From Z to BON/Eiffel

Richard F. Paige

Jonathan S. Ostroff

Technical Report CS-98-05

July 7, 1998

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

From Z to BON/Eiffel

Richard F. Paige and Jonathan S. Ostroff

Department of Computer Science, York University,
4700 Keele Street, Toronto, Ontario, Canada, M3J 1P3.
{paige,jonathan}@cs.yorku.ca

Abstract. It is shown how to make a transition from the formal specification notation Z [10] to the Business Object Notation (BON) [11], so as to be able to relate the former notation with object-oriented specifications and implementations. The transition is applied in a case study that shows how to move from Z to BON, and finally through to executable Eiffel programs. The translation lays the groundwork for a semi-automated tool that spans the semantic gap from abstract Z specifications to concrete Eiffel implementations.

1 Introduction

The Z formal notation [10] is receiving growing attention: in industry, where it is being used for the formal specification of large systems [3, 6]; and in academia, where it is being taught for formal specification and development. Z is a mathematical notation, with a rich collection of idioms based on typed set theory for specifying systems. It has seen successful use in writing and analyzing specifications, and in proving properties about specifications [4, 5].

Z has limitations. Its complex mathematical syntax has been found to have too steep a learning curve for some. It is not an object-oriented notation, which may be too limiting for modern software development. And, like many formal methods, it has been accused of producing incomprehensible, only expert-accessible specifications. One noted limitation of Z is that it is hard to relate specifications to implementations. A reason for this is that Z is not a wide-spectrum language; unlike the notation in [8], Z does not include a programming language subset. Techniques have been developed for Z to bridge the gap between specification and implementation, for example, via translation to a notation like specification statements [8], or by embedding a small programming language subset in Z [12], but these approaches suffer from several problems.

- A bridge to a concrete implementation, i.e., in an executable programming language like Eiffel or Java, must still be made. If an object-oriented implementation is desired, the transition from Z may be more complex, since Z specifications must be mapped into classes. This limitation is not obviated by Object-Z [2], the object-oriented Z dialect, since it too must be mapped to a concrete implementation.
- The approaches used may not scale up to large-scale system development. In particular, these approaches do not directly allow specification in terms of classes, which is important in producing extendible, reusable software [7].
- The notations used in bridging the gap may be inaccessible to practitioners, and likely will be weakly supported by industrial strength tools, like CASE diagrammers and theorem provers.

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 [11], which is part of a *seamless development method* that results in Eiffel programs. In linking these two notations, we have the following aims.

- To show how Z specifications can be related to correct, concrete, executable implementations.
- To provide the ability to develop object-oriented programs from Z specifications. Object-oriented programs have been proposed as essential for producing extendible and reusable systems [7]. By linking Z with BON, which can then be used in deriving Eiffel programs, the gap from Z to Eiffel may be reduced.
- To attempt to convince the reader that it need not be necessary to specify in Z—that it is sufficient, and in some cases more useful, to specify in BON from the start. One advantage of this is that specifications are object-oriented from the beginning.
- To lay the groundwork for a semi-automated tool that can span the semantic gap from abstract Z specifications to concrete Eiffel implementations.

The link between Z and BON is based on defining translations between the notations, and then by making use of the theory of [7] to develop Eiffel implementations from BON specifications. Translations from Z to BON are presented in Section 3, and make use of the existing Eiffel class libraries. Our concern with these translations is presenting mappings that allow us to link the techniques together. We are interested in showing the *feasibility* of linking Z with BON. We therefore do not attempt to show a complete mapping of Z constructs to BON and Eiffel constructs (space constraints also prevent us from giving full details). Here, we concentrate on demonstrating the use of one mechanism for linking the notations.

Because Z is not an object-oriented notation, whereas BON is object-oriented, the mapping from Z to BON must add information. This is done by, effectively, following a standard Z style of specification, which encompasses first specifying system state, then related system operations. The related specifications are mapped into a BON class. This is discussed more in Section 3.

After describing translations between Z and BON, we use the translations in solving the classic birthday book problem [10], so as to illustrate the approach. Finally, we summarize the technique, mention some observations, and propose avenues for future work.

2 Overview of Notations

2.1 The Z Formal Specification Notation

The formal notation Z is due to Abrial (a good reference is [10]); it is a notation based on typed set theory. The fundamental specification construct in Z is the schema, which groups declarations with predicates. There are two kinds of schemas: *state* schemas, which provide declarations and definitions of system state; and *operation* schemas, which specify operations upon the system state. Here is an example of a state schema (which we will revisit in Section 5):

$$
\begin{array}{|l}
\hline
\textit{BirthdayBook} \\
\textit{known} : \mathbb{P}\,\textit{NAME} \\
\textit{birthday} : \textit{NAME} \rightarrow \textit{DATE} \\
\hline
\textit{known} = \operatorname{dom} \textit{birthday} \\
\hline
\end{array}
$$

The schema part above the horizontal line contains the state declarations and inclusions for the schema; in particular, the schema above declares a set *known* (the birthdays recorded in the system) and a function *birthday*. The part below the line is the state invariant that must be obeyed by all operations that use the state.

An operation schema uses state schemas (and possibly other local declarations) in defining system operations. Here is an example of an operation schema that uses the state schema *BirthdayBook*.

$$
\begin{array}{|l}
\hline
\textit{AddBirthday} \\
\Delta \textit{BirthdayBook} \\
n? : \textit{NAME} \\
d? : \textit{DATE} \\
\hline
n? \notin \textit{known} \\
\textit{birthday}' = \textit{birthday} \oplus \{n? \mapsto d?\} \\
\hline
\end{array}
$$

The operation schema includes (via Δ convention) schema *BirthdayBook*. The Δ notation means (informally) include two versions of *BirthdayBook*, one with all variables unprimed, and the other with all variables annotated with primes. Primed variables denote final values, while unprimed variables denote initial values. $n?$ and $d?$ are inputs to the operation. Below the line in the operation schema is the predicate. The schema establishes a final state where the name $n?$ has been added to the set known, and where the function *birthday* has been extended (via the functional overriding operation $\oplus$) to include the mapping from $n?$ to $d?$.

Z specifications are typically constituted of a set of state schemas and operation schemas that use the state schemas. Large specifications are constructed by parts from smaller specifications, using the *schema calculus*. The calculus includes combinators like conjunction ($\wedge$), disjunction ($\vee$), restriction, and others, as described in [10]. These combinators are often used in making schema operations total (i.e., able to handle all values of inputs). The notation is often used in the context of an 'established strategy', which is a set of informal procedures designed to organize the construction of a specification, and to suggest how implementation is to occur. Part of the established strategy that is used in the case study in this paper is as follows.

1. *Describe the abstract state*, using a collection of one or more state schemas.
2. *Specify the initial state of the system,* as an operation schema. Usually, a proof is carried out to show that there is an initial state.
3. *Specify the successful case of each system operation,* as an operation schema. Error cases are ignored (e.g., when the set *known* is full in *AddBirthday*).
4. *Calculate schema preconditions,* in order to determine when each operation can be successfully applied. This is done by existentially quantifying each schema over its poststate.
5. *Specify the error cases* for each system operation, as operation schemas.
6. *Make the operations total.* This is usually done by disjoining the error case schemas with the operation schemas. This may introduce nondeterminism that should be documented at this point.
7. *Provide assistance to the reader,* by informally summarizing and documenting the purpose of each schema (this might be done with comment clauses in Eiffel).

Z is not object-oriented, though it has been suggested that it can be used to describe such systems [1]. In our translations, we shall map collections of schemas that are related by schema inclusion to a BON class.

2.2 The Business Object Notation

The Business Object Notation (BON) [11] 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 small collection of techniques for expressing dynamic relationships (e.g., message passing) among objects. The notation also includes an *assertion language* for specifying preconditions and postconditions of class methods. The assertion language is a simple dialect of predicate logic (that includes quantifiers).

BON is supported by a rich set of tools (e.g., EiffelCase [7]), and is designed to work seamlessly with the Eiffel programming language and its tools and libraries. An important implication of this is that in BON specifications the Eiffel class libraries can be used.

Here is a short example of a BON textual specification of a class *CITIZEN* (BON also has an equivalent graphical syntax that we do not use here—see [11]). The class has four attributes, and two methods: a query *single* (which results in a *BOOLEAN*) and a command *divorce*, which changes the state of an object. Class *SET* is a generic class from the Eiffel library, as is class *BOOLEAN*. Preconditions of methods are written as **require** clauses, and postconditions as **ensure** clauses. An important feature of BON assertions is that postconditions can refer to the value of any *expression* when method was called; this is done by prefixing an expression with the keyword **old** (see method *divorce*, below). Classes may also have *invariants*, which are predicates that must be maintained by all visible methods. Visibility of methods and attributes is expressed using the **feature** clause: **feature** $\{NONE\}$ means that all following class members are invisible outside of the class; **feature** $\{ANY\}$ means that all following class members are visible (omission of *ANY* is equivalent to including it).

class *CITIZEN* **feature** $\{NONE\}$
  $name, sex, age : VALUE$
  $spouse : CITIZEN$
  $children, parents : SET[CITIZEN]$
  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. The practical difficulties with the translation are twofold. First, Z is not object-oriented, while in BON the fundamental construct is the class. Therefore, in the translation, we must somehow 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 should belong to the same class (that also includes attributes from the shared state schemas). The second difficulty is in mapping Z's predicate notation into BON **require** and **ensure** clauses. A mapping is given in Section 3.2.

The Z notation includes a significant toolkit, containing basic types (e.g., integers, reals), type constructors, and set-based operators. The toolkit is rich and substantial; we do not attempt to write down a complete translation into BON's assertion language here. Due to space constraints, we assume that it is generally possible to map Z types into assertion language types. We mention one important translation here: for Z function types (i.e., the type of *birthday* in the schema *BirthdayBook*).

For representing Z function types, we first represent tuples using generic class *PAIR*, which takes two types *F* and *G*, and has two attributes.

$$
\begin{array}{l}
\textbf{class}\ \mathit{PAIR}[F, G]\ \textbf{feature} \\
\quad \mathit{first} : F \\
\quad \mathit{second} : G \\
\textbf{end}
\end{array}
$$

We then introduce a generic class *FUNC*, which is a translation of the Z function type. *FUNC inherits* from class *SET*; a function *is-a* set of ordered pairs. In translating the full Z toolkit, we would produce a BON class for each Z construct. Each class encapsulates the logic necessary to use the construct. Inheritance could be of further use here: class *FUNC* could inherit from a class *PFUNC* (a translation of the Z partial function type). In the following, comments are prefixed by a double dash, $--$.

$$
\begin{array}{l}
\textbf{class}\ \mathit{FUNC}[D, R]\ \textbf{inherit}\ \mathit{SET}[\mathit{PAIR}[D, R]]\ \textbf{feature} \\
\quad \mathit{domain} : \mathit{SET}[D]\ \textbf{ensure}\ \mathit{Result} = \{ d : D \mid \exists\, p : \mathit{PAIR}[D, R] \bullet p.\mathit{first} = d \wedge \mathit{has}(p) \}\ \textbf{end} \\
\qquad\quad -- \mathit{returns\ the\ domain\ of\ the\ function} \\
\quad \mathit{select}(d : D) : R \\
\qquad \textbf{require}\ \exists\, p : \mathit{PAIR}[D, R] \mid p.\mathit{first} = d \bullet \mathit{has}(p) \\
\qquad \textbf{ensure}\ \exists\, p : \mathit{PAIR}[D, R] \mid p.\mathit{first} = d \bullet \mathit{has}(p) \wedge \mathit{Result} = p.\mathit{second} \\
\qquad\quad -- \mathit{apply\ the\ function\ to\ element}\ d \\
\quad \textbf{end} \\
\textbf{invariant}\ \forall\, d : D, r, r_1 : R \mid (\mathit{select}(d) = r \wedge \mathit{select}(d) = r_1) \bullet (r = r_1) \\
\qquad\quad -- \mathit{ensure\ that\ set\ is\ a\ function} \\
\textbf{end}
\end{array}
$$

3.1 Translating State Schemas

A Z state schema describes data components that are to be used in a system. Here is an example for translating the schema *BirthdayBook* introduced in Section 2. The state schema can be translated into a textual BON class that contains translated data attributes and possibly a class invariant. Schema *BirthdayBook* can be translated into the following BON textual class (supposing that we have declared deferred classes *NAME* and *DATE*).

$$
\begin{array}{l}
\textbf{class}\ \mathit{BIRTHDAY\_BOOK}\ \textbf{feature}\ \{\mathit{NONE}\} \\
\quad \mathit{known} : \mathit{SET}[\mathit{NAME}] \\
\quad \mathit{birthday} : \mathit{FUNC}[\mathit{NAME}, \mathit{DATE}] \\
\textbf{invariant}\ \mathit{known} = \mathit{birthday}.\mathit{domain} \\
\textbf{end}
\end{array}
$$

The general translation from a state schema to a BON class is as follows. Let S be an arbitrary state schema.

$$
\begin{array}{|l}
\multicolumn{1}{l}{S} \\
a_1 : T_1 \\
\ldots \\
a_k : T_k \\
\hline
P \\
\hline
\end{array}
$$

(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.

$$
\begin{array}{l}
\textbf{class}\ S\ \textbf{feature} \\
\quad a_1 : T_1 \\
\quad \ldots \\
\quad a_k : T_k \\
\textbf{invariant}\ P \\
\textbf{end}
\end{array}
$$

where P in the class invariant of S is a syntactic translation of the Z predicate P into BON's predicate notation. P is a predicate on prestate only. One possible problem with translation will be in the different interpretations of conjunctive and disjunctive operators in BON. Operators in Z are not short-circuiting, whereas they are in BON. Since both interpretations will coincide at the specification level, we need not worry about this issue.

In some specifications, state schemas include others (by writing the name of a schema in the declaration part of the including schema). Such hierarchies can be flattened, by substituting the body of the included schema for all occurrences of its name. We can translate schemas that use inclusion by first flattening the hierarchies.

3.2 Translating Operation Schemas

An operation schema represents some operation that the system can perform. Typically, an operation schema *includes* a state schema, and modifies the system state in some way. Here is an example of translating the operation schema *AddBirthday*, introduced in Section 2. Translating an operation schema to BON requires mapping the schema to a command or query of a class. The class is the result of translating a state schema that is affected by the operation. So, in the example, state schema *BirthdayBook* is affected by *AddBirthday*, thus operation schema *AddBirthday* can be translated into a command of class *BIRTHDAY_BOOK*.

$$
\begin{array}{l}
\mathit{AddBirthday}(\mathit{name} : \mathit{NAME}, \mathit{date} : \mathit{DATE}) \\
\quad \textbf{require}\ \mathit{name} \notin \mathit{known} \\
\quad \textbf{ensure}\ \mathit{birthday} = \textbf{old}\ \mathit{birthday} \cup \{p : \mathit{PAIR}[\mathit{NAME}, \mathit{DATE}] \mid p.\mathit{first} = \mathit{name} \land p.\mathit{second} = \mathit{date}\} \\
\textbf{end}
\end{array}
$$

It is possible that different operation schemas can share state schemas—i.e., two operation schemas can include some of the same state schemas. When translating the schemas, we therefore have to decide to which class the translated operations should belong. We suggest using a simple syntactic algorithm, based on the textual occurrence of a state schema in an operation schema, to help make the decision. This algorithm would form the basis of a semi-automated tool that performs the mapping from Z to BON. The algorithm parses Z schemas, and if two operation schemas share 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. It can be left to the translator to decide the name of this class; a semi-automated tool might simply choose the name of the first included state schema. This simple "collection" algorithm may lose some of the structure of the Z specification (e.g., the inclusion relationships). It may also give inappropriate results. Therefore, it may ideally be used to give a first-pass approximation to a class design, which can be further refined by developers.

Use of this algorithm gives translators and specifiers some very useful information: it can suggests an object-oriented design for a system. And if it is desired to produce a reusable, extendible system, then it may have been best to use such an object-oriented design from the start of development.

The general translation from a Z operation schema into a BON feature is as follows. Let *Op* be a generic schema as follows, and let *S* be a state schema with variables *w*. Let *T* be a state schema that operation *Op* can use, but cannot change. This is expressed using the Ξ convention.

$$\begin{array}{|l}
\multicolumn{1}{l}{Op} \\
\hline
\Delta S \\
\Xi T \\
inputs? : I \\
outputs! : O \\
\hline
P \\
\hline
\end{array}$$

(Thus, *Op* can only change variables *w* from *S*, but can use any variables in *T*. Variables in *T* are implicitly constrained, by the Ξ convention, to not change.) The operation also has inputs *inputs*? and produces *outputs*!. In translating the operation to BON, we assume that we can translate the state schemas *S* and *T* to BON, and that the types *I* and *O* can be constructed and expressed in BON as well. 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, say, class *S* that takes the following form.

$$\begin{array}{l}
Op(inputs : I) : O \\
\quad \textbf{require}\ \exists\, w', outputs! \bullet P \\
\quad \textbf{ensure}\ P[\textbf{old}\ w/w][w/w'][Result/outputs!] \\
\textbf{end}
\end{array}$$

The **require** clause is the precondition of the operation schema *Op*. Existential quantification hides the postcondition in *P*, revealing only those terms that constrain the precondition in *P*. The **ensure** clause is the predicate part of the operation schema, but with Z's primed-unprimed notation rewritten using BON's **old** notation. In the **ensure** clause, we use the Z substitution notation: $P[a/b]$ means "substitute *a* for *b* in *P*". Substitution is left-associative. Therefore, in the **ensure** clause, **old** *w* is first substituted for *w* in *P*, and then *w* is substituted for w'.

If either *inputs*? or *outputs*! in the Z schema are omitted, then they are omitted from the BON translation. If *outputs*! is missing, then *Op* is a command, and the last substitution involving *Result* can be omitted. Note that a Z operation schema can include a state schema by Δ convention, and can also have operation outputs (specified using the ! annotation). This corresponds to an operation that both changes state and returns a result — i.e., an operation that is both procedure and function. Since BON and Eiffel do not allow features with side-effects, we cannot directly translate such operation schemas into BON. Instead, we could remove the operation result, and make the result an attribute of the class to which the translated operation belongs.

4 Example: The Birthday Book

The *Birthday Book* example is a well-known and standard problem for explaining the use of Z and other formal methods. A detailed Z specification and development of the problem is in [10]. The problem requires specifying a system that records birthdays and is able to issue a reminder when the appropriate day arrives. The traditional Z development of the system begins by introducing *basic types* to stand for the names and birth dates.

$$[\, NAME, DATE \,]$$

(These could correspond to basic types in BON, or to deferred classes.) The advantage of basic types is that it lets the specifier name sets without saying what kind of things they contain.

The first part of the system specification is its state space, described as a state schema *BirthdayBook*, which we repeat here for reference.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{BirthdayBook}} \\ \hline
\mathit{known} : \mathbb{P}\, \mathit{NAME} \\
\mathit{birthday} : \mathit{NAME} \rightarrow \mathit{DATE} \\ \hline
\mathit{known} = \mathrm{dom}\, \mathit{birthday} \\ \hline
\end{array}$$

The schema predicate, $known = \mathrm{dom}\, birthday$, states that the set *known* is the same as the domain of the function *birthday*—the set of names to which the function can be validly applied.

Operations for the system can now be specified. We have already seen the operation schema for adding a new birthday. The next system operation is that to find the birthday of a person known to be in the system. This is described as operation *FindBirthday*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{FindBirthday}} \\ \hline
\Xi \mathit{BirthdayBook} \\
\mathit{name}? : \mathit{NAME} \\
\mathit{date}! : \mathit{DATE} \\ \hline
\mathit{name}? \in \mathit{known} \\
\mathit{date}! = \mathit{birthday}(\mathit{name}?) \\ \hline
\end{array}$$

The declaration Ξ*BirthdayBook* indicates that the operation does not change the state (therefore, it corresponds to a query). Including Ξ*BirthdayBook* above the schema line is equivalent to including Δ*BirthdayBook* above the line, and the equation

$$known' = known \wedge birthday' = birthday$$

below it. The *FindBirthday* operation takes a name $name?$ as input, and yields the corresponding birthday (in the range of function *birthday*) as a result.

The final system operation is that to find which people have birthdays on a given date. The operation has one input, $today?$, and one output $cards!$, which is a set of names of people to whom we want to send birthday cards. Note that there may be zero, one, or many people with the same birthday.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Remind}} \\ \hline
\Xi \mathit{BirthdayBook} \\
\mathit{today}? : \mathit{DATE} \\
\mathit{cards}! : \mathbb{P}\, \mathit{NAME} \\ \hline
\mathit{cards}! = \{ n : \mathit{known} \mid \mathit{birthday}(n) = \mathit{today}? \} \\ \hline
\end{array}$$

To finish the specification, the initial state of the system must be specified. This is written using an operation schema *InitBirthdayBook*. A consequence of the operation is that $birthday'$ is empty, too.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{InitBirthdayBook}} \\ \hline
\Delta \mathit{BirthdayBook}' \\ \hline
\mathit{known}' = \varnothing \\ \hline
\end{array}$$

We do not consider steps 5 and 6 of the Z established strategy, due to space constraints. However, the translations that we have described so far apply without change after having applied these steps to a Z specification.

4.1 Mapping to BON

A transition from Z to the assertion language of BON can now occur. The system consists of one state schema, *BirthdayBook*, and a number of operation schemas. We will therefore translate the specification into a single class, *BIRTHDAY_BOOK*. The class, with only its state attributes, was shown in Section 3.1, but we repeat it here for completeness.

$$
\begin{array}{l}
\textbf{class } \mathit{BIRTHDAY\_BOOK} \textbf{ feature } \{\mathit{NONE}\} \\
\quad \mathit{known} : \mathit{SET}[\mathit{NAME}] \\
\quad \mathit{birthday} : \mathit{FUNC}[\mathit{NAME}, \mathit{DATE}] \\
\textbf{invariant } \mathit{known} = \mathit{birthday}.\mathit{domain} \\
\textbf{end}
\end{array}
$$

Translations of Z operation schemas can now occur. The operation schemas will be translated to methods of class *BIRTHDAY_BOOK*. Operation schema *AddBirthday* was translated in Section 3.2. The *FindBirthday* operation schema is translated to the following method of class *BIRTHDAY_BOOK*.

$$
\begin{array}{l}
\mathit{FindBirthday}(\mathit{name} : \mathit{NAME}) : \mathit{DATE} \\
\quad \textbf{require } \mathit{name} \in \mathit{known} \\
\quad \textbf{ensure } \mathit{Result} = \mathit{birthday}.\mathit{select}(\mathit{name}) \\
\textbf{end}
\end{array}
$$

Finally, the *Remind* method is as follows.

$$
\begin{array}{l}
\mathit{Remind}(\mathit{today} : \mathit{DATE}) : \mathit{SET}[\mathit{NAME}] \\
\quad \textbf{ensure } \mathit{Result} = \{n : \mathit{NAME} \mid \mathit{birthday}.\mathit{select}(n) = \mathit{today}\} \\
\textbf{end}
\end{array}
$$

To translate the *InitBirthdayBook* operation schema, we update class *BIRTHDAY_BOOK* to include a **creation** method of the same name.

$$
\begin{array}{l}
\textbf{class } \mathit{BIRTHDAY\_BOOK} \textbf{ creation} \\
\quad \mathit{InitBirthdayBook} \\
\quad \textbf{feature} \\
\quad\quad \mathit{InitBirthdayBook} \textbf{ ensure } \mathit{known}.\mathit{empty} \textbf{ end} \\
\quad\quad -- \mathit{other\ methods\ as\ written\ above} \\
\textbf{end}
\end{array}
$$

BIRTHDAY_BOOK is now a type, and (for free) we can create as many instances of the *BIRTHDAY_BOOK* as we need. By comparison, to create multiple *BirthdayBook*s in Z, we would need to strengthen the specification to include a set of *BirthdayBook*s and new operation schemas to manipulate the strengthened state.

It is also worthwhile to point out that in the translation, the resulting BON specifications are no larger or more complex than the initial Z specifications. In fact, the BON specifications are, to us, more readable and understandable, because they use a more consistent syntax and because they use fewer mathematical notations and more object-oriented idioms.

4.2 Implementation

The BON specification that we have given is very close to an implementation (*AddBirthday* can be implemented using the *put* method of class *SET*, *Remind* can be implemented with a loop, and *FindBirthday* is already implemented). For the purposes of illustrating further refinement, we show how to transform the specification into an equivalent one that uses arrays.

We can represent the birthday book by two arrays (of a constant *max* size), *names* and *dates*, and a counter *count*, specifying how much of the arrays is in use. We refine class *BIRTHDAY_BOOK* so that it includes these new attributes. Each operation *FindBirthday*, *Remind*, and *AddBirthday* is then refined to use the new data structures. In order to prove that these refinements are consistent with the initial specification in terms of sets, we have to produce an *abstraction relation* [7] that relates the new data structures with the original data structures. An abstraction relation, written in BON's assertion language, is

$$\begin{aligned}&known = \{i : 1..count \bullet names.item(i)\} \wedge \\ &\forall\, i : 1..count \mid \exists\, j \in birthday \mid j.first = names.item(i) \wedge j.second = dates.item(i)\end{aligned}$$

The abstraction relation states that the set *known* corresponds to all valid entries in the array *names*, while every entry in the array *dates* has a corresponding entry in *birthday*. Applying the abstraction relation to the class *BIRTHDAY_BOOK* would produce the following additions to *BIRTHDAY_BOOK*. First are the attributes and refined creation command (we are omitting the the technical details and simplifications that need to be carried out, but they are straightforward and follow the techniques suggested in [7]).

$$\begin{array}{l}\textbf{class}\ \mathit{BIRTHDAY\_BOOK}\ \textbf{creation}\\ \quad \mathit{InitBirthdayBook}\\ \quad \textbf{feature}\ \{\mathit{NONE}\}\\ \qquad \mathit{names} : \mathit{ARRAY}[\mathit{NAME}]\\ \qquad \mathit{dates} : \mathit{ARRAY}[\mathit{DATE}]\\ \qquad \mathit{count} : \mathit{INTEGER}\\ \textbf{end}\end{array}$$

Each method must also be transformed to use the new representation. The results of this transformation are as follows (we omit the details and simplifications, which are straightforward). Method *InitBirthdayBook* is refined to the following specification.

$$\mathit{InitBirthdayBook}\ \textbf{ensure}\ \mathit{count} = 0\ \textbf{end}$$

The **ensure** and **require** clauses of *AddBirthday* are transformed as follows.

$$\begin{array}{l}\mathit{AddBirthday}(\mathit{name} : \mathit{NAME}, \mathit{date} : \mathit{DATE})\\ \quad \textbf{require}\ \forall\, i : 1..count \bullet name \neq names.item(i) \wedge count < max\\ \quad \textbf{ensure}\ count = \textbf{old}\ count + 1 \wedge names.item(count) = name \wedge dates.item(count) = date \wedge\\ \qquad\qquad \forall\, i : 1..\textbf{old}\ count \bullet names.item(i) = \textbf{old}\ names.item(i) \wedge dates.item(i) = \textbf{old}\ dates.item(i)\\ \textbf{end}\end{array}$$

To add a new entry that is not already in the birthday book, we increment *count* and fill in the name and date in the arrays, without changing the rest of the arrays. The second operation, *FindBirthday*, is transformed to the following.

$$\begin{array}{l}\mathit{FindBirthday}(\mathit{name} : \mathit{NAME}) : \mathit{DATE}\\ \quad \textbf{require}\ \exists\, i : 1..count \bullet name = names.item(i)\\ \quad \textbf{ensure}\ \exists\, i : 1..count \bullet name = names.item(i) \wedge Result = dates.item(i)\\ \textbf{end}\end{array}$$

This specification says that there is an index i at which the *names* array contains the input, *name*, and for which the result is the corresponding entry of the *dates* array. Finally, we transform the *Remind* operation.

$$
\begin{array}{l}
\mathit{Remind}(\mathit{today} : \mathit{DATE}) : \mathit{SET}[\mathit{NAME}] \\
\quad \textbf{ensure } \mathit{Result} = \{ j : 1..\mathit{count} \mid \mathit{dates}.\mathit{item}(j) = \mathit{today} \bullet \mathit{names}.\mathit{item}(j) \} \\
\textbf{end}
\end{array}
$$

An advantage of mapping Z into BON/Eiffel is that Eiffel supports sets. In the standard refinement of the Z specification for operation *Remind* [10], the *SET* result is implemented as an array and a second counter. This makes the development longer than we need to have in a mapping to Eiffel.

Operations can now be implemented in Eiffel. We assume that there are *is_equal* queries for *NAME* and *DATE*, and a constant array size *max*. We strengthen method *InitBirthdayBook* so that it includes Eiffel code to attach *ARRAY* objects to *names* and *dates*. We omit implemented **ensure** clauses due to lack of space, but we include translations of **require** clauses.

```
InitBirthdayBook is
  do count := 0   !!names.make(1, max)   !!dates.make(1, max) end
AddBirthday(name : NAME;  date : DATE) is
  require not names.full and not names.has(name)
  do
    count := count + 1
    names.put(name, count)
    dates.put(date, count)
  end
FindBirthday(name : NAME) : DATE is
  require names.has(name)
  local i : INTEGER
  do
    from i := 1
    until names.item(i).is_equal(name)
    loop i := i + 1 end
    Result := dates.item(i)
  end
Remind(today : DATE) : SET[NAME] is
  local i : INTEGER
  do
    from i := 1
    until i > count
    loop
      if dates.item(i).is_equal(today) then Result.put(names.item(i)) end
      i := i + 1
    end
  end
```

5 Discussion and Conclusions

We have outlined how Z users can make a transition to object-oriented specifications and executable implementations via a mapping into BON. This transition integrates the Z established strategy of [1] with a seamless development method, and can be used to suggest an object-oriented design for Z specifications. While we have not shown all the technicalities associated with a transition (due to space constraints), we have provided sufficient detail so that developers can make a move from Z to BON and thereupon to Eiffel. The technique suggests a strategy for producing an object-oriented design from a (non-object oriented) Z specification. This provides a basis for semi-automatable tool that implements a mapping from Z to BON (and thereafter to Eiffel). An implementation of such a tool would be able to make use of the existing Eiffel environment, as well as the existing and substantial Eiffel libraries.

We have shown that in the mapping, the specifications that result from translating Z to BON are no more complex to read or use than the initial Z specifications. Hopefully, we have provided some evidence to show that in the situations where an object-oriented design or implementation is desired, Z can be avoided, and specifications and designs can be done using BON from the start, without loss of expressiveness or of the ability to write concise specifications.

Use of the approach has several other advantages. A transition from Z to BON can eliminate some of the weaknesses associated with development in Z, such as the lack of an implementation of Z constructs. The transition also can give Z developers the ability to use object-oriented technologies, which are important in developing reusable, extendible quality software.

On the other hand, the approach has some limitations. Translation of the schema calculus—and in particular, schema inclusion—may require unfolding of operations, or unfolding of state schemas, before translation. This could result in large translations and a complicated translation process, though this is not apparent in the example that we have shown here. As well, in translating we must be able to express the Z toolkit (and, in particular, Z types) in BON's assertion language. Therefore, there may be some preprocessing work that has to be done before the translation from Z specifications into BON can be carried out.

The integration of the Z established strategy with the seamless development method bypasses a standard step: making Z operations total, by applying the Z schema calculus. We do not make the operations total, because this is defensive programming that is obviated by design-by-contract, which is inherent in BON. Therefore, Z developers who are used to making their operations total may find it difficult to transition to BON. On the other hand, experience with Z in practice has often showed that specifiers are confused or uncomfortable with using the Z schema calculus. So by transitioning to BON, we could make Z more attractive to some developers.

The aim of any method integration is to acquire some of the strengths and eliminate some of the weaknesses of the methods involved [9]. In combining Z with BON, we have removed several of the noted limitations with Z, while acquiring the strengths of a seamless development method. Future work will explore the transition in larger case studies, as well as an implementation of the mapping, based on the suggested algorithm.

References

1. R. Barden, S. Stepney, and D. Cooper. *Z in Practice*, Prentice-Hall, 1994.
2. R. Duke, G. Rose, and G. Smith, Object-Z: A Specification Language advocated for the description of standards, *Computer Standards and Interfaces* **17**(5) (1995).
3. J. Gibbons. Formal Methods: Why Should I Care? The development of the T800 transputer floating-point unit. In *Proc. 13th New Zealand Computer Society Conference,* 1993.
4. A. Hall, Using Formal Methods to Develop an ATC Information System, *IEEE Software* **13**(2), 1996.
5. M. Hinchey and J. Bowen. *Applications of Formal Methods,* Prentice-Hall, 1995.
6. I. Houston and S. King. CICS Project report: experience and results for the use of Z in IBM. In *Proc. 6th Annual Z Users Meeting,* Springer-Verlag, 1992.
7. B. Meyer. *Object-oriented Software Construction*, Second Edition, Prentice-Hall, 1997.
8. C.C. Morgan. *Programming from Specifications,* Prentice-Hall, Second Edition, 1994.
9. R.F. Paige. A Meta-Method for Formal Method Integration. In *Proc. Formal Methods Europe 1997,* LNCS 1313, Springer-Verlag, 1997.
10. J.M. Spivey. *The Z Notation: A Reference Manual*, Prentice-Hall, Second Edition, 1992.
11. K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Architecture*, Prentice-Hall, 1995.
12. J.B. Wordsworth. *Software Development with Z,* Addison-Wesley, 1992.