From Z to BON/Eiffel

Richard F. Paige and Jonathan S. Ostroff
Department of Computer Science, York University
Toronto, Ontario M3J 1P3, Canada.
{paige,jonathan}@cs.yorku.ca

Abstract

It is shown how to make a transition from the Z formal notation [3] to the Business Object Notation (BON) [4]. It is demonstrated that BON has the expressive power of Z, with the additional advantages of object-orientation and a supporting seamless development method. The transition is illustrated for some widely used Z constructs. The translation lays the groundwork for a semi-automated tool for extracting classes from Z specifications.

1. Introduction

The Z formal specification notation [3], with a rich collection of concepts based on typed set theory, is receiving growing attention. One noted limitation with Z is that it can be hard to relate specifications to implementations. Techniques have been developed for Z to bridge the gap between specification and implementation—for example, via translation to a wide-spectrum language, or by embedding a small programming language in Z—but these approaches suffer from problems.

- A bridge to an executable language must still be made. If an object-oriented implementation is desired, the transition from Z will be all the more complex.
- The notations used in bridging the gap may be inaccessible to practitioners, and likely will be weakly supported by industrial strength software tools.

With these limitations in mind, we are interested in showing the feasibility of bridging the gap from Z to implementations without all of the problems inherent in previous approaches. In particular, we want to link Z with BON [4], which is part of a *seamless development method* that can result in Eiffel programs, thus providing the ability to develop object-oriented programs from Z specifications.

As an auxiliary goal, we also aim to show that BON has the expressive capabilities of Z with the advantages of object-orientation and a seamless development method.

Finally, the transition from Z to BON lays the groundwork for a semi-automated tool for objectifying Z, by extracting class specifications from Z specifications.

The link between Z and BON is based on defining translations between the notations. We illustrate the principle by giving translations for some widely used Z constructs in Section 3. Further details are in [2].

Because Z is not an object-oriented notation, the mapping from Z to BON must produce classes. This is done by, effectively, following a standard Z style of specification, which encompasses first specifying system state, and then operations that use the state.

2. Overview of BON

We assume some familiarity with Z; the standard Z reference is [3]. We summarize some key BON concepts here.

BON is a simple graphical and textual notation for specifying and describing object-oriented systems. It provides mechanisms for specifying inheritance and client-supplier relationships between classes, and has a collection of techniques for expressing dynamic relationships. The notation also includes a predicative assertion language for specifying preconditions and postconditions of class features.

BON is supported by a rich set of tools [1], and is designed to work seamlessly with the Eiffel programming language. An implication of this is that in BON specifications the Eiffel class libraries can be used.

Here is an example of a BON specification of a class *CITIZEN* (BON also has an equivalent graphical syntax; we use the textual notation in this short paper for conciseness). The class has four attributes, a *BOOLEAN* query *single*, and a state-changing command *divorce*. **require** clauses are preconditions, and **ensure** clauses are postconditions. Postconditions can refer to the value of an expression when the feature was called by prefixing the expression with the keyword **old**. Classes may also have *invariants*, which are predicates that must be maintained by all visible routines. Visibility of features is expressed by annotating **feature** clauses with lists of permitted client classes.

class *CITIZEN* **feature** {*NONE*}
name, *sex*, *age* : *VALUE*
spouse : *CITIZEN*
feature {*ANY*}
single : *BOOLEAN* **ensure** *Result* = (*spouse* = *Void*) **end**
divorce
require not *single*
ensure *single* **and** (**old** *spouse*).*single* **end**
end

3. Translating Z to BON

We now outline a semi-automatable scheme for translating a Z specification into a BON specification. A practical difficulty with the translation is that Z is not object-oriented, unlike BON. In the translation, we must therefore produce classes from collections of related Z schemas. The relationship that we use is that if operation schemas share state schemas, then the translated operations and shared variables should belong to the same class.

In translating Z to BON, there are three significant Z concepts that we have to be able to translate: the *Z toolkit*, which includes basic types, constructors, and operators; Z *state schemas*, which are used to specify the data of a system; and Z *operation schemas*, which specify system operations. We illustrate the translation from Z to BON by example. Further details are available in [2].

3.1. Translating Z toolkit features

The Z toolkit is rich and substantial. We will not attempt to write down a complete translation into BON, but will illustrate a general approach by example.

A useful Z toolkit construct is the *function type*. Consider the state schema *BirthdayBook*, below.

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{dom}\, birthday$$

birthday is a partial function from domain *NAME* to range *DATE* (where *NAME* and *DATE* are declared sets which can be translated into BON deferred classes). Partial functions can be extended, restricted, and applied to arguments. There is no equivalent to function types in BON, so we must formulate one using BON's assertion language.

To represent Z function types, we first represent tuples using the generic BON class *PAIR* parameterized by two types *F* and *G*.

class *PAIR*[*F*, *G*] **feature** *first* : *F* *second* : *G* **end**

We introduce a generic class *FUNC*, shown in Figure 1, which is a translation of the Z function type (the inherited query *has* in Figure 1 is set membership). *FUNC inherits* from class *SET*; a function *is-a* set of ordered pairs. In translating the full toolkit, a class would be produced for each construct.

Other toolkit features can be translated in a similar manner. We envision producing an object-oriented BON library that expresses the concepts of the Z toolkit.

3.2. Translating Z state schemas

A Z state schema describes the data that are to be used in a system. Here is an example for translating the schema *BirthdayBook* introduced in Section 3.1. *BirthdayBook* can be translated into the following BON class (visibility clauses may also be added).

class *BIRTHDAY_BOOK* **feature**
known : *SET*[*NAME*]
birthday : *FUNC*[*NAME*, *DATE*]
invariant *known* = *birthday*.*domain*
end

The general translation from a state schema to a BON class is as follows. Let *S* be the following schema.

$$S \mathrel{\widehat=} [\, a_1 : T_1;\ \ldots;\ a_k : T_k \mid P \,]$$

(The T_i are Z types, and *P* is a predicate on prestate.) *S* is translated to the following textual BON class, under the assumption that the types $T_1, \ldots, T_k$ can be translated into BON types or classes.

class *S* **feature** $a_1 : T_1$ ... $a_k : T_k$ **invariant** *P* **end**

where *P* in the class invariant of *S* is a syntactic translation of the Z predicate *P* into BON's predicate notation.

A state schema can include others (by writing the name of a schema in its declaration part). Such hierarchies can be flattened by substituting the body of the included schema for all occurrences of its name. Schema inclusion can be translated by first flattening the hierarchies. An alternative approach is to treat inclusion as a *has-a* relationship, and to translate each state schema as a separate class, ignoring inclusion. The schema inclusion can then be translated to a *client-supplier* relationship [4], where the included schema is a supplier to the including schema. The latter approach can preserve some of the structure of the Z specification.

class $FUNC[D, R]$ **inherit** $SET[PAIR[D, R]]$ **feature**
$domain : SET[D]$ **ensure** $Result = \{d : D \mid \exists p : PAIR[D, R] \bullet p.first = d \wedge has(p)\}$ **end**
$select(d : D) : R$ —apply function to element d
require $\exists p : PAIR[D, R] \mid p.first = d \bullet has(p)$
ensure $\exists p : PAIR[D, R] \mid p.first = d \bullet has(p) \wedge Result = p.second$
end
$override(d : D;\ r : R)$ —function override
ensure $\forall q : \{p : PAIR[D, R] \mid (has(p) \vee p.first = d \wedge p.second = r) \wedge (p.first = d \rightarrow p.second = r)\} \bullet has(q)$
end
invariant $\forall d : D, r, r_1 : R \mid (select(d) = r \wedge select(d) = r_1) \bullet (r = r_1)$
end

Figure 1. A BON function type class

3.3. Translating Z operation schemas

An operation schema represents some operation that the system can perform. Typically, an operation schema *includes* a state schema, and may modify the system state in some way. Here is an example. An operation given in the birthday book example in [3] is *AddBirthday*, which adds a new name and date to the birthday book.

AddBirthday
$\Delta BirthdayBook$
$n? : NAME;\ d? : DATE$

$n? \notin known$
$birthday' = birthday \oplus \{n? \mapsto d?\}$

The schema includes (via Δ convention) schema *BirthdayBook*. The Δ means include two versions of *BirthdayBook*, one with all variables unprimed, and the other with all variables annotated with primes. $n?$ and $d?$ are inputs to the operation. The operation establishes a final state where *birthday* has been extended (via the override operation $\oplus$) to include the mapping from $n?$ to $d?$.

Translating an operation schema requires transforming the schema to a routine of a class. This class is the result of translating a state schema that is affected by the operation. In the example, *BirthdayBook* is changed by *AddBirthday*, thus operation schema *AddBirthday* is translated into a command of class *BIRTHDAY_BOOK*.

$AddBirthday(name : NAME;\ date : DATE)$
require $name \notin known$
ensure $birthday = (\textbf{old}\ birthday).override(name, date)$
end

It is possible that two or more operation schemas can include some of the same state schemas. When translating these schemas, we have to decide to which class the operations should belong. A simple syntactic algorithm can be used, based on the occurrence of a state schema in an operation schema. This algorithm would form the basis of a semi-automated tool that maps Z into BON.

The algorithm parses a Z specification consisting of an arbitrary number of schemas. If two operation schemas include some or all of the same state schemas, then the operations *as well* as the attributes of the state schemas should belong to the same class. The translator or the algorithm can decide the name of this class; the algorithm might simply choose the name of the first included state schema. This simple approach may lose some of the structure of the Z specification. Therefore, it should be used to give an approximation to a class design which can be further refined by developers to include hierarchical information.

The general translation from an operation schema into a feature is as follows. Let *Op* be an operation schema, and let *S* be one or more state schemas with variables *w*. Let *T* be zero or more state schemas that *Op* can use, but cannot change (expressed using the Ξ convention).

$$Op \mathrel{\widehat=} \left[\Delta S;\ \Xi T;\ i? : I;\ o! : O \mid P\right]$$

The operation has inputs $i?$ and produces $o!$. The state components of *S* and *T* will be mapped to attributes of a class. The schema *Op* will then be translated to a feature of the same class that takes the following form.

$Op(i : I) : O$
require $\exists w', o! \bullet P$
ensure $P[\textbf{old}\ w/w][w/w'][Result/o!]$
end

The **require** clause is the precondition of the operation schema *Op*. Existential quantification over the poststate reveals only those terms that constrain the precondition in *P*.

The **ensure** clause is the property of the operation schema, but with Z's primed-unprimed notation rewritten to BON's **old** notation. The **ensure** clause uses the Z substitution notation: $P[a/b]$ means "substitute a for b in P". Substitution is left-associative. Thus, **old** w is first substituted for w in P, and then w is substituted for w'.

If either schema inputs or outputs are omitted, then they are omitted from the BON translation. If $o!$ is missing, then *Op* is a command, and the last substitution involving *Result* can be dropped. An operation that has both a Δ inclusion and outputs corresponds to an operation that is both function and procedure. BON does not allow features with side-effects, so we cannot translate such schemas directly. Instead, we can make the operation results attributes of the class to which the translated command belongs.

Z operations can be specified by parts, using the *schema calculus*. To translate such composed operation schemas, we can flatten the compositions. This feature of Z is most frequently used to specify error conditions of operations. As we shall discuss later, this feature can be viewed as at odds with BON's design-by-contract.

3.4. Example: The Birthday Book

The *Birthday Book* example [3] is a well-known and standard problem for explaining the use of Z. A BON translation of the Z birthday book specification is given below as a single class, *BIRTHDAY_BOOK*.

class *BIRTHDAY_BOOK* **creation** *InitBirthdayBook* **feature**
known : *SET*[*NAME*] *birthday* : *FUNC*[*NAME*, *DATE*]
AddBirthday(*name* : *NAME*; *date* : *DATE*) **is**
require *name* $\notin$ *known*
ensure *birthday* = (**old** *birthday*)**.***override*(*name*, *date*)
end
FindBirthday(*name* : *NAME*) : *DATE*
require *name* $\in$ *known*
ensure *Result* = *birthday***.***select*(*name*)
end
Remind(*today* : *DATE*) : *SET*[*NAME*]
ensure *Result* = $\{n : NAME \mid birthday.select(n) = today\}$
end
InitBirthdayBook **ensure** *known***.***empty* **end**
invariant *known* = *birthday***.***domain*
end

Class *BIRTHDAY_BOOK* can be used to create as many instances of the *BIRTHDAY_BOOK* as needed. Schema *BirthdayBook* can also be used as a type, but to create multiple *BirthdayBook*s in Z, a set of *BirthdayBook*s must be specified, and *BirthdayBook* operations would have to be *promoted* [3] to manipulate this set. A data refinement of this class, as well as an Eiffel implementation, can be found in [2].

4. Discussion and Conclusions

We have outlined how Z specifiers can make a transition to object-oriented specifications via a mapping into BON. This transition integrates Z with a seamless development method, and can be used to suggest an object-oriented design for Z specifications. It provides a basis for semi-automatable tool that implements a mapping from Z to BON. An implementation of such a tool would be able to make use of the existing Eiffel environment.

The translation from Z to BON can suggest a disciplined way of using Z that is easy to map into BON. The translation suggests that each operation schema should represent a query or a command, and that commands should change the attributes of one state schema. By using Z in such a manner, the mapping into BON can be simplified.

The approach does have limitations. Translation of the schema calculus—and in particular, schema inclusion—may require unfolding of operations or inclusions before translation. It would be preferable to attempt to derive class structure from a Z specification, and to use inheritance to reduce the size of the resulting classes.

The integration of Z with BON bypasses a standard specification step: making Z operations total by applying the Z schema calculus. This is defensive specification that is obviated by design-by-contract. Z developers who are used to producing total operations may find it difficult to transition to BON. However, experience with Z in practice has shown that specifiers can be confused by the schema calculus. So by transitioning to BON, we could make Z more attractive to these developers.

In combining Z with BON, we have removed several of the noted limitations with Z, while acquiring the advantages of a seamless development method. Future work will explore the transition in larger case studies, as well as an implementation of the translation.

References

[1] B. Meyer. *Object Oriented Software Construction.* Prentice Hall, 1997.

[2] R. Paige and J. Ostroff. From Z to BON/Eiffel. Technical Report TR-98-05, York University, July 1998.

[3] J. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1992.

[4] K. Walden and J.-M. Nerson. *Seamless Object Oriented Software Architecture*. Prentice Hall, 1995.