Developing BON as an Industrial-Strength Formal Method

Richard F. Paige and Jonathan S. Ostroff

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

Abstract. The emerging Unified Modelling Language has been touted as merging the best features of existing modelling languages, and has been adopted by leading companies and vendors as a universal software modelling language. Some researchers are also looking to UML as a basis for formal methods development. A less known approach is BON (the Business Object Notation), which is based on the principles of seamlessness, reversibility and design by contract, making it an ideal basis for industrial-strength formal methods development of object-oriented software. In this paper, we argue that BON is much more suited for the application of formal methods than UML. We describe the properties that an industrial-strength formal method must have, show how algorithm refinement can be done in BON (as an example of using BON for formal development), and contrast BON with other approaches, including UML, Z, B and VDM.

1 Introduction

> *"UML is in the middle of of a standardization process with the Object Management Group, and I expect it will be the standard modeling language in the future."* Martin Fowler, **UML Distilled** [5].

The emerging Unified Modelling Language has been touted as merging the best features of existing software design notations, and has been adopted by leading companies and vendors as a universal software modelling language. Some researchers are also looking to UML as a basis for the application of formal methods in software development [4]. A less known approach is the Business Object Notation (BON) [21]. BON addresses roughly the same problem space as UML. We argue in this paper that BON is much more suited for the application of formal methods than UML and other alternatives.

In the previous decade, many companies made a move from C to C++ in the hope of developing more reliable software. This move may not have resulted in the the expected gains — or worse, may actually have impeded software development and maintenance [8]. We have to ask ourselves whether the move to UML may also be premature.

The UML notation summary of approximately sixty pages makes its syntax as difficult to master as any of the most opaque formal methods, even without any formal semantics. A notation that supports every conceivable feature is not necessarily the best notation for development. Rather, what is needed is an integrated set of features that capture the essence of software abstractions, and which are directly mappable to

implementation (and the reverse, from implementation to abstractions) so that the development can proceed seamlessly. Finally, a notion of software contracting in which clients and suppliers have obligations and benefits is needed as a basis for formal specification and analysis of software products. BON is just such a method, and its focus on seamlessness, reversibility and design by contract make it an ideal basis for industrial-strength formal methods development of object-oriented software.

In this paper, we outline the BON method of Walden and Nerson [21]. We explain how BON satisfies a number of criteria that have been suggested as being essential for a mathematical method to be industrially applicable. Our thesis is that BON provides a superior method for building large-scale, reliable, reusable software systems, and allows emphasis of mathematical or informal development, as the software engineer sees fit. Further, we attempt to suggest why the BON specification language has significant advantages over widely used informal and formal approaches like UML (with its constraint language, OCL), Z, and B.

The paper is organized as follows. We commence by suggesting a number of criteria for an industrial-strength formal method. BON is then presented, and an example is used to illustrate how BON supports the design of large systems, and how it can be used as a purely mathematical method. In presenting the use of BON as a formal method, we demonstrate an example of *method reuse*, and show that Z algorithm refinement rules can be reused and applied with only minor syntactic changes to BON specifications. We discuss which industrial-strength criteria BON does not currently support, and how we plan to amend these limitations. Finally, we contrast BON with other approaches, and consider future work.

2 Criteria for an Industrial-Strength Formal Method

An industrial-strength formal method is a mathematically based technique that is applicable to large-scale, complex software development problems. We suggest several properties that such a method might possess, synthesized from the experience of others, particularly [7, 22], and our own findings.

An industrial-strength formal method should satisfy at least the following.

1. *Restrictability of formality*, the ability to restrict the use of formal techniques – e.g., specification and verification – to those aspects of the development that require them [22].
2. *Gradual introduction capabilities*, the ability to introduce formality gradually, over time, into a development setting [17, 22].
3. *Tool support*, for both standard software engineering tasks like compilation, debugging, version control, and diagramming, as well as for formal manipulations.
4. *Modularity*, the ability of the formal method to produce modular designs [14].
5. *Seamlessness.* The method should apply to the full software life cycle, in a way that minimizes the gaps between successive activities.
6. *Education.* The method must be founded on an appropriate educational programme.

We now consider each aspect in more detail, and suggest why it is important for any industrial-strength formal method to satisfy all.

2.1 Restrictability of formality

Formality in software development is often expensive to use, and it is typically infeasible to apply in full. Software systems are usually too large and too complex to fully develop via formal methods. Further, formal method use often depends on having clear, precise requirements; for a system with vague requirements, formal methods may be of only minimal help.

It should therefore be possible for engineers to use formality only when their experience and education suggests that it is appropriate. Specifications that are deemed to be complex by engineers should be subject to formal techniques, and those requirements that are either too vague, or are straightforward to specify and implement via informal techniques, can be handled differently.

Some critical system components may need to be developed formally, in which case restrictability is not an issue. However, we can view the development of such systems as being at one end of the spectrum of use of formality: that part where only formal methods are used. Thus, all industrial-strength formal methods need to be restrictable.

Because we require restrictability of formality, we should be able to use other informal techniques for developing the remaining components of our system. It should be possible to directly code some components, produce others by reuse, formally specify and informally develop others, and even abstract specifications from programs. An implication of restrictability is that it will be useful for specification languages to be *wide-spectrum* [9], encompassing both abstract, mathematical specifications, as well as an executable programming language subset.

Restrictability is also useful for coping with complex specifications and their refinements. When refining significant specifications, invariably the engineer loses track of the initial specification after only a small number of refinement steps: it is hard to maintain a view of the context in which the refinement is taking place. Restrictability, in cooperation with modularity (see Section 2.4, below) can help in ensuring that the engineer does not lose track of the refinement context.

2.2 Gradual introduction of formality

Many software development projects do not currently use formal methods, and so if it is desired to move towards the use of mathematical methods, they need to be introduced into the existing engineering macro- and micro-processes. Ideally, the introduction needs to be done so that the engineers and managers can adapt to the use of the new methods, evaluate the use of formal techniques (primarily with respect to their own processes), and decide the extent to which formality is usable in their context; this may have to be done on a project-by-project basis.

Thus, it should be possible to produce a migration path for a formal method, so that its use can be brought into practice gradually over time. With gradual introduction, it is possible to assess the effectiveness of the formal method in the development setting, and to determine the appropriate extent in which formality should be used. Gradual introduction is important in helping to fit formal methods to the development context—and in particular, to the macro-process in which formality is being used.

Gradual introduction goes hand-in-hand with restrictability: if one has a restrictable formal method, then one can gradually introduce it over time. An ideal form of restrictability is one in which the developers can change the extent of use of formality even while development is ongoing. In particular, if it is discovered that the current extent of use of formality is inappropriate, developers should be able to change the extent of use without altering the rest of the specification, design, and development.

2.3 Tool support

Industrial-sized software engineering projects usually require industrial-strength tools to support development. In projects that do not apply formality, tool support is typically for diagramming, design, and animation (e.g., via CASE tools), configuration management and version control, compilation and debugging. For industrial-strength formal methods, tools should also exist to support the production, syntax, and type-checking of specifications, as well as automatic or semi-automatic discharge of the proof obligations that arise in refinements. Tools may also be provided to help manage any refinement or verification process. Analysis tools should be as transparent to the engineer as possible, and, ideally, should be integrated or integrable with other standard development tools, like the aforementioned CASE systems, compilers, and debuggers. In integrating analysis tools with other development tools, an *integrated development environment*, or *application framework* may have to be produced in order to simplify communication between tools, and to hide the tool integration details from the software engineers.

2.4 Modularity

A modular software design is often a quality software design, in part, because it can lead to maintainable software [14]. If a software system is designed and implemented as a set of precisely specified interacting modules, then software maintenance can be easier to carry out. A common form of a module in software design is a *class* [14].

Modularity is an essential quality in an industrial-strength formal method. Modularity lets developers focus their attentions on small parts of a system at a time. Small parts of a system can be easier to understand and develop; therefore, usually, small parts of a system are easier to formally analyze. Further, if developers are able to restrict their attentions at one time to small parts of the system, they will be less likely to lose track of the context in which the formal analysis is being carried out. A formal method that can be applied to a class or a package, for example, will be applied strictly to the functions and procedures of the class or package. A modular software development method thus allows developers to focus their attentions on separate components of the system.

Simply having a modular software development method does not guarantee the production of quality systems, nor does it guarantee maintainability. Design principles, e.g., for obtaining cohesive, low-coupled modules, must still be followed.

2.5 Seamlessness

A seamless method is one that can be applied to the entire software life cycle (analysis through to implementation and maintenance). The method is devised in such a way so

as to minimize the gaps that exist between successive development activities. Seamlessness attempts to recognize the unity that exists in software development activities. Whether we are analyzing a problem, designing a solution, or implementing code, the same intellectual challenges arise and the same structuring mechanisms are needed.

A seamless method, such as BON, provides many benefits. For one, semantic gaps (impedence mismatches) can be avoided when making transitions between development steps. Mismatches between analysis and design, design and implementation, etc., have caused significant trouble in software development. Further, by using one form of model from the start of development, a close correspondence between the problem description and the solution description can be maintained. Finally, seamlessness facilitates reversibility, the inevitable backward adjustments that must occur during development.

Reversing a process is inevitable in software development. Invariably, when implementing a program, a developer learns that the software could accomplish its tasks better. The method should support the developer in making backward adjustments when they are discovered. Seamlessness is essential in facilitating this task.

2.6 Education

For a method or tool to be used and accepted within industry, it must be taught in an educational programme. This is certainly the case with formal methods, which require a solid mathematical and Computer Science background to understand and apply. Teaching a formal method or using a software tool in a university programme can have a significant effect on practice. The paper [16] raises the example of Spice in the domain of electrical engineering. Electrical engineering programmes taught differential equations and used the Spice tool; now, Spice is part of the toolset of electrical engineers, and is widely applied outside of the university environment. Methods and tools of similar strength, soundness, and ease of use are needed for the practice of software engineering. And educational programmes founded on such methods and tools are essential for software engineering to become a professional discipline.

3 BON: Towards an Industrial-Strength Formal Method

An ideal industrial-strength formal method satisfies the properties listed and outlined in the previous section. We now suggest a method that satisfies some, but not all, of these properties. In the next section, we explain changes or extensions to the method that need to be made to support all of the properties given in Section 2.

The Business Object Notation (BON) [21] is a method possessing a recommended process as well as a graphical and a separate textual notation for specifying and describing object-oriented systems. The notation provides mechanisms for specifying inheritance and usage relationships between classes, and has a small collection of techniques for expressing dynamic relationships (e.g., message passing). The notation also includes an *assertion language* for specifying preconditions and postconditions of class features as well as class properties. It is supported by the EiffelCase tool [14].

BON was designed to support three main techniques: seamlessness (discussed in Section 2.5), reversibility (the ability to extract BON graphical specifications from programs automatically), and software contracting (discussed in Section 3.1). As a result, BON provides only a small collection of powerful modeling features that guarantee seamlessness and full reversibility on the static modeling notations.

The fundamental specification construct in BON is the class. A BON class has a name, an optional *class invariant*, and a collection of *features*. A feature may be a query—which returns a value and does not change the system state—or a command, which does change system state. BON does not include a separate notion of attribute. Conceptually, an attribute should be viewed as a query returning the value of some hidden state information. Thus, identical syntax is used for accessing and writing attributes as for queries; this is the so-called *uniform access* principle [14].

Fig. 1 contains a short example of a BON graphical specification of a class *CITIZEN*. Class features are in the middle section of the diagram, while the class invariant is at the bottom. The invariant is a predicate (conjoined terms are separated by semicolons) that must be *true* whenever an instance of the class is used by another object (in the invariant @ refers to the current object). Class *CITIZEN* has seven queries, and one command. In particular the class has 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 with the usual operators (e.g., $\in$, *add*).

Feature behaviour is written using *contracts*, in a pre- and postcondition form. Preconditions of features are annotated with question marks in boxes, while postconditions are annotated with exclamation marks in boxes. Visibility of features is expressed by sectioning and use of the feature clause. By default, features are accessible to client classes; this can be changed by writing a new section of the class interface and prefixing the section with a list of client classes that may access the features.

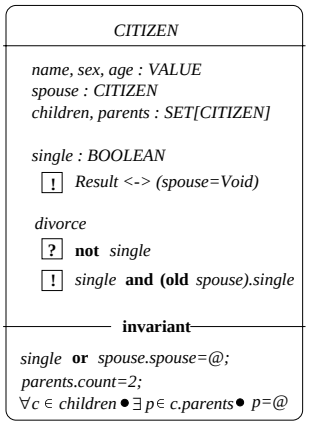

Fig.1. Class Interface for *CITIZEN*

3.1 Design by contract

The pre- and postconditions of BON feature specifications are called contracts. The notion of design by contract [14] is central to BON. Each feature of a class has a contract, and interactions between the class and *client* classes must be via this contract. The contract is part of the official documentation of the class; the class specification and the contract are not separated. The feature contract places obligations on the client of the feature (who must establish the precondition) and supplies benefits to the client of the feature (who can rely that the feature will establish the postcondition).

Design by contract is tightly integrated into BON. The recommended BON process suggests providing contracts for all features, written using the BON assertion language. BON obeys all rules with respect to contracts that Eiffel does. In particular, BON obeys the Eiffel rules with respect to redefinition of features that have contracts: a feature that has its semantics redefined in a subclass can only weaken the precondition and strengthen the postcondition of the feature in the parent class.

3.2 The BON assertion language

Contracts in BON are written in an assertion language, a dialect of predicate logic. Assertions are statements about object properties. These statements can be expressed directly, using predicate logic, or indirectly by combining boolean queries from individual objects. The basic assertion language contains the usual elements, such as propositional and arithmetic operators and constants. Expressions constructed from these elements are easy to translate into executable code; effectively, this subset of the BON assertion language is the Eiffel assertion language.

The BON predicate logic can be used when more expressive power is needed. The predicate logic introduces set operations (e.g., membership, union), and universal and existential quantifiers. The quantifier notation in BON is akin to that used in the Logic **E** of Gries and Schneider [6] (with a few minor syntactic differences).

The BON assertion language can also be used to refer to the prestate in the postcondition of a feature. The **old** keyword, applied to an expression *expr*, refers to the value of *expr* before the feature was called. **old** can be used to specify how values returned by queries may change as a result of executing a command. Most frequently, **old** is used to express changes in abstract attributes. For example, *count* $=$ **old** *count* $+ 1$ specifies that *count* is increased by one.

3.3 Seamlessness

BON was originally designed to work seamlessly with Eiffel; in particular, the syntax of textual BON closely resembles Eiffel's syntax for pre- and postconditions. Making a transition from a BON specification to an Eiffel program is often easier than transitioning to alternative languages (e.g., Java or C++), because Eiffel supports pre- and postconditions (including the **old** keyword) as well as class invariants. The EiffelCase tool supports the automatic generation of Eiffel code from BON specifications, as well as the full reverse engineering of specifications from code.

In this paper, we view an Eiffel specification as a special case of a BON specification: one that just happens to be immediately executable. We therefore view BON as a wide-spectrum language, which supports easy refinement of (abstract) BON specifications into (concrete) Eiffel programs.

3.4 Architectural diagrams

A great deal of the value of BON comes from its assertion language, but also from its ability to clearly and precisely specify architectural elements of an object-oriented design. BON provides a small selection of *relationships* that can be used to specify how classes, in a design, interact. There are two ways in which classes can interact in BON.

- *Inheritance:* one class can inherit behavior from one or more parent classes.
- *Client-supplier:* one class uses a second class in some way. This is used to specify the 'has-a' or 'part-of' relationships between classes.

Fig. 2 contains a non-trivial architectural diagram using BON, demonstrating examples of both inheritance and client-supplier. Classes are drawn as ellipses. Thin vertical arrows (e.g., between *EXP* and *SD*) represent inheritance. Double-line arrows with thick heads (e.g., between *FTS* and *TRANSITION*) represent client-supplier. Dashed boxes are *clusters*, which encapsulate subsystems; clusters are a purely syntactic notion. Inheritance and client-supplier relationships are recursively extended to be applicable to clusters, as the figure shows.

The EiffelCase drawing tool supports production and browsing of such diagrams, as well as automatic code generation from the diagrams, and the reverse engineering of such diagrams from Eiffel programs. The CASE tool works cooperatively with the EiffelBench compiler and debugger, the latter of which can be used to syntax and type check specifications (that do not use quantifiers).

3.5 Algorithm refinement in BON

With BON, the use of mathematics is restrictable; an engineer need not formally specify the behavior of features or classes. But BON can also be used in a completely mathematical manner. That is, BON specifications can be treated as mathematical expressions, and formal analysis and reasoning can be applied to these expressions.

Let us consider an example, for implementation correctness, i.e., for showing that an implementation of a class in Eiffel satisfies a specification of a class in BON. This is a feature-by-feature process, carried out by showing that each Eiffel feature implementation satisfies its corresponding BON feature specification. Implementation correctness can be a verification process or a refinement process; in the latter, implementation and proof are developed hand-in-hand. We consider the refinement process here.

Suppose that we have a feature *S* in a class that also possesses attributes *a*. The feature also introduces local variables *v*. A *specification* associated with feature *S* is a predicate with free variables *a* or *v*, where a free variable may optionally be prefixed with the keyword **old**.

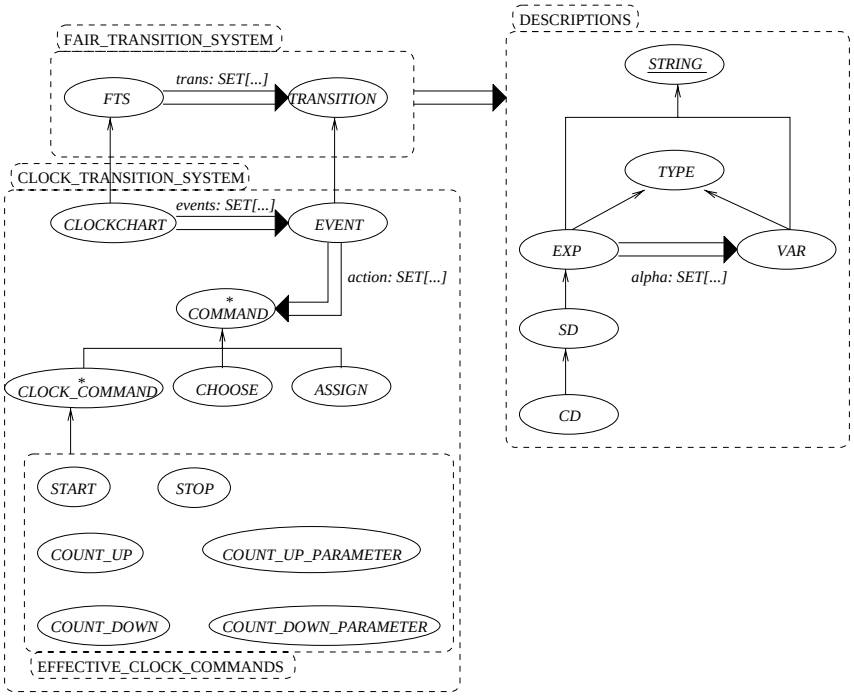

Fig.2. BON architectural diagram for fair transition systems

There is one particular specification associated with a feature that is of most interest to us in refinement; this specification arises when a feature has a contract. This specification, **spec** *S*, formally describes the meaning of the feature based on the contract. Suppose we have feature *S* with **require** clause *P* and **ensure** clause *Q* (*P* and *Q* each include the invariant). The **require** clause of *S* is a predicate on the prestate of the object, while the **ensure** clause is a predicate on the pre- and poststate. The meaning of this specification is given by **spec** *S*, defined below.

$$\textbf{spec}\ S \quad \widehat{=} \quad (\textbf{old}\ P) \rightarrow Q$$

If the precondition **old** *P* is *false*, then any behavior will satisfy the specification.

An algorithm refinement relationship can be defined for BON feature specifications, allowing the stepwise refinement of a specification down to the level of an Eiffel program. This is possible for two reasons: because BON is part of a seamless development method that can result in Eiffel programs; and, because we will allow a specification to appear wherever Eiffel allows a statement.

In BON, refinement is just implication. Suppose we have a feature *T* that we think refines *S*. *S* is refined by *T* (written $S \sqsubseteq T$) if

$$\forall\, \textbf{old}\ \sigma, \sigma \bullet (\textbf{spec}\ S \leftarrow \textbf{spec}\ T)$$

where σ is the attributes of the class in which feature *S* occurs. In a refinement, we need only concern ourselves with the attributes of the class; this can reduce the complexity of the proof obligations that we must discharge in refinement.

Refinement of a BON specification (or verification of a program against a BON specification) can be applied to large software systems, because the refinement relationship $\sqsubseteq$ is monotonic over class relationships. Suppose we have a class *A*, with a number of feature specifications, which we want to refine to code. The process is as follows. First, determine **ind** *A*, the set of all classes that *A* directly depends on (via inheritance or client-supplier). Then, determine **spec** *A*, the set of contracts for all features in *A* (including inherited features). To refine *A* to an Eiffel program, we must refine each contract of *A*, i.e.,

$$\forall\, s \in \textbf{spec}\ A \bullet s \sqsubseteq \textbf{body}\ s$$

where **body** *s* is an implementation of *s* in Eiffel. To carry out this process, we *only* need **spec** *A* and the contracts of **ind** *A*. We do not need any implementations of **ind** *A*, nor do we (in most cases) need to consider the entire system. Thus, refinement can be done compositionally, feature by feature.

The ultimate goal of algorithm refinement is to formally derive a program from a specification. Because BON works seamlessly with Eiffel it is simplest to refine BON specifications into Eiffel. This requires us to provide rules for refining BON specifications into Eiffel.

An interesting result associated with BON refinement rules is that we can *reuse* all existing Z refinement rules — such as those from [23] — assuming that a consistent notation for distinguishing pre- and poststate is used (we will use BON's **old** notation to distinguish prestate from poststate). In other words, a valid refinement rule for Z is a valid refinement rule for BON features. This follows from Theorem 1.

Theorem 1. *Let S and T be BON specifications as described above, and let* $\sqsubseteq_Z$ *be the Z definition of refinement (see [20], or the proof below). Then, under syntax,*

$$S \sqsubseteq_Z T \rightarrow S \sqsubseteq T$$

Proof. Expand the definitions of $\sqsubseteq_Z$ and $\sqsubseteq$. The PVS theorem prover discharges the proof automatically, after applying the definitions, using one instance of `(grind)`. The PVS specification of the conjecture used to discharge the proof is as follows (in the theory, `p` and `q` are the pre- and postcondition of *S*, while `u` and `v` are the pre- and postcondition of *T*). Equational proof details can be found in [18].

```
refmaps : THEORY
BEGIN
 STATETYPE : TYPE
 old_s, s  : VAR STATETYPE
 p(old_s:STATETYPE)   : bool
 q(old_s,s:STATETYPE) : bool
 u(old_s:STATETYPE)   : bool
 v(old_s,s:STATETYPE) : bool

 refmapping : CONJECTURE
   (FORALL old_s, s :
      (p(old_s) IMPLIES (u(old_s) AND
      (v(old_s,s) IMPLIES q(old_s,s))))
      IMPLIES
      ((u(old_s) IMPLIES v(old_s,s)) IMPLIES
       (p(old_s) IMPLIES q(old_s,s))))
END refmaps
```

An implication of this theorem is that we do not have to formulate new refinement rules for BON, and can make use of the extensive collection of Z refinement rules that already exist, providing that we transform the Z syntax into BON's syntax.

We can use the refinement rules in Wordsworth [23] with BON. Wordsworth treats Z schemas as summarizing the operation of programs written in a simple guarded command language. Values of program variables before and after the execution of the corresponding program make up the declaration part of the schema, and the predicate part expresses the relation between those values induced by the execution of the schema. We can do the same, treating BON feature contracts as summarizing the operation of Eiffel programs. Here are some examples, translated into BON syntax. In the following, let S, A, and B be BON feature specifications. We extract the precondition of a feature with the **pre** operator, and the postcondition with the **post** operator. Assume that refinement is being done within a single class, with attributes x and y (we use σ to stand for both x and y). We use $\sqsubseteq$ to denote the refinement relation.

First, we consider how to introduce an assignment statement in a refinement.

Rule 2. Introducing assignment. Let e be an expression whose type is compatible (based on the BON/Eiffel notion of compatibility [14]) with x. Then $S \sqsubseteq x := e$ if

$$\forall \textbf{ old } \sigma, \sigma \bullet (\textbf{pre } S \wedge (x = \textbf{old } e) \wedge (y = \textbf{old } y)) \rightarrow \textbf{post } S$$

The effect of the assignment $x := e$ is formally described by the BON specification

$$\begin{array}{l}\textbf{require } \mathit{true}\\ \textbf{ensure } x = \textbf{old } e \wedge y = \textbf{old } y\end{array}$$

The next rule is for introducing a selection. A selection has the following form.

$$\textbf{if } c_1 \textbf{ then } P_1 \textbf{ elseif } c_2 \textbf{ then } P_2 \ \ldots \ \textbf{else } P_k \textbf{ end}$$

where the c_i are conditions and the P_i specifications. In the refinement rule, we consider the two-branch case, which generalizes in the obvious way to the multi-branch setting.

Rule 3. Introducing if. Let b be a *BOOLEAN* expression. Then

$$S \ \sqsubseteq \ \textbf{if } b \textbf{ then } A \textbf{ else } B \textbf{ end}$$

providing that

$$\begin{array}{c}\forall \sigma, \textbf{old } \sigma \bullet (\textbf{spec } S \wedge b) \leftarrow \textbf{spec } A\\ \forall \sigma, \textbf{old } \sigma \bullet (\textbf{spec } S \wedge \neg\, b) \leftarrow \textbf{spec } B\end{array}$$

This rule is a formalization of the notion of refinement by *cases* [9].

The refinement rules become more complicated when we consider sequencing and repetition, because of the need to introduce intermediate states. Before we consider the rule for sequencing, we introduce some new notation, allowing us to talk about intermediate states. We annotate specifications with primes (e.g., A') to indicate systematic addition of primes to *variable names* used within the specification. This notation is borrowed from [23]. However, we must make one slight adaptation in BON, because of its use of **old** to distinguish pre- and poststate: a prime applied to an **old** expression removes the **old** keyword. Here is an example.

$$\begin{array}{r}(x = \textbf{old } y \wedge y = \textbf{old } (x+y))'\\ = x' = y \wedge y' = (x+y)\end{array}$$

Now we can provide a rule for introducing sequencing.

Rule 4. Introducing sequencing. $S \sqsubseteq A;\ B$ providing that

$$\begin{array}{r}\textbf{pre } S \rightarrow \textbf{pre } A\\ (\textbf{pre } S \wedge \textbf{spec } A) \rightarrow (\textbf{pre } B)'\\ (\textbf{pre } S \wedge \textbf{spec } A \wedge \textbf{spec } B') \rightarrow (\textbf{spec } S)[\_'/\_]\end{array}$$

where, in the last line, the notation $X[\_'/\_]$ reads "substitute primed versions of variables for unprimed versions of variables in X" (and don't change the **old** variables).

Finally, we can give a rule for introducing a loop. We use Wordsworth's initialized loop rule. Loops in Eiffel have the following syntax.

$$Loop \mathrel{\widehat{=}} \textbf{from } Init \textbf{ until } b \textbf{ loop } P \textbf{ end}$$

Init and P are specifications with **require** and **ensure** clauses, while b is a boolean expression.

Rule 5. Introducing an initialized loop. Let b be a boolean expression, I a loop invariant, and *Loop* a loop as above. Then $S \sqsubseteq Loop$ if

$$\begin{aligned}
\textbf{pre } S &\rightarrow \textbf{pre } Init \\
\textbf{pre } S \wedge \textbf{spec } Init &\rightarrow I \\
\textbf{pre } S \wedge I \wedge b &\rightarrow \textbf{spec } S \\
\textbf{pre } S \wedge I \wedge \neg\, b' &\rightarrow (\textbf{pre } P)' \\
\textbf{pre } S \wedge I \wedge \neg\, b' \wedge (\textbf{spec } P)' &\rightarrow I[_'/_]
\end{aligned}$$

We have shown that Z refinement rules can be applied to BON. We intend to develop a collection of refinement rules, written directly in BON, and expect the rules to be simpler to write and use than the reused ones.

Refinement (and verification) in BON is industrial-strength; it is restrictable and can be applied on a class-by-class and feature-by-feature manner. Further, with BON there is no system-level validity check that has to be discharged to show system correctness. To show that a system is correct, we must verify all classes (including the root class). When the root class is finally verified, the system has been shown to be correct.

3.6 Example of refinement

This section illustrates a simple example of refinement in BON, demonstrating the use of Z refinement rules with BON. The refinement will transform a BON specification into an Eiffel program.

The problem we will solve is a simple one, taken from [23], to find the maximum of a non-empty array (or sequence) of integers. We suppose that we have a class, *FOO*, that includes a feature that will be used to determine the maximum of the array. This class has the following BON specification.

The array for the computation is a feature *s* of the class. The function *max_array* calculates the maximum of the array. The function introduces a local variable, *i*, as a loop index. A local variable *result*, automatically declared for the function, will hold the result of the computation. The introduction of these variables is formally defined by existential quantification over the specification. The notation $s.item(j)$ is the Eiffel syntax for the array index operation. $s.capacity$ is the maximum size of the array, while $s.lower$ and $s.upper$ are its lower and upper bounds. $s.inrange(i)$ is *true* iff *i* is in range of the subscripts of *s*.

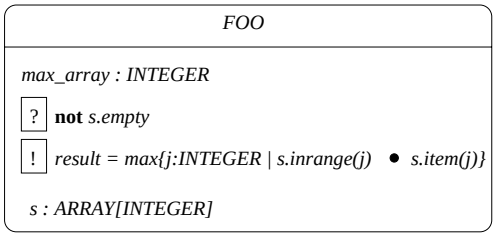

Fig.3. Class *FOO* for refinement example

We can now refine the specification to code. Clearly, the implementation will require a loop. The first step of the refinement will introduction such a loop, with invariant

$$I \mathrel{\widehat{=}} result = max\{j : INTEGER \mid s.lower \leq j \leq i \bullet s.item(j)\} \wedge i \leq s.upper \wedge s = \textbf{old}\ s$$

and guard $b \mathrel{\widehat{=}} (i = s.upper)$. The first refinement step is

$$\begin{array}{l} \quad \textbf{spec}\ max\_array \\ \sqsubseteq\ \textbf{from spec}\ P \\ \quad \textbf{until}\ i = s.upper \\ \quad \textbf{loop}\ \ \textbf{spec}\ Q\ \ \textbf{end} \end{array}$$

where

$$\begin{array}{lcl} \textbf{spec}\ P & = & i = s.lower \wedge result = s.item(s.lower) \\ \textbf{spec}\ Q & = & i = \textbf{old}\ i + 1 \wedge result = max\{\textbf{old}\ result, s.item(i)\} \end{array}$$

and where *max* is extended to sets as in [20]. The proof obligations that must be discharged are as follows.

$$\textbf{not}\ s.empty \rightarrow true$$

i.e., that the precondition of *max_array* establishes the precondition of the initialization.

$$\textbf{not}\ s.empty \wedge i = s.lower \wedge result = s.item(s.lower) \rightarrow I$$

i.e., that the initialization together with the precondition of *max_array* establish the invariant; this is *true*, by one application of the one-point rule.

$$\begin{array}{l} \textbf{not}\ s.empty \wedge I \wedge b \rightarrow (\textbf{not old}\ s.empty \rightarrow \\ \qquad\qquad result = max\{j : INTEGER \mid s.inrange(j) \bullet s.item(j)\}) \end{array}$$

which is also *true*, by simplification.

$$\textbf{not}\ s.empty \wedge I \wedge \neg\ b' \rightarrow true$$

i.e., that the invariant and negated guarded together establish the precondition of the loop body (which is clearly *true*).

$$\textbf{not}\ s.empty \wedge I \wedge \neg\ b' \wedge i' = i + 1 \wedge result' = max\{result, s.item(i')\} \rightarrow I[\_'/\_]$$

i.e., that the body of the loop maintains the invariant. (We should also demonstrate termination, but this is straightforward and omitted.)

The body of the loop, **spec** Q, can be refined as follows

$$\textbf{spec}\ Q \sqsubseteq i := i + 1;\ \ result := result.max(s.item(i))$$

(*max* is a feature of the class *INTEGER*.) The initialization can be implemented as

$$i := s.lower;\quad result := s.item(i)$$

This refinement follows similar steps to a refinement of the same problem in Z.

3.7 BON and the criteria

In Section 2, we suggested six criteria for an industrial-strength formal method. We now very briefly discuss the criteria, and how BON supports – or fails to support them. In Section 4, we discuss extensions to BON which can help eliminate these limitations.

1. *Restrictability.* The use of mathematics in BON specifications is under the control of the software engineer, for both specification and for formal manipulations like refinement. Each feature of each class may be given a contract written in predicate logic, or in an executable subset. The contract may be written as comments. Or the contract may be omitted entirely. Further, refinement need only be applied to those features that the engineer wants. The engineer can therefore restrict use of formality to those parts of the system they believe, in their judgment, require mathematics for specification or refinement.
2. *Gradual introduction.* Because BON has restrictable use of formality, gradual introduction can be obtained by a software engineer carefully managing the extent of use of formality over time, in some process.
3. *Tool support.* BON has CASE tool support [14], and this tool is integrated with an Eiffel compiler and debugger. It does not possess version control tool support (tools like RCS or CVS can be used on the side), nor analysis tools, e.g., theorem provers, simulators, and model checkers.
4. *Modularity.* BON is a pure object-oriented notation, and is therefore modular. It also provides a clustering notation to support modularity at the system level.
5. *Seamlessness.* BON is part of a seamless method that produces Eiffel programs. This method is reversible: BON specifications can be produced from Eiffel.
6. *Education.* BON is not as widely taught as other object-oriented approaches, e.g., UML. It is being taught in some universities throughout the world, and by consultants. Our experience in teaching BON at all levels of an undergraduate programme is that it is appropriate and superior to other methods for introducing and teaching object-oriented concepts. We discuss this more in Section 6 and elsewhere [16].

Meyer [14] suggests that object-oriented software construction, with design by contract – which BON supports – is a gentle progressive exposure to formal techniques that doesn't overwhelm the students and allows them to produce real software. Our experience agrees with this, but we add that the teaching must be founded on an appropriate course in logic and calculation, e.g., supplied by the text [6].

4 Making BON Industrial-Strength

BON does not satisfy all of the criteria listed in Section 3. In particular, it has no tool support for refinement or formal analysis regarding classes and contracts, and it is not yet appropriate for reactive and real-time systems. In this section we outline our plans for extending BON so that it satisfies the properties listed in Section 3.

We see three clear areas for extension and development of BON.

1. *Education:* teaching the notation and method. We discuss this more in the conclusions and elsewhere [16].
2. *Analytic tools:* providing support for proof, discussed in Section 4.1.
3. *Extension to real-time and reactive systems,* discussed in Section 4.2.

4.1 Analytic tools

The current toolset for BON consists of the EiffelCase tool, which is integrated with a collection of base libraries, a compiler/debugger, a GUI builder, and a number of smaller supporting tools. The CASE tool is capable of generating BON diagrams (akin to those in Figs. 1 and 2), of generating Eiffel code automatically from diagrams, and can reverse-engineer diagrams from Eiffel programs. Such tools are constantly under further development. No tools currently exist for formal analysis of BON specifications, e.g., for simulation or proof.

Our current research is focusing on providing analytic tools for BON. We are proceeding by providing links between BON and PVS. BON and PVS will be linked by a general- purpose translator, based on the EiffelLex and EiffelParse libraries and automated tools, which will generate PVS theories from BON specifications. The PVS theories can then be subjected to rigorous analytic manipulation, e.g., involving proof of properties, consistency, and implementability.

The basic mapping from simple BON specifications to PVS is straightforward: a single class is mapped into a PVS theory. Class features can be mapped into PVS variables, boolean functions, or functions on local state variables. An example is given below, for a single class *PASSWORD_MANAGEMENT* and a PVS theory `password`.

class *PASSWORD_MANAGEMENT* **feature**
alarm, *operate* : *BOOLEAN*
$p1$: *PASSWORD*
i : *INTEGER*
$--$ initially $\neg$ *alarm* $\wedge \neg$ *operate* $\wedge\ (i = 0) \wedge (p1 = \ldots)$

verify_user(p : *PASSWORD*)
require $\neg$ *alarm* $\wedge \neg$ *operate*
ensure $(g_1 \rightarrow e_1) \wedge (g_2 \rightarrow e_2) \wedge (g_3 \rightarrow e_3)$
invariant $i \geq 0$
end

where

$$g_1 = \textbf{old}\ i < 6 \wedge \textbf{old}\ p1 = p$$
$$g_2 = \textbf{old}\ i < 6 \wedge \textbf{old}\ p1 \neq p$$
$$g_3 = \textbf{old}\ i \geq 6 \wedge \textbf{old}\ p1 \neq p$$
$$e_1 = (i = 0) \wedge \textit{operate} \wedge \neg\ \textit{alarm} \wedge (p1 = \textbf{old}\ p1)$$
$$e_2 = (i = \textbf{old}\ i + 1) \wedge \neg\ \textit{operate} \wedge \neg\ \textit{alarm} \wedge (p1 = \textbf{old}\ p1)$$
$$e_3 = (i = 0) \wedge \neg\ \textit{operate} \wedge \textit{alarm} \wedge (p1 = \textbf{old}\ p1)$$

Our proposed tool will map this automatically into the following PVS theory, with the conjecture `implementability` added by the user.

```
password : THEORY begin
passwordtype: TYPE
alarm, old_alarm, operate, old_operate: VAR bool
i, old_i: VAR nat
p1,p2,old_p1: VAR passwordtype

verify_user(i,old_i,operate,alarm,p1,old_p1,p2):bool =
   (NOT old_alarm AND NOT old_operate)
   IMPLIES
   ((old_i<6 AND old_p1=p2 IMPLIES
      (i=0) AND operate AND NOT alarm AND p1=old_p1)
    AND
    (old_i<6 AND old_p1 /= p2 IMPLIES
      (i=old_i+1) AND NOT operate AND NOT alarm AND p1=old_p1)
    AND
    (old_i >= 6 AND old_p1 /= p2 IMPLIES
      alarm AND NOT operate AND i=0 AND p1=old_p1))

implementability: CONJECTURE
   (EXISTS i, operate, alarm, p1:
    NOT old_alarm AND NOT old_operate
    IMPLIES
    verify_user(i,old_i,operate,alarm,p1,old_p1,p2))
 end password
```

The PVS specification includes a boolean function, `verify_user`, as well as a user-added conjecture. The conjecture `implementability` is to ensure that the *verify_user* specification is implementable.

In general, a BON specification will involve many classes interacting in client-supplier and inheritance relationships. A simple class-to-theory mapping will not suffice. Here, we intend to map BON classes to PVS types, and to treat objects as PVS variables. Features of BON classes will be transformed into functions on objects. This is akin to the work reported in [10] which maps Java to PVS. There may be advantages to mapping select BON classes to PVS datatypes, but it is not yet clear if this can be

automated, nor if the datatype mechanism is sufficiently expressive. An advantage with mapping BON to PVS is that we do not need to transform BON specifications (and hence Eiffel programs) into an intermediate representation (as is done in [10]), because BON and Eiffel specifications include contracts.

Work is underway, starting January 1999, on a translator from BON to PVS.

4.2 Real-time

Currently, BON does not provide support for reasoning about temporal properties of systems. This makes it difficult to develop real-time, object-oriented programs with BON. Eiffel currently provides support for concurrency [14], so it would be a useful extension to be able to talk about concurrency and real-time properties at the level of BON. We are proposing to extend the BON assertion language to real-time temporal logic [15], and to extend the BON toolset to support writing such specifications.

We also are proposing to support the temporal logic dialect of BON with a combined theorem prover and model checker, namely STeP, so as to provide automated analytic tools for reasoning about such specifications. The integration of the extended BON toolset with STeP will follow the work reported in [12], which considered integrating a visual toolset with STeP for a different (non-object-oriented) notation.

5 Related Work

The development of so-called 'industrial-strength' software engineering methods that have a sound yet practical mathematical basis is of current interest. Powerful general-purpose methods, such as B [1], VDM [11], and those based upon Z have seen success, as have more specialized methods. We now briefly contrast some of these alternatives — particularly Z, UML combined with the Object Constraint Language, VDM and B — with BON. A detailed comparison of BON and UML can be found in [19].

5.1 BON compared with Z and UML

Two widely used approaches are Z and UML; a great deal of past and current research has focused on these notations.

The Z notation is mathematical and, as we have discussed, can be used for refinement. However, Z is not modular (except at the level of an operation) – hence the development of object-oriented and modular dialects of the notation – and therefore provides only minimal support for producing modular – and hence reusable, and maintainable systems. It is left to the discipline of the software engineer to use Z so that maintainable, reusable designs are produced.

UML is the current focus of much activity; it is quickly becoming a 'standard' notation for object-oriented modeling. It has recently been combined with the Object Constraint Language (OCL), to provide better support for formal modeling and design by contract. The Precise UML group has studied formalizing parts of the notation [4]. In terms of the criteria discussed in Section 2, UML (including the OCL) satisfies most of the criteria: restrictability, gradual introduction, tool support, and modularity are clearly

satisfied, although tools for supporting the UML and OCL are needed. In terms of education, suitable textbooks for the UML are beginning to appear, but more are needed. Where UML falls short is with seamlessness: it is difficult to have seamless development with UML and the OCL, because of the auxiliary use of finite state machines for object semantics, data modelling, and stereotypes [19]. Seamlessness for UML is usually supported only for a subset of the language. Further, even though design by contract can be used with UML and OCL, it is not well-supported, as a number of examples in [19] demonstrate.

5.2 BON contrasted with VDM and B

The Vienna Development Method [11] has seen industrial success, primarily in Europe. The method supports specification in a pre- and postcondition style, not unlike BON, using the logic of partial functions. It is supported by tools for proof and refinement. Its fundamental specification component is the operation.

The B method [1] was developed by Abrial and has been applied successfully to substantial software engineering problems. The method uses the modular abstract machine notation for specification. Refinement relationships between machines are used to produce executable programs from specifications. The B Method has tools (the B-Tool and Atelier B) that support semi-automatic generation of code from specifications.

The SPECTRUM project [3, 13] has focused on combining VDM and B. In this method, VDM is to be used for specification and validation, while B is to be used for development of the specification, via refinement and code generation. The integration provides techniques for synthesizing B designs from flat VDM specifications. The integration provides the benefits of both VDM and B: expressive specification, tool support, and modularity. It also provides restrictability of both B and VDM, and therefore the means to gradually introduce one or the other method into practice.

With BON, no integration is necessary to provide expressive specification capabilities or modularity; the method supports these features by default. Further, BON also provides restrictability of the use of mathematics, unlike VDM integrated with B. Finally, BON is object-oriented, whereas B combined with VDM is a modular notation. For the purposes of building reusable, maintainable systems, object-oriented methods can provide significant advantages over methods that are not object-oriented [14].

An advantage that VDM+B has over BON is in tool support; the B-Tool and IFAD VDM are powerful toolsets that support proof and refinement. Similar tools are required for BON, and are currently under development.

6 Conclusions

BON provides a solid foundation for a software engineering method that is practical and has a sound theoretical basis. It provides fundamental techniques, like object-orientation, restrictability of formalism, and tool support, that can be further supplemented by a suitable educational programme and analytic tools.

BON, and other object-oriented methods, are not always appropriate for software development, e.g., systems with complicated real-time constraints, information processing systems, concurrent systems, and operating system kernels. Tool support for

BON, especially with regards to formal analysis, needs to be improved. However, the basic techniques that BON supports — restrictability, object-orientation, and design-by-contract — provide a method for immediate practical use, as well as a foundation for further research and development.

Our experience is that BON is also a suitable method for introducing object-oriented methods to students. We use BON in our third year software design and OO programming courses, and introduce its basic concepts to our first-year students. The notation has proven to be teachable, and the students find that it is easy to quickly apply in non-trivial projects. More details on our experiences and plans for teaching using BON can be found in [16].

References

1. J.-R. Abrial. *The B-Book*, Cambridge, 1996.
2. L. Baresi and M. Pezze. Toward Formalizing Structured Analysis, *ACM Trans. Soft. Eng. Method.* 7(1), January 1998.
3. J. Bicarregui, B. Matthews, and B. Ritchie. Investigating the integration of two formal methods. In *Proc. FMICS '98*, CWI, 1998.
4. A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. *Computer Standards and Interfaces* **19**(7), 1998.
5. M. Fowler. *UML Distilled*, Addison-Wesley, 1997.
6. D. Gries and F. Schneider. *A Logical Approach to Discrete Math*, Springer, 1993.
7. A. Hall. Seven Myths of Formal Methods. *IEEE Software*, September 1990.
8. L. Hatton. Does OO Sync with How We Think? *IEEE Software*, May/June 1998.
9. E.C.R. Hehner. *A Practical Theory of Programming*, Springer-Verlag, 1993.
10. B. Jacobs et al. Reasoning about Java Classes (Preliminary Report). In *Proc. OOPSLA'98*, ACM Press, Oct. 1998.
11. C.B. Jones. *Systematic Software Development using VDM*, Prentice-Hall, 1990.
12. L. Lo. *Modular Design for Reactive Systems*, MSc Thesis, Department of Computer Science, York University, 1998.
13. B. Matthews, B. Ritchie, and J. Bicarregui. Synthesizing structure from flat specifications. In *Proc. B'98*, LNCS 1393, Springer-Verlag, 1998.
14. B. Meyer. *Object-Oriented Software Construction*, Prentice-Hall, 1997.
15. J.S. Ostroff. *Temporal Logic for Real-Time Systems*, Wiley, 1989.
16. J.S. Ostroff and R.F. Paige. Formal Methods in the Classroom: The Logic of Real-Time Software Design. In *Proc. Real-Time Software Engineering Education Workshop'98*, IEEE Press, 1999.
17. R.F. Paige. A Meta-Method for Formal Method Integration. In *Proc. FME'97*, LNCS 1313, Springer, 1997.
18. R.F. Paige. Heterogeneous Notations for Pure Formal Method Integration. *Formal Aspects of Computing* **10**(3):233-242, June 1999.
19. R.F. Paige and J.S. Ostroff. A Comparison of BON and UML. Technical Report CS-1999-03, York University, May 1999.
20. J.M. Spivey. *The Z Notation: A Reference Manual*, Prentice-Hall, 1989.
21. K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Development*, Prentice-Hall, 1995.
22. D. Weber-Wulff. Selling Formal Methods to Industry. In *Proc. FME '93*, LNCS 670, Springer-Verlag, 1993.
23. J. Wordsworth. *Software Development with Z*, Addison-Wesley, 1994.