JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 3, March–April 2004

Integrating BON and Object-Z

Richard Paige, Department of Computer Science, University of York, U.K.
Phillip J. Brooke, School of Computing, Communications, and Electronics, University of Plymouth, U.K.

A significant limitation with object-oriented formal specification languages, such as Object-Z, is that they lack development and management processes, which can be used to guide the production of reliable, robust object-oriented systems. An integration of an object-oriented methodology, BON, and Object-Z is presented in order to add an industrially validated development process to Object-Z. An extensible CASE tool for BON is also described that supports the integration with an Object-Z code generation engine.

1 INTRODUCTION AND MOTIVATION

Object-Z [16] is a formal specification language for object-oriented (OO) systems. Based on Z [19], it provides a rich collection of specification idioms for modelling OO systems in a precise, unambiguous notation. Using the language's formal semantics, *refinement rules* have been defined that can be used for rigorously transforming Object-Z specifications into programs, while at the same time generating a proof that said programs satisfy the original specification [17].

There are significant limitations with languages such as Object-Z: their tools are weak, providing little support for typical systems development tasks such as testing, debugging, and verification and validation; there is limited support for the full systems development process, showing how to take customer requirements, business rules, and architectural constraints through to an executable system; and the languages themselves are often very different from those that are familiar to systems developers, such as statecharts, use case diagrams, and UML in general. It is the second of these points (and, to a lesser extent, the first) that we aim to address in this paper: a development process is essential if a language like Object-Z is to be considered a full-fledged methodology, to be applicable to solving industrial systems engineering problems.

This paper shows how to integrate Object-Z with an existing OO methodology, BON [22], effectively providing a risk-driven development process for Object-Z. It shows how BON models can be used to produce Object-Z specifications, and how Object-Z tools might be applied in the development process. It also demonstrates tool support for the integrated methodology, via a CASE tool for BON that allows for automatic generation of Object-Z specifications.

BON and Object-Z each have the following properties:

- Object-Z and the BON modelling language provide the means for formally documenting the properties of *classes*, via contracts. In this sense, the languages are *compatible*; thus, it is relatively straightforward to define a structural translation from BON to Object-Z, though technical details remain in terms of a semantic translation. Effectively, integrating BON with Object-Z is a *lightweight* means to providing a development process for Object-Z.

- Each approach has limitations, both at the language level and at the methodological level. We discuss this more in the sequel.

- Object-Z possesses integrations with complementary formal specification languages for modelling real-time and concurrent systems, e.g., Timed CSP [15]. BON provides a number of industrially applicable graphical modelling notations, and its tools support these notations as well as links to programming languages, such as Java and Eiffel. In this sense Object-Z and BON are complementary [12].

In this paper, we outline a tool-supported integration of BON and Object-Z, in order to address the limitations of each approach. In the terminology of Stirewalt and Dillon [20], the integration is both *artifactual* (it is done to exploit tools and elements of the individual techniques) and *effectual* (it is done so as to carry out tasks that could not be carried out with one or both of the separate techniques). We briefly outline preliminary tool support for the integration via a CASE tool for BON that automatically generates Object-Z specifications (in LaTeX format). The tool can thus be used to support a methodology that applies graphical object-oriented modelling and use of Object-Z.

2 BACKGROUND

BON

BON is a method possessing a recommended process as well as a graphical language for specifying OO systems. The process is iterative, risk-driven, and architecture-centric. The language provides mechanisms for specifying classes and objects, their relationships, and assertions (written in first-order predicate logic) for specifying the behaviour of routines and invariants of classes. The fundamental construct in BON is the class. Fig. 1 contains an example of a BON diagram for the interface of a class *CITIZEN*. A class has a name, an optional class invariant, and zero or more features. A feature may be an *attribute* (e.g., *name*), a *query* (e.g., *single*) – which returns a value and does not change the system state – or a *command* (e.g., *divorce*), which changes system state but returns nothing. In [22], attributes are treated as

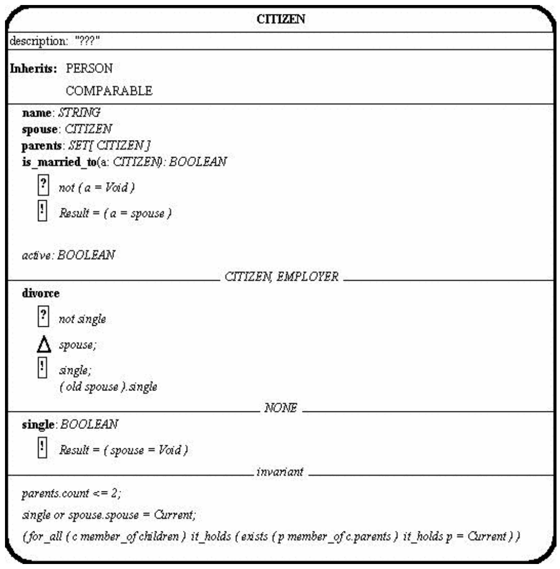


Figure 1: Class *CITIZEN*

parameterless queries without assertions; we distinguish attributes to make it easier to generate code (particularly Object-Z).

Preconditions and postconditions of features are indicated using **?** and **!** in boxes, respectively. We have introduced a notion of a frame to BON class interfaces. The Δ clause, adopted from Z, specifies a bunch of attributes that may be changed by the feature. Attributes are, by default, of reference type (except primitives such as *INTEGER*). Aggregation relationships can be used to introduce value types. The class invariant specifies properties that must be true before and after any client-side call to a feature of the class.

BON class diagrams consist of one or more classes organized in *clusters* (drawn as dashed rounded rectangles that may include classes and other clusters). Classes and clusters interact via two general kinds of relationships. The relationships are drawn in Fig. 2, which provides an example of a BON class diagram.

- **Inheritance:** Inheritance defines a subtyping relationship between a child and parents. It is drawn from class *CITIZEN* to class *PERSON* in Fig. 2. The behaviour of child classes must conform to the behavioural specifications of their parents. To this end, BON permits only *covariant* adaptation of feature signatures, precondition weakening, and postcondition strengthening.

- **Client-supplier:** there are two client-supplier relationships, association (drawn between *CITIZEN* and *EMPLOYER*) and aggregation (drawn between *CITIZEN* and *CITIZENSHIP*). Both relationships are directed from a *client* to a *supplier.*

BON also provides notation for dynamic diagrams, showing the messages passed between objects, in a manner akin to UML's collaboration diagram [2]. Examples

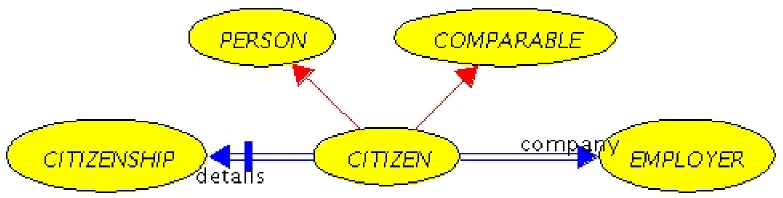


Figure 2: BON class diagram notation

can be found in [14]. These diagrams, and others, are supported by the BON-CASE tool [14].

The BON process

The BON development process is iterative, risk-driven, and idealised; compatibility with the BON process is defined in terms of producing a required set of document deliverables, including class diagrams, dynamic diagrams, scenarios and charts, etc. Each task in the process has a set of input sources, produces a set of deliverables, and is controlled by acceptance criteria, which take into account the risk of proceeding. Each task may be iterated several times with feedback. The BON process is not use case driven, but it is architecture-centric; use cases may be applied in the process's early stages. The process also emphasises using contracts to capture the behaviour of modelling abstractions. The process is sketched in Fig. 3.

Task	**Description**
1	Delineate system borderline.
2	List candidate classes.
3	Select classes and group into clusters.
4	Define classes and their features using class diagrams.
5	Sketch system behaviours using dynamic diagrams.
6	Define public features and contracts.
7	Refine system.
8	Factor out common behaviour.
9	Complete and review system.

Figure 3: The BON idealised process

Object-Z

Object-Z is a formal specification language for OO systems, based on Z [19]. It provides a full range of OO specification constructs, including classes, attributes, methods, contracts, polymorphism, information hiding, containment (aggregation), parameterized classes, class union, and inheritance. It has a formal semantics and tool support (particularly, via LaTeX style files, the Wizard type checker [5], the graphical editor Moby/OZ [8], and ZML [3], with ongoing work on embeddings in Isabelle/HOL [17]). It has been integrated with several complementary formal specification languages, such as Timed CSP [15]. It has also been used to formalize parts of UML, and parts of the UML metamodel [6].

Fig. 4 provides an example of a class specified in Object-Z; it demonstrates most of the fundamental Object-Z notation, and is taken from [4]. An Object-Z class is introduced via a class schema. A visibility list specifies those features that may be accessed by clients. Axiomatic definitions specify local functions and constants. State attributes and state invariants follow. Then, an initial state may be specified; this is essential for grounding inductive arguments. Finally, operation schemas defining methods of a class may appear. The syntax for predicates and expressions is derived from standard Z, with some syntactic sugar and extra constructs that are particularly suited for object-oriented specification.

A systematic approach to method integration

A systematic approach to integrating formal and semiformal methods was presented in [9, 10]. The approach emphasises integrating *modelling languages* while providing assistance in generalizing and linking processes. The usefulness of the approach and the methods it produces has been validated on a number of case studies, e.g., [11, 13]. The method is based on the construction of heterogeneous languages [9], and on defining relationships between tasks and deliverables in processes. We start with a brief overview of heterogeneous languages and their construction, and then discuss the steps of the method itself, which will indicate how relationships between processes are to be defined.

Heterogeneous languages and bases

Modelling languages play a critical role in software development methods. Modelling languages (which consist of a notation as well as a metamodel) play a key role in how we integrate methods: we combine languages as the first step. A *heterogeneous language* is composed from several different languages and is used to write heterogeneous specifications, which are composed from parts written in two or more different modelling languages. A formal semantics for a heterogeneous specification can be given by formally defining the meaning of the composition of partial specifications in some language. In general, we may want to build a heterogeneous language from several formal or informal languages. In this case, we can construct a *heterogeneous basis* [9], a set of languages and *translations* between languages, which can be used to give a formal semantics to heterogeneous specifications via translation to a homogeneous specification. This also allows semi-formal languages, such as data flow diagrams or variants of object modelling languages, to be given an appropriate semantics depending on the context in which they are used. This is consistent with

$\mathit{CreditCard}$

$\upharpoonright(\mathit{limit}, \mathit{balance}, \mathit{INIT}, \mathit{withdraw}, \mathit{deposit}, \mathit{withdrawAvail})$

$\mathit{limit} : \mathbb{N}$

$\mathit{limit} \in \{1000, 2000, 5000\}$

$\mathit{balance} : \mathbb{Z}$

$\mathit{balance} + \mathit{limit} \geq 0$

INIT

$\mathit{balance} = 0$

$\mathit{Withdraw}$

$\Delta(\mathit{balance})$
$\mathit{amount?} : \mathbb{N}$

$\mathit{amount?} \leq \mathit{balance} + \mathit{limit}$
$\mathit{balance}' = \mathit{balance} - \mathit{amount?}$

$\mathit{Deposit}$

$\Delta(\mathit{balance})$
$\mathit{amount?} : \mathbb{N}$

$\mathit{balance}' = \mathit{balance} + \mathit{amount?}$

$\mathit{WithdrawAvail}$

$\Delta(\mathit{balance})$
$\mathit{amount!} : \mathbb{N}$

$\mathit{amount!} = \mathit{balance} + \mathit{limit}$
$\mathit{balance}' = -\mathit{limit}$

Figure 4: The Object-Z class *CreditCard*

the work of Baresi and Pezzé [1] on formalising families of languages, such as those available in Structured Analysis.

Fig. 5 depicts an example of a heterogeneous basis; it is a substantial extension of one presented in [9]. Translations between many of the languages are documented in [9, 10, 11]. A mapping from Z to BON is given in [13].

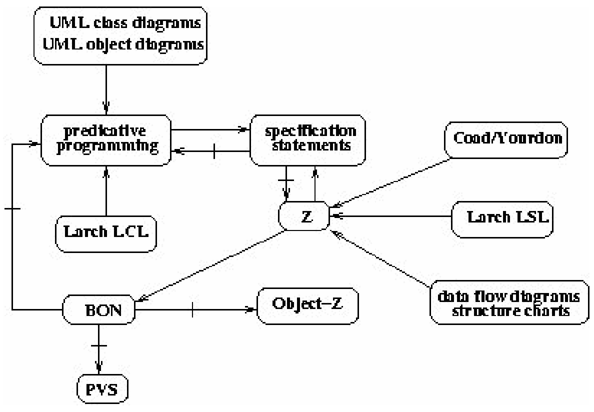


Figure 5: Heterogeneous basis

In Fig. 5, the arrows represent translations that have been defined between the languages. The arrow ⇸ between languages represents a partial translation: there may exist constructs in the source language that are inexpressible – and thus, are not translated – in the target language; or, a translation of every construct in the metamodel of the source language has not been presented or is currently unavailable. Translations are given in terms of the metamodel of a source language: each construct that appears in the metamodel of the source language is mapped to a construct or set of constructs in the metamodel of the target language. In this manner, translations can be defined for text-based languages (they are expressed in terms of abstract syntax trees), and for visual languages as well.

The approach

We recap the steps of the approach here for the sake of completeness; full details are in [9].

1. **Ensure complementarity of the methods.** This step provides motivation for the integration. Generally, methods may be complementary in terms of their modelling languages, their processes, and their supplementary tools, but other pragmatic rationales for integration may be provided as well [12].

2. **Select a base method and choose invasive techniques.** A *base* method provides a process that is to be supported and complemented by other (invasive) methods. Selecting a base method is aimed at assisting integrators in

determining the roles that the separate methods will play in the integrated approach. The processes of *invasive* methods augment, are embedded in, or are interleaved with that of the base method. Overlaps between processes must be reconciled. If a modelling language is being integrated with a methodology, then process incompatibilities are a deprecated issue, and effectively all that must be shown is how to use the new modelling language with the existing process.

3. **Construct or extend a heterogeneous basis.** This is accomplished by defining translations, or by adding languages from the base and invasive methods to an existing heterogeneous basis. At this point, a *basis language* can be selected. This language is chosen from the heterogeneous basis and is used to provide a formal semantics to heterogeneous specifications.

4. **Define how the individual processes cooperate.** It is informally described how the processes of the methods are to work together in the new method. Process cooperation can be specified using UML activity diagrams. Two forms of cooperation are particularly common.

   - **Generalisation.** The process of the base method is generalized to use heterogeneous notations constructed from those of the base and invasive methods. Effectively, notations are added to an existing method, and its process is generalized to use the new notations.
   - **Interleaving of processes.** Interleaving relationships between the process of the base method and the processes of the invasive methods are defined. A selection of different process interleavings are documented in [9, 13]; the latter involves a link between Z and BON.

5. **Guidance to the user.** Hints and examples on how the integrated method can be used is provided.

3 INTEGRATING THE BON METHODOLOGY AND OBJECT-Z

We now describe the integration of the BON methodology and Object-Z, following the approach presented in the previous section. The integration will generalise the BON process by adding Object-Z to it, by including use of Object-Z tools within the process, and by integrating feedback from Object-Z tools and reasoning techniques into the process. To carry this out, in part, we will show how to translate BON to Object-Z.

Ensuring complementarity and compatibility

For any integration of languages or methods, it is critical to justify the complementary nature of the techniques, so as to justify the usefulness of the integration. BON and Object-Z are complementary techniques in the following ways:

- BON provides graphical languages for modelling, whereas Object-Z is text-based; these languages are supported by a CASE tool, which also supports generation of a number of programming languages. The integration allows developers to use all the graphical notations of the CASE tool, such as class, collaboration, and use case diagrams, with Object-Z. The value of this should not be underestimated: if Object-Z specifications can be generated by developers in a reasonably familiar way (e.g., in the same way as program code) then adoption of Object-Z can be made easier.

- Object-Z currently provides richer tool support for analysis than does BON. No full-fledged type checker currently exists for BON, and only a partial PVS embedding of selected BON constructs exists for reasoning support. Integrating the techniques will allow developers to use the Wizard typechecker, as well as LaTeX. With some additional work, it will be possible to use the Isabelle/HOL embedding of Object-Z [17] and ZML [3].

- Object-Z provides a different set of constructs for modelling object-oriented systems than BON, particularly: class unions, secondary variables, a richer notion of inheritance, and schema operators. By contrast, BON provides a notion of attachment and reattachment for reference attributes, constrained genericity, and a richer notion of information hiding. The modelling languages are thus complementary.

- BON is a methodology, whereas Object-Z is a modelling language. Integrating BON and Object-Z adds development process support to Object-Z.

Selecting a base method and invasive technique

The base method in the integration will be BON, in part because it provides a process that spans requirements analysis, through detailed design, generalization, and validation. The BON process will be generalized with the use of Object-Z and its tools. The invasive technique in this integration is Object-Z. The BON process will be extended to make use of Object-Z specifications, and thereafter Object-Z tools, in the generalization of the BON process.

Extending a heterogeneous basis

The next step is to combine the languages of the techniques of interest. We will thus add Object-Z to the heterogeneous basis presented in Fig. 5. The extension will occur by rigorously defining a translation from BON to Object-Z. The details of the translation are quite long; we thus provide an overview here. The translation will be presented in terms of the metamodels of BON and Object-Z; we show how each meta-concept in BON can be mapped to one or more meta-concepts in Object-Z.

BON and Object-Z are both based on four key OO concepts, namely: *classes, features* of classes, *properties* of features and classes, and *relationships* between classes. We describe the translation in terms of these four basic concepts.

Translating classes

All BON classes (except primitives, two specific generic classes, and constrained generic classes) are translated into Object-Z classes. Both BON and Object-Z support generic classes with an arbitrary number of parameters; these are also translated directly and recursively. A BON class may be annotated with stereotypes, e.g., deferred, effective, reused, interfaced, persistent, and root. These stereotypes are dropped in translation to Object-Z: the deferred stereotype is captured by the Object-Z concept of a class; the effective, reused, external, and interfaced stereotypes in BON are given only to aid the reader – these can be translated as comments. The persistent stereotype indicates that instances of the BON class persist between executions of the system. We translate this for now by adding a comment to the resultant Object-Z class indicating that instances should persist. We envision future extensions of the translation that make use of the interfaced stereotype, in particular, for generating Mobile Object-Z specifications [21].

BON supports constrained generic classes; these classes are parameterized, but the parameters are syntactically constrained to conform to specific interfaces. For example, a generic class with parameter G might be constrained so that G conforms to the *COMPARABLE* interface. Such constructs cannot be directly translated to Object-Z, and so in translation the interface constraints are treated as comments.

BON built-in types (*INTEGER*, *REAL*, *CHARACTER*, *STRING*, *BOOLEAN*) are mapped to their Object-Z equivalents ($\mathbb{Z}$, $\mathbb{R}$, *CHAR*, seq *CHAR*, $\mathbb{B}$). The only significant changes need to be made with BON generic sets and sequences, i.e., $SET[G]$ and $SEQUENCE[G]$. These are translated recursively to $\mathbb{P}\, G$ and $\mathrm{seq}\, G$, respectively, and their operations are translated to operators on sets and sequences.

Every BON class that is translated into Object-Z is a supertype of the Object-Z class *None*. This is so that we mimic the lattice of types of BON in Object-Z, and so that we can use *Void* references as in BON. Thus, each BON system, consisting of classes $S1, S2, \ldots, Sn$, produces the Object-Z class

```
┌─ None ──────────────────────────
│ S1
│ S2
│ ...
│ Sn
└─────────────────────────────────
```

Thereafter, *None* will be used as the type of a predeclared entity called *Void*, enabling an embedding of BON reference types in Object-Z.

Translating features

A feature in BON is either an attribute, a query, or a command. Attributes in BON are translated into Object-Z attributes included in the state schema. BON does not support secondary variables, thus all attributes are considered to be primary. Each attribute in BON has a default initial value (e.g., *INTEGER*s have default 0, *BOOLEAN*s have default *false*). Thus, each attribute also generates a default predicate that is added to the *INIT* schema in the corresponding Object-Z class. BON differs from Object-Z in that reference attributes (e.g., *spouse* in Fig. 1) have initial value *Void*. As well, any reference attribute in BON can be tested against *Void*, or equated to *Void*. *Void* references are discussed further in the sequel.

In BON, all attributes of non-primitive type are potentially polymorphic; their Object-Z translations must be potentially polymorphic as well. Thus, an attribute $a : A$ in BON is translated to $a :\downarrow \tau(A)$ in Object-Z (where $\tau(A)$ represents the translation of class A).

Queries in BON are side-effect free functions. One might expect that these should be translated to Object-Z operation schemas. However, it is commonplace in BON to use queries in the specification of contracts; in a postcondition of a query, command, or class invariant one might see a call to a query. It is not possible to use Object-Z operation schemas as calls in the predicate part of other operation schemas. Thus, we translate BON queries into Object-Z functions and specify them as axiomatic definitions. Further technical details on this are in the next section.

Commands in BON can change the state of an object, but cannot return a value. Consequently, they are translated to Object-Z operation schemas. Arguments to the BON command are annotated with ? in Object-Z, and the command's frame is translated to an Object-Z Δ list that is included in the corresponding operation schema.

BON classes may also possess *deferred* features, which are routines that are to be implemented by child classes. These may or many not possess contracts (we discuss the translation of contracts in the next subsection). Whether or not a feature is deferred in BON has no effect on its translation to Object-Z.

Each feature in BON has a list of client classes that have permission to access the feature; the list may range from any client (public), to a select list, to no clients (private). In translating this to Object-Z, we translate all non-private feature access to inclusion in the Object-Z class's projection list. Thus, only private features are excluded from this list. This results in a loss of information since features that were accessible only to selected clients are now accessible to all clients. This is not ideal, but it is the only way to translate such BON constructs to Object-Z.

Translating contracts

A BON feature may have a precondition and postcondition, and a class may have an invariant. As well, commands may possess a frame, indicating the attributes that may be changed by the command. These constructs exist in Object-Z as well. When mapping a routine's pre- and postcondition into Object-Z, two approaches are taken:

- *queries:* a BON query is translated to an Object-Z function, specified as an axiomatic definition. The precondition and postcondition of the BON query must be combined in the Object-Z specification. Since BON queries have no side-effects, and their semantics is undefined if they are called with their precondition unsatisfied, this translation is semantics-preserving. The translation has one complexity (due to the semantics of functions in Z and Object-Z), and an example will help to clarify the situation. Consider the following BON query, *imt* (is-married-to), a routine of class *CITIZEN*.

  $$\begin{array}{l} imt(a : CITIZEN) : BOOLEAN \\ \textbf{require } a \neq Void \\ \textbf{ensure } Result = (a = spouse) \end{array}$$

  The naive translation of *imt* into Object-Z would be to an axiomatic definition that takes a *CITIZEN* as an argument, returns a boolean, and has a definition that directly translates the BON postcondition into an Object-Z assertion. However, Object-Z functions only apply on their domains. Thus, the following translation must be used.

  $$\begin{array}{|l} imt : CITIZEN \rightarrow\!\!\!\!\!+ \mathbb{B} \\ \hline \forall\, a : CITIZEN \bullet \neg\ (a = Void) \Rightarrow a \in \operatorname{dom} imt \\ \forall\, a : CITIZEN \bullet a \in \operatorname{dom} imt \Rightarrow imt(a) = (a = spouse) \end{array}$$

  The first conjunct says that every a of type *CITIZEN* that satisfies the precondition $\neg\ (a = Void)$ is in the domain of *imt*. The second conjunct says that every a in the domain of *imt*, when applied to *imt*, generates the value $(a = spouse)$.

- *commands:* preconditions and postconditions of BON commands are conjoined in the resulting Object-Z operation schema. The semantics of BON commands is that if they are called with unsatisfied precondition, the behaviour of the command is unspecified. Thus, the translation is semantics-preserving.

Some minor syntactic rewriting must be carried out during the translation of BON assertions to Object-Z predicates. Much of this rewriting is straightforward (e.g., mapping BON boolean operators like **and** to $\wedge$, and BON's **old** expressions in postconditions to Object-Z's primed-unprimed expressions).

A BON class invariant can be mapped directly to an Object-Z state invariant. Fig. 6 shows the Object-Z translation of the invariant of *CITIZEN* shown in Fig. 1. This will be included in the state schema; separate lines are implicitly conjoined.

$$\begin{array}{l} \mathit{single} \vee \mathit{spouse}.\mathit{spouse} = \mathit{self} \\ \#\mathit{parents} \leq 2 \\ \forall\, c \in \mathit{children} \bullet \exists\, p \in c.\mathit{parents} \bullet p = \mathit{self} \end{array}$$

Figure 6: Object-Z translation of *CITIZEN* invariant

Translating relationships

BON possesses three kinds of class relationships: inheritance, association, and aggregation. The latter two relationships, which define *has-a* and *part-of* relations between client and supplier classes, are translated directly to Object-Z (reference) attributes and compositions. Thus, an aggregation relation from class A to class B, with label b, is translated to the Object-Z attribute $b : B_{\copyright}$. This accurately represents the semantics of BON associations and aggregations. Note that aggregations are not potentially polymorphic in BON, and this is also reflected in the Object-Z translations.

Inheritance in BON defines a subtyping relationship. It is mapped into Object-Z's class schema inclusion. When a BON feature is inherited, it can be renamed (via BON's **rename** clause) and redefined (i.e., its behaviour can be changed). Object-Z supports renaming of features, via substitutions on the included schema. Thus, a BON rename clause of the form

$$\mathit{CLASS}1 \;\; \textbf{rename}\ f\ \textbf{as}\ g\ \textbf{end}$$

(which means, inherit class $\mathit{CLASS}1$ and rename its feature f to g in the inheriting class) is translated to $\mathit{CLASS}1[f/g]$ in Object-Z. Renamings of multiple features from one class is possible in BON, and these are translated directly into Object-Z by the obvious generalization of the above approach. The only complication with the above is that class interfaces must be preserved, under renaming, in Object-Z; this is not the case in BON. Thus, if a parent class has a feature f that is renamed to g in a child class, then the child class must also possess a feature named f, so as to maintain substitutability. Thus, when translating a renaming as above, an additional definition is added to the Object-Z specification of the form $f \mathrel{\widehat=} g$, which redeclares f in the child class, and defines it as a synonym for g.

For most redefined features in BON, nothing special needs to be done for translation. Feature signatures in BON can be covariantly redefined; this is not permitted in Object-Z so it must be checked that feature signatures are not changed in BON in order to translate.

Complications and expressiveness

The potential for problems arises in translating the following BON constructs into Object-Z.

1. *Constrained genericity*, *information hiding*, and *covariant feature redefinition,* as discussed above.

2. *The assertion language.* The BON assertion language is, informally, equivalent to Object-Z, in its support for first-order boolean operators, primitive types, and relations. BON provides one construct that is not available in Object-Z: the colon operator :, which can be used to determine the *dynamic* type of an object attached to a variable. For example, the expression

$$e : \mathit{EMPLOYEE} \wedge e.\mathit{salary} \geq 20000$$

   is *true* iff the type of the object attached to e is *EMPLOYEE*, and the employee's salary is at least 20000. The static (declared) type of e need not be *EMPLOYEE*. Object-Z does not provide such an operator; it is thus translated as a comment. We note that the colon operator is used infrequently in BON, and is often replaced by polymorphic calls.

3. *Changing export policy.* BON allows a class to change the export policy of a feature that it is inheriting, via an **export** clause. For example, a feature that was publicly accessible in the parent could become private in a child. Object-Z does not allow changes in the export policy of features that are inherited. Thus, in order to translate BON to Object-Z, a contextual check must be carried out that no change in export policy occurs in the BON specification.

4. *BON reference types.* In Section 3.3.2, we discussed the BON constant *Void*, which is the initial value for reference attributes. Such a construct is not directly supported in Object-Z. There are several strategies to embedding this concept in Object-Z. The approach we take is to introduce a new Object-Z class, *None*, which inherits from every translated BON class. *Void* is then introduced as a global constant of type *None*. An alternative approach, which is also taken in [18], would be to introduce an attribute $\mathit{Void} : \mathbb{B}$ in each class; the *INIT* schema would then contain clauses initializing each attribute. We prefer the former approach since it directly expresses the semantics of BON classes.

Example

Consider the BON class diagram shown earlier, in Fig. 2; it illustrates much of the standard BON notation, such as associations, aggregations, inheritance, and classes. The interface details of the *CITIZEN* class were shown in Fig. 1. The interface contained much of the standard BON interface notation, including attributes, queries, commands, assertions, information hiding details, and renaming of inherited features. The Object-Z specification in Fig. 7 is automatically generated by BON-CASE for *CITIZEN*, in LaTeX format.

The aggregation relationship is mapped to an Object-Z composition, the multiple inheritance relationships are mapped to class schema inclusion, and the associations are mapped to attributes. The assertion language for BON is mapped to Object-Z predicates.

BON-CASE supports the generation of code for individual classes, or for an entire class diagram. This class diagram may include clusters, in which case the code generation process recurses through the cluster structure. Thus, from the diagram in Fig. 2, Object-Z will be generated for *PERSON*, *EMPLOYER*, *CITIZENSHIP*, and *COMPARABLE*. In Fig. 2, no interface details are provided for *EMPLOYER* or *CITIZENSHIP*; thus, empty class declarations are automatically generated for these. An option is being added to the code generator to allow classes with empty interfaces to be mapped to uninterpreted types.

Define the integrated process

The next step in combining BON and Object-Z is to provide an integrated development process that describes how the techniques are to be used together. We explain how the BON process is to be *generalised* to make use of Object-Z and the feedback that Object-Z tools provide.

The BON process was described in Section 2.1.1. Task 6 of the process, defining public features and contracts, is the first place wherein assertions are added to features and classes. It is after or during this task where we generalise the process to use Object-Z. Object-Z could be applied earlier, but it would be of less use, since contracts in the BON model would not yet have been produced.

The generalised process is sketched in Fig. 8, using a UML activity diagram. A decision point has been introduced, wherein Object-Z is automatically generated and analyzed, e.g., using Wizard or an Object-Z embedding in a theorem prover or model checker. At this point, several approaches can be taken.

1. The results of analyzing the generated Object-Z may point to the need to refine and improve the BON specifications. Feedback from analysing the Object-Z is therefore used to re-factor the BON specifications.

2. The results of analysis may suggest that the BON specifications are acceptable.

CITIZEN

$\upharpoonright(name, spouse, parents, is\_married\_to, active, divorce)$

$PERSON[active/retired]$

$COMPARABLE$

$Void : None$

$single : \mathbb{B}$

$single = (spouse = Void)$

$is\_married\_to : CITIZEN \rightarrow \mathbb{B}$

$\forall\, a : CITIZEN \bullet \neg\,(a = Void) \Rightarrow a \in \operatorname{dom} is\_married\_to$

$\forall\, a : CITIZEN \bullet a \in \operatorname{dom} is\_married\_to \Rightarrow is\_married\_to(a) = (a = spouse)$

$retired \mathrel{\widehat{=}} active$

$name :\downarrow \operatorname{seq} CHAR$

$spouse :\downarrow CITIZEN$

$children :\downarrow \mathbb{P}\, CITIZEN$

$parents :\downarrow \mathbb{P}\, CITIZEN$

$company :\downarrow EMPLOYER$

$details : CITIZENSHIP$©

$parents.count \leq 2$

$single \lor spouse.spouse = \mathrm{self}$

$\forall\, c \in children \bullet \exists\, p \in c.parents \bullet p = \mathrm{self}$

INIT

$company = Void$

$details = Void$

$name = \langle\,\rangle$

$spouse = Void$

$children = \{\,\}$

$parents = \{\,\}$

divorce

$\Delta(spouse)$

$\neg\, single$

$single'$

$spouse.single$

Figure 7: LaTeX processed output from automatically generated Object-Z

The process now splits; in one branch, the typical BON development process can be followed through implementation, refactoring, and review, typically targetting Eiffel as a programming language. Thus, we effectively use Object-Z as an analysis tool.

3. In the second branch of the split, an Object-Z development can be followed, e.g., via formal refinement. This branch likely will not target a specific programming language, and thus in a sense it is more flexible than the BON branch.

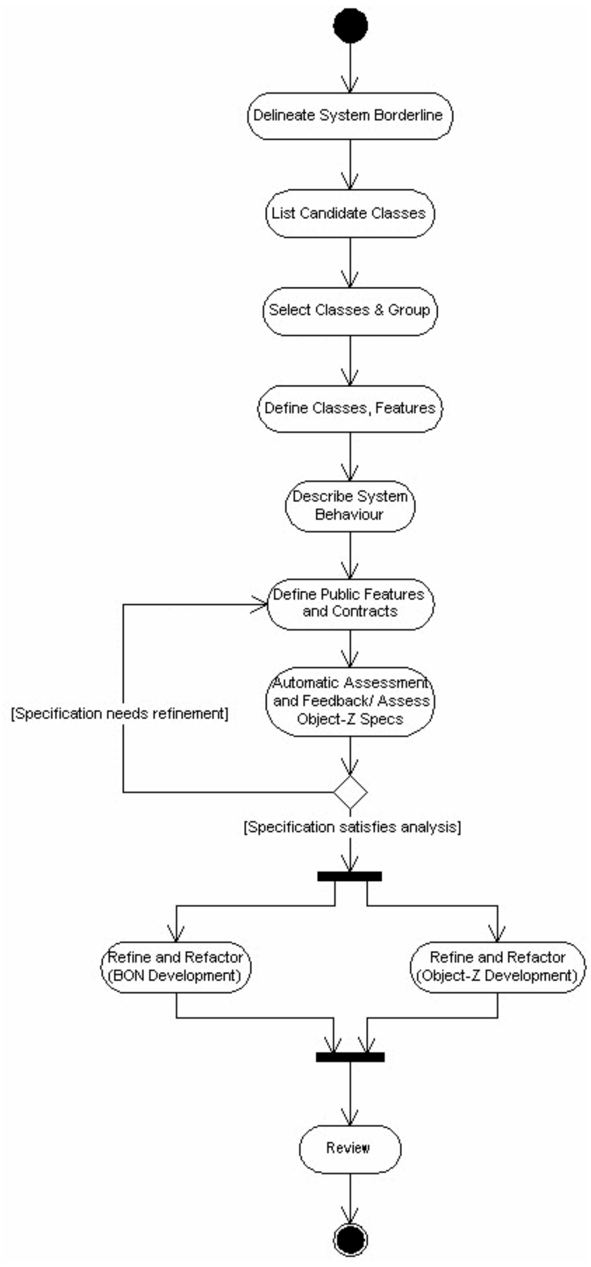

Delineate System Borderline
List Candidate Classes
Select Classes & Group
Define Classes, Features
Describe System Behaviour
Define Public Features and Contracts
Automatic Assessment and Feedback/ Assess Object-Z Specs
[Specification needs refinement]
[Specification satisfies analysis]
Refine and Refactor (BON Development)
Refine and Refactor (Object-Z Development)
Review

Figure 8: Generalised BON process integrating Object-Z

We next describe the BON-CASE tool, which not only supports the construction of BON models, but also helps to automate many stages of the generalized process, including the automatic generation of Object-Z specifications.

4 TOOL SUPPORT FOR THE INTEGRATED METHOD

BON-CASE [14] is an open-source CASE tool for BON. It provides support for automatic generation of Object-Z, as we discuss shortly. The CASE tool (version 0.5b3) supports the full BON notation, including static diagrams (classes and interfaces, clusters, relationships, and assertions), and collaboration diagrams (objects, messages, and scenarios). It also supports UML use case diagrams, including syntax for templates, use case relationships, and stereotypes.

A critical component of the tool is its code generation engine. The code generator, designed using the Template pattern, abstracts the code generation process from concrete implementations of abstract syntax tree walkers. Current code generators supported by BON-CASE include: ASCII BON, XML, C#, Java, Eiffel, JML [7], and now Object-Z. The Object-Z code generator implements the translation rules discussed in Section 3.

The BON-CASE tool can be used during the first six tasks of the generalized BON process. The Object-Z code generator can be applied during task 7, and the results of applying the Object-Z Wizard type checker or the Isabelle/HOL embedding to the automatically generated Object-Z specifications can be fed back in to the BON process in task 8. The feedback provided by the Object-Z checker is, of course, given in terms of Object-Z syntax, but the relative similarity of this syntax to that of BON does not make it difficult to use this feedback in modifying the original BON models. Adding reverse engineering facilities to BON-CASE, as we are currently doing, will help here. We note as well that the semantics of BON and Object-Z have much in common (e.g., in terms of contracts and classes) so that we can expect and have received useful feedback from the Object-Z checker about errors or problems with our BON models.

5 DISCUSSION, LESSONS LEARNED, AND FUTURE WORK

The integration of BON and Object-Z presented in this paper provides Object-Z developers access to mainstream software engineering languages and tools. A secondary motivation for defining the translation from BON to Object-Z, and for implementing the translation within the BON-CASE tool, was to be able to use Object-Z tools with BON, and to use OO graphical notations and programming languages with Object-Z. However, there were other reasons for carrying out the integration as well.

- *Integration with mainstream OO languages.* BON-CASE supports several OO diagramming notations, particularly collaboration diagrams and use case diagrams. Since the tool also supports several programming languages (e.g., Java, Eiffel, C#), it provides a means for using these technologies as well.

- *Providing a development process for Object-Z.* By integrating BON and Object-Z, we have augmented the latter with the development process of the former. Since this development process includes tasks and phases that are intrinsic to most systems development process – e.g., risk assessment, feedback – this provides a way of showing how to use Object-Z techniques, if desired, in more mainstream development processes.

- *Graphical views of Object-Z specifications.* This integration provides a visual front-end for Object-Z.

- *Exploiting Object-Z integrations.* Object-Z has been integrated with several other technologies, e.g., Timed CSP. The integration of BON and Timed CSP thus lets developers use BON together with these other technologies.

A point worth mentioning is the generality of the approach to language integration and process augmentation that was applied. Without any change, the approach described in this paper has been applied to integrating BON with the JML modelling language, and a case study demonstrating the effectiveness of the methodology – and some of the limitations of the tool support for the integration – has been carried out.

We have taken a pragmatic approach to integration in this paper: we desired to use Object-Z tools with BON, and BON tools with Object-Z and we implemented a translation to effect this. The translation helped us attain the practical goal of adding a development process to Object-Z. We have not yet proven the soundness of the translation, nor the correctness of its implementation. Our experiments with Object-Z's tools (particularly Wizard and LaTeX styles) have given us greater confidence in the correctness of the translation. It would be beneficial to have a proof of soundness. This is made more challenging by the size of the BON and Object-Z languages, and the relative imprecision that still remains in the semantics of BON.

A key limitation with the integration – and the implementation in the BON-CASE tool – is the inability to reverse the translation, i.e., to take manually modified Object-Z specifications and reverse engineer a BON specification from it. The BON-CASE tool alpha currently supports reverse engineering only for Eiffel programs, though the infrastructure to allow extensions is present in its design. We are currently defining a reverse mapping, from Object-Z to BON, and plan to implement it along with other reverse engineering facilities in the tool in the near future.

REFERENCES

[1] L. Baresi and M. Pezzé. Toward formalising structured analysis. *ACM Trans. Soft. Eng. and Method.* 7(1):80-107, January 1998.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The UML Reference Guide,* Addison-Wesley, 1999.

[3] J. Sun, J.S. Dong, J. Liu, and H. Wang. Object-Z web environment and projections to UML. In *Proc. WWW'01*, ACM Press, May 2001.

[4] R. Duke and G. Rose. *Formal Object-Oriented Specification using Object-Z*, Palgrave, 2001.

[5] W. Johnston. A Type-Checker for Object-Z, SVRC Technical Report TR96-24, University of Queensland, July 1996.

[6] S.-K. Kim and D. Carrington. A formal mapping between UML models and Object-Z specifications. In *Proc. ZB-2000*, LNCS 1878, Springer-Verlag, 2000.

[7] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML. Technical Report 98-06i, Department of Computer Science, Iowa State University, Feb. 2000.

[8] Moby Research Group. Moby/OZ Tool, 2002. http://theoretica.informatik.uni-oldenburg.de

[9] R.F. Paige. A meta-method for formal method integration. In *Proc. Formal Methods Europe 1997*, LNCS 1313, Springer-Verlag, September 1997.

[10] R.F. Paige. Pure formal method integration via heterogeneous notations. *Formal Aspects of Computing* 10(3), June 1998.

[11] R.F. Paige. Integrating a program design calculus with a subset of UML. *The Computer Journal* 42(2), March/April 1999.

[12] R.F. Paige. When are methods complementary? *Information and Software Technology* 41(3), February 1999.

[13] R.F. Paige and J.S. Ostroff. From Z to BON/Eiffel. In *Proc. Automated Software Engineering 1998*, IEEE Press, October 1998.

[14] R.F. Paige, L. Kaminskaya, J.S. Ostroff, and J. Lancaric. BON-CASE: an extensible CASE tool for formal specification and reasoning. *Journal of Object Technology* 1(3):65-87, August 2002.

[15] S. Schneider. *Concurrent and Real-Time Systems*, Wiley, 2000.

[16] G. Smith. *The Object-Z Specification Language,* Kluwer, 2000.

[17] G. Smith, F. Kammüller, and T. Santen. Embedding Object-Z in Isabelle/HOL. In *Proc. ZB-2002*, LNCS 2272, Springer-Verlag, January 2002.

[18] G. Smith. Introducing Reference Types by Refinement. In *Proc. ICFEM 2002*, LNCS 2495, Springer-Verlag, October 2002.

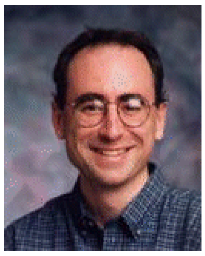
[19] J.M. Spivey. *The Z Specification Language,* Second Edition, Prentice-Hall, 1992.

[20] K. Stirewalt and L. Dillon. A component-based approach to building formal analysis tools. In *Proc. ICSE 2001*, IEEE Press, May 2001.

[21] K. Taguchi and J.S. Dong. An overview of Mobile Object-Z. In *Proc. ICFEM 2002*, LNCS 2495, October 2002.

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

ABOUT THE AUTHORS



Richard Paige is a lecturer at the University of York, United Kingdom, where he works with the High-Integrity Systems Group and is a co-leader of the Software and Systems Modelling Team. He completed his PhD in Computer Science at the University of Toronto in 1997. E-Mail: paige@cs.york.ac.uk.

Phillip Brooke is a senior lecturer in the School of Computing, Communications, and Electronics at the University of Plymouth, United Kingdom, where he works with the Network Research Group. He completed his DPhil in Computer Science at the University of York in 1999. E-Mail: phil.brooke@plymouth.ac.uk.