessary proof obligations have been carried out.

(3) In specifying an operation in Z by means of a schema you need to explicitly mention whether or not a variable is affected by means of the operation. For example, in the schema *AddMember* above we have had to include the formula $telephones' = telephones$ to indicate that this operation does not change the value that the variable *telephones* has before and after the operation has been carried out. When a large specification is being written with a large number of variables it becomes tedious to have to include all variables whose values are left unchanged by an operation. In AMN, by contrast, there is no need to mention variables that are unchanged by an operation.

(4) The preconditions of an operation specified in AMN must be explicitly mentioned, whereas in a Z schema the preconditions are not singled out in any way. It is possible, however, to calculate the precondition schema from any schema used to specify an operation.

There are also further differences between Z and AMN that have not appeared in the small example considered in this paper. Some of the most important of these will be mentioned here:

(5) Formal specifications of real-life systems tend to become large and they are difficult to manipulate in an error-free way by hand. It is highly desirable, therefore, to have some sort of machine support to help with the manipulation of specifications. In the case of Z the provision of such machine support is in an early stage of development. Some of the existing tools are: *f*UZZ (a syntax and type checker which can also typeset Z specifications), CADiZ (a suite of tools whose functionality is similar to that of *f*UZZ), ProofPower and Zola (both of which are claimed to be able to carry out some proofs about Z specifications). With AMN, however, the situation is very different. There exist several sophisticated pieces of software, namely the B-Tool and the B-Toolkit, which greatly help with the production of specifications written in AMN. Moreover, if the B-method is being used, then given a specification written in AMN the B-Toolkit will generate all the proof obligations that need to be discharged with respect to it and it is able to discharge many of these automatically. A small amount of human assistance may be needed for a complicated proof obligation to be discharged.

(6) There is currently no universally accepted way in the Z community of relating a low-level specification to program code, though several approaches are being investigated. (For some of these, see [3], [7],[9] and [11]. This is not meant to be an exhaustive list.) The situation is very different in the case of AMN. Here a low-level specification is related to program code by means of Dijkstra's weakest-precondition calculus and

[9]This paper relates Z and the refinement calculus expounded in [8].

this is integrated into the entire specification-design-implementation process associated with the AMN language.

We draw no conclusions as to whether Z or AMN is the better specification language. This question really does not have a clear sense. Specification languages are complicated systems with large numbers of properties and it is possible to compare them in many different ways and along several dimensions. It is ridiculous to attempt to give a single, unified, global rating to either Z or AMN. The purpose of this paper is to draw the reader's attention to AMN, which—we believe—is not as well known as it should be and especially to its modularisation features.

References

[1] Abrial JR. Abstract machines: Part I: Basic concepts—introduction, 1991. Draft.

[2] Carrington D. ZOOM workshop report. In Nicholls [9], pp 352–364.

[3] Diller A. Z and Hoare logics. In Nicholls [9], pp 59–76.

[4] Diller A. Z: An Introduction to Formal Methods. Wiley, Chichester, second edition, 1994. Forthcoming.

[5] Docherty R, Diller A. CAVIAR in AMN. Research Report CSR–93–3, School of Computer Science, University of Birmingham, 1993.

[6] Duke R, King P, Rose G, Smith G. The Object-Z specification language: Version 1. Technical Report 91–1, Software Verification Research Centre, The University of Queensland, 1991.

[7] King S. Z and the refinement calculus. Technical Monograph PRG–79, Programming Research Group, Oxford University Computing Laboratory, 1990.

[8] Morgan C. Programming from Specifications. Prentice Hall International Series in Computer Science, edited by C. A. R. Hoare. Prentice Hall, Hemel Hempstead, 1990.

[9] Nicholls JE (ed). Z User Workshop: York 1991. Springer-Verlag, London, 1992.

[10] Spivey JM. The Z Notation: A Reference Manual. Prentice Hall International Series in Computer Science, edited by C. A. R. Hoare. Prentice Hall, Hemel Hempstead, second edition, 1992.

[11] Wordsworth JB. Software Development with Z: A Practical Approach to Formal Methods in Software Engineering. Addison-Wesley, Wokingham (England), 1992.