A Comparison of the Business Object Notation and the Unified Modeling Language

Richard F. Paige and Jonathan S. Ostroff

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

Abstract. Seamlessness, reversibility, and software contracting have been proposed as important techniques to be supported by object-oriented methods. These techniques are used to provide a framework for the comparison of two modeling languages, the Business Object Notation (BON) and the Unified Modeling Language (UML). Elements of the UML and its constraint language that do not support these techniques are discussed. Suggestions for further improvements to both BON and UML are described.

1 Introduction

> *...There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.*
>
> C.A.R. Hoare, *Turing Award Lecture 1980* [7].

As described by Brooks [1], the key factor in producing quality software is specifying, designing and implementing the conceptual construct that underlies the program. This conceptual construct is usually complex and highly changeable. It is abstract but has many different representations. The complexity of the conceptual construct underlying software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence.

A suitable modeling language is needed to describe the conceptual construct, its design and implementation. A satisfactory description of the conceptual construct for an industrial-strength software system prior to its construction is as essential as having a blueprint for a bridge or a large building, before its construction commences.

In 1994, there were between 20 and 50 such languages [14]. Often users had to choose from among many similar modeling languages with minor differences in overall expressive power. But in a landmark meeting in 1994, methodologists and tools producers agreed that users needed a standard. At that moment, the seeds were sown for the UML, and it has since been embraced by leading software developers. From 1997, development of the UML standard – through the Object Management Group – has continued.

This paper conjects that the standardization on UML is premature and perhaps even counter-productive. The reason we use the word ‘conject’ is that a rational critique of

UML would first require building a theory of software quality and then developing metrics for measuring the quality of software developed via a particular approach. Such a theory and consequent metric is not currently available and we must thus resort to a more qualitative, and hence more subjective, analysis.

In the absence of a theory of quality, our starting point will be the remarks quoted at the beginning of this paper. Our main goal will be simplicity of language. But we will need criteria which will allow us to reject a feature of a language as being excessive.

We define quality software as software that is *reliable* and *maintainable*. Reliable software must be *correct:* it must behave according to specification; and, it must be *robust:* it must respond appropriately to input from outside the domain of the specification. Maintainable software must be *extendible:* that is, easy to change with changing requirements; and *reusable:* that is, it can be re-used in different applications.

Reliability is the key quality requirement. If the software does not work correctly and supply the required functionality it is unusable despite having other qualities. Maintainability is the other key requirement because maintenance often accounts for 70% or even more of the cost of the product.

In order to assess UML's contribution to quality, we will compare it to the BON notation and method. The BON approach to quality is to stress three techniques: *design by contract*, as a contribution towards reliability; and *seamlessness* and *reversibility*, as contributions towards maintainability. These terms are defined as follows.

– **Seamlessness.** Seamlessness allows the mapping of abstractions in the problem space to implementations in the solution space without changing notation. Seamless software development occurs by adding new classes, or by enriching already existing classes with additional features.
– **Reversibility.** Changes made during one stage of development can be automatically reflected back to earlier stages. A modification made to an implementation class can be reflected in changes to a BON design class. CASE tools exist to support such reversibility for BON and Eiffel.
– **Design by contract.** The obligations on and benefits of using features of classes are precisely specified, using assertions, with the class. BON was designed to support design by contract, and this coupled with support in programming languages like Eiffel helps to satisfy the seamlessness and reversibility requirements.

With the above definitions now in place, we will look for the simplest set of features that will allow us to describe the conceptual construct underlying our software. The following will be rejected.

– Any feature that militates against contracting, seamlessness or reversibility.
– Any feature that duplicates one already in the notation. (Note that this does not prevent using different *views* of a model, but, as we discuss in the conclusions, these views should ideally be generated automatically from a single model, to maintain consistency.)
– Any feature that is in the notation merely because a competing notation has it.

The language summary for UML (version 1.3) is 161 pages, whereas the summary for BON is just a few pages [10]. Further, BON has only one classifier (the class),

while UML has an additional seven classifiers (e.g., datatype, use case). Among the UML classifiers, a case can be made for redundancy; e.g., datatype and interface can be encompassed by class. We fail to see why all of the UML classifiers are needed. The power of using only the class as a classifier is that it unifies modules (information hiding) with hierarchical subtyping, and thereby abets simplicity and seamlessness.

There are three ways to defeat our arguments.

1. Develop a scientific theory of software quality, and do suitable studies and comparisons to other methods to show the efficacy of UML.
2. Disagree with the notion of software quality, as defined above (although we feel that most developers will want to have reliability and maintainability).
3. Prove UML does at least as good a job as BON at reliability and maintainability.

The rest of the paper will focus on point 3. We hope to show that BON does a significantly better job than UML. We also suggest what changes could be made to UML to better support contracting, seamlessness and reversibility.

Our comparison of BON and UML is founded on the standards for both languages. The BON standard reference is [17]; the UML reference is [16]. We do not consider techniques or extensions beyond the standards. In part, we therefore do not consider UML stereotypes beyond those documented in the standard reference. We refer the reader to [13] for a longer version of this report.

2 Introduction to BON

BON is an object-oriented method possessing a recommended process as well as a graphical and a separate textual notation for modeling object oriented systems. The notation provides mechanisms for modeling inheritance and usage relationships between classes, and has a small collection of techniques for expressing dynamic relationships. The notation also includes an *assertion language*, discussed in more detail in Section 4; the method is predicated on the use of this assertion language. In this sense, BON is based on behavioral modeling. This should be contrasted with UML which is grounded in data modeling. The method is supported by the EiffelCase tool from ISE.

As previously mentioned, BON supports three main techniques: seamlessness, reversibility, and software contracting. As a result, BON provides only a small collection of powerful modeling features that guarantee seamlessness and full reversibility on the static modeling notations.

Early steps of the BON recommended process make use of the informal *chart* (CRC index card) notation for documenting potential classes, clusters of classes, and properties of classes. Intermediate steps rely on the BON static and dynamic modeling notations, which we summarize in following sections. The final step involves mapping a BON model into an OO programming language.

The BON method is not driven by use-cases, unlike UML and its compatible processes. In this sense, we would claim that BON is architecture-centric and contract driven, but not use-case driven. BON does implicitly apply use-cases with its object communication diagrams (they are called ‘scenarios’ therein), but they are not an emphasized part of the method, and are usually applied after design classes have been discovered.

2.1 What is not in BON?

BON is also distinguished by the so-called 'standard' modeling elements that it omits, in particular *data modeling* (e.g., via some variant of entity relationship modeling) and *state machines*. Using these elements breaks seamlessness and reversibility [17]. The modeling advantages that these elements offer are far outweighed by the advantages of seamlessness and reversibility.

Modeling the behavior of objects using finite state machines introduces an impedence mismatch, which requires translation of finite state machines into code or surrender of the class concept. We also lose seamlessness with data modeling, in part because of its reliance on binary associations, and in part because associations as a modeling concept break encapsulation [3, 4, 17]. It is claimed in [17] that using simple OO primitives, and not binary associations, for class relationships is sufficient for specifying all the interesting relationships between classes, and is guaranteed to maintain seamlessness.

We have previously mentioned that BON does not support use-cases directly, though they are implicitly applied during the later stages of the process where object communication diagrams are developed.

3 Seamlessness and Reversibility

In this section, we outline the basic BON modeling language, concentrating on those aspects of the language that support seamlessness and reversibility. As we shall see, all of the static diagram elements of BON are designed for this purpose. These elements will be compared with equivalents in UML, and we will discuss the support these UML elements provide to the aforementioned techniques.

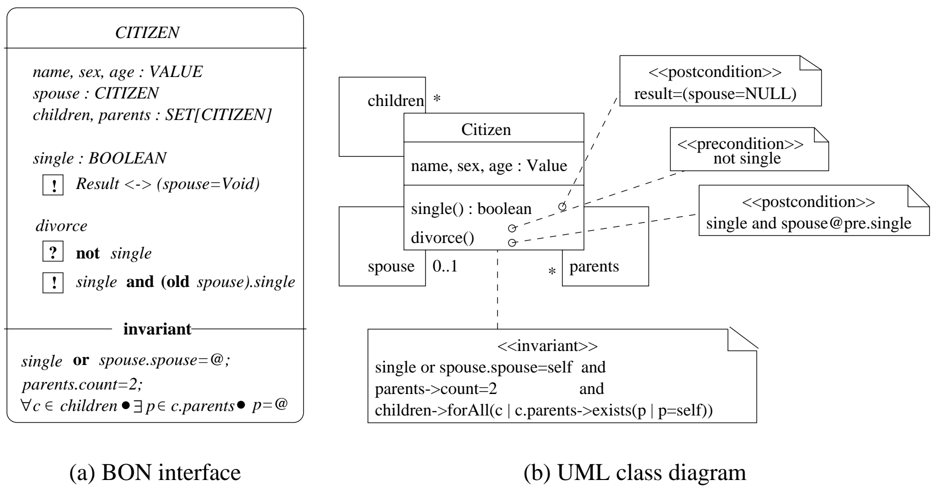
3.1 Class interfaces

The fundamental construct in BON is the *class*; in UML terminology, the class is the only form of classifier available. A BON class is both a module and a type; it is a possibly partial implementation of an abstract data type. With BON, a class is the only way to introduce new types; this is because of the requirement for seamlessness.

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.

Figure 1(a) contains a short example of a BON graphical specification of the interface of a citizen class. Class features, with optional behavioral specifications, are in the middle section of the diagram (there may be an arbitrary number of sections, annotated with visibility tags, as discussed later). An optional class invariant is at the bottom of the diagram. The class 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, the symbol @ refers to the current object; it corresponds to `this` in C++

and Java. Class *CITIZEN* has seven queries and one command. For example, *single* is a *BOOLEAN* query, while *divorce* is a parameterless command. Class *SET* is a generic predefined class with the usual operators (e.g., $\in$, *add*); it is akin to a parameterized class in UML, or a template in C++.



(a) BON interface (b) UML class diagram

Fig.1. A citizen class in (a) BON and (b) UML

A textual dialect of BON also exists [17]. It is syntactically similar to the Eiffel programming language.

A UML class diagram for a citizen is shown in Fig. 1(b). It is drawn assuming that we want to represent all details shown in Fig. 1(a) in UML (OCL is used for writing constraints); later, we discuss how the class diagram can be simplified.

Let us discuss the fundamental differences between the diagrams. First, consider the types of the attributes. In UML, attributes are intended to be used to represent data types (i.e., primitive types and enumerations, perhaps with simple multiplicities). A citizen class thus is not used as a type of an attribute in a class. So, *spouse*, *children*, and *parents* from the BON class interface must be modeled as associations in the UML class diagram, thus making the UML diagram more complicated and making seamlessness difficult to support. In BON, any class may be used in an interface. This leads to simpler models, abets seamlessness, and allows modelers to visually emphasize the most important relationships in their diagrams, thus aiding readability.

A second difference between the UML diagram and the BON diagram is with the behavioral specifications. We shall return to this point in detail in Section 4, but for now we mention that pre- and postconditions and invariants can be modeled using notes, extra boxes, and stereotypes. This clutters the diagram, as Fig. 1(b) shows. For this reason, behavioral details for classes are frequently omitted from diagrams and instead

are presented using a textual assertion language, such as the OCL, separate from the diagram. Separation introduces the potential for maintenance and consistency problems.

3.1.1 Features Each class in BON has a collection of features, which may be queries or commands. All features of an object are accessed by standard dot notation. Identical syntax is therefore used to access attributes, and parameterless queries; this is the so-called *uniform access* principle [10], and is a clear difference between BON and UML. In UML, one must distinguish between using a parameterless function and an attribute by suffixing the former with `()`. This is not necessary in BON, and because of it, it is possible to hide implementation details from clients of the class, and allow the redefinition of functions to attributes under inheritance.

3.1.2 Compressed interfaces Often, specifiers do not want to include all the details of a class interface in a diagram. For this purpose, BON has a *compressed form* for a class. In this form, a class is written as an ellipse containing its name. The compressed form can be annotated with special header information, indicating further properties about the class. Some examples (more are in [13]) are shown in Fig. 2.

Graphical form	Explanation	Graphical form	Explanation
NAME[G,H]	Class is parameterized.	● *NAME*	Class is (potentially) persistent.
* *NAME*	Class is deferred. It has no instances, and is used for classification purposes.	▲ *NAME*	Class is interfaced with the outside world; some feature encapsulates external communication.

Fig.2. Compressed views and headers in BON

The ellipse notation in BON is equivalent to the rectangle in UML. The `*` header in BON, for a *deferred* class, roughly corresponds to an *abstract* class in UML. A deferred class has at least one unimplemented feature. The correspondence between deferred and abstract class is not exact. In UML, classes where all operations are implemented can still be marked as abstract, while this is not possible with BON. Deferred classes are also not the same as UML interfaces, since the former can contain attributes and behavioral specifications, while the latter cannot. Thus, the deferred class notion encompasses the UML notion of interface, and the most common uses of abstract class as well. It is not clear why both a notion of abstract (or deferred) class and interface need to be present in UML.

3.1.3 Visibility Visibility of features in BON is expressed by sectioning the feature part of the class interface, and by use of the feature clause. By default, features are accessible to all client classes that would use them. This is almost the same as public visibility in UML, except that in BON no client class can *change* the value of any query (that is, BON features are externally read-only). This restriction is necessary if we want proper information hiding.

More restrictive visibility of features can be expressed by writing a new section of the class interface and prefixing the section with a *list* of client classes. For example, a section prefix of `feature{A,B}` indicates that only classes `A` and `B` may access the features in the section. This form of visibility is directly implementable in Eiffel, and can be mapped directly to C++ via `friend` features and classes.

This should be contrasted with the mechanism supported by UML, which by default permits the C++/Java style of public, private, and protected features, via tagging each feature with a symbol. Tagging can be applied at both the class and the package level. The BON visibility mechanism is more flexible and general. It is also very helpful in the analysis and design phase, when class communication and coupling is being developed [10, 17].

3.2 Static architecture diagrams

BON provides a small, yet powerful selection of *relationships* that can be used to indicate how classes in a design interact. These relationships work seamlessly and reversibly with those that are supported by modern OO programming languages–especially Eiffel. There are only two ways that classes can interact in BON.

- **Inheritance:** one class inherits behavior from one or more parent classes. Inheritance is the subtyping relationship. It corresponds to generalization in UML: everywhere an instance of a parent class is expected, an instance of a child class can appear. There is only one inheritance relationship in BON; however, the effect of the relationship can be varied by changing the form of the parent classes (e.g., making parents deferred), and by using feature modifiers, e.g., *rename* and *redefine*.
  In BON, renaming mechanisms can be used to resolve multiple inheritance problems. By contrast, UML provides no mechanism for resolving such conflicts. According to [14], it is the responsibility of the designer to resolve class conflicts in multiple inheritance, for example, based on some provided programming language mechanism. This approach increases generality, but breaks seamlessness.
- **Client-supplier:** a client class has a feature that is an instance of a supplier class. There are two basic client-supplier relationships, association and aggregation, which are used to specify the *has-a* or *part-of* relationships between classes, respectively (the difference between the relationships is defined in Section 3.2.1). Both relationships are uni-directed; there is no undirected association in BON. These two relationships correspond to *singly navigable* associations and compositions, respectively, in UML, or to usage dependencies. There is no equivalent to UML's aggregation in BON. Client-supplier relationships can also be bidirectional and self-directed; we provide examples later.

Fig. 3 contains a non-trivial architectural diagram using BON, demonstrating examples of both inheritance and association. Thin vertical arrows (e.g., between *EXP* and *SD*) represent inheritance. Double-line arrows with thick heads (e.g., between *FTS* and *TRANSITION*) represent association. On the associations, names and optionally types of client features that use the supplier class can be specified, e.g., feature *events* on the association between *CLOCKCHART* and *EVENT*. The type of *events* is *generic*; *events* is a set of instances of *EVENT*. The BON naming notation for client-supplier relationships roughly corresponds to the UML notation for roles.

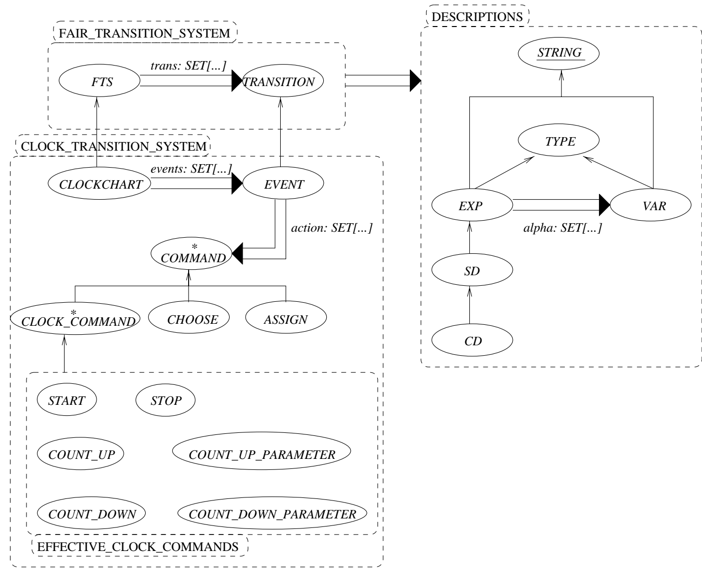


Fig.3. BON architectural diagram for fair transition systems

3.2.1 Client-supplier relationships Client-supplier relationships in BON are between *classes*, and constrain classes. The BON relationships can be mapped directly to attributes or functions in Eiffel and Java, and can be reversibly generated from Eiffel and Java programs.

Associations and aggregations in BON have no object multiplicities; class invariants can be used to express such constraints. In this manner constraint details are kept solely within classes, and thus it is easier to maintain them and to understand their relationships. Multiplicities are just one of many different kinds of constraints that one might want to write on a relationship. Instead of providing a multiplicity notation, BON provides a single, uniform and expressive notation to express all kinds of constraints on relationships.

BON also provides a notion of *aggregation,* which is commonly used to represent the 'part-of' relationship. Aggregation has a precise semantics in BON: it corresponds to the notion of *expanded* type [10]. A variable of expanded type is not a reference. An implication of this is that in BON, aggregates are created and are destroyed with the whole. This most closely corresponds with UML's notion of *composition*.

3.2.2 Clustering In Fig. 3, dashed boxes are *clusters*, which encapsulate subsystems. In BON, clusters are a purely syntactic notion. They can be used to present different views of a system. Clusters roughly correspond to the notion of package in UML, but there are several differences.

The first difference pertains to the extension of BON's relationships to clusters. With BON, inheritance and client-supplier relationships are recursively extended to be applicable to clusters as well as classes, as the figure shows. Precise rules for such ex-

tensions can be found in [17]. The extension for client-supplier relationships is similar in meaning to package dependencies (via the `<<imports>>` stereotype) in UML. A difference arises with inheritance. UML supports generalization between packages, but it differs in meaning from inheritance involving BON clusters. In UML, package generalization defines a substitutability relationship among packages; in BON, it simply means that everything in the child cluster inherits from something in the parent cluster.

The second important difference between clusters and packages is that UML packages introduce import and export facilities. Things inside a UML package cannot see out of the package by default. Further, things outside of a package cannot see inside the package. This can be changed by the specifier by introducing visibility tags on things inside a package. Packages can also explicitly import visible components of other packages, via the `<<imports>>` stereotype. BON supports none of these features; visibility and accessibility is determined and specified by the modeler at the class level. Clusters provide no namespace control, visibility control, and import/export facilities. All of these features are *only* provided at the class level, because of the requirement for seamlessness.

The limitation with the UML approach is that it makes it difficult to express fine-grained visibility for specific features of classes; this is discussed more in [13].

3.3 Dynamic diagrams

BON provides a simple, uniform expressive notation for specifying message passing and object interactions. This dynamic notation presents a complementary view to that of a static model.

We view the BON dynamic notation as useful for producing *rough sketches* of system behavior [8]. Rough sketches provide informal details of how elements in a system interact. There are two categories of dynamic BON notations: the charts, and the object communication diagram. The charts are an informal card-based notation for describing system events and scenarios. The object communication diagram models objects and the messages that are passed between objects. Objects are represented as rectangles enclosing the name of their class, perhaps with an object name qualifier. Messages are depicted as dashed arrows, optionally annotated with sequence numbers representing order of calls. Sequence numbers can be cross-referenced to entries in a *scenario box.*

The object communication diagram corresponds most closely to UML's collaboration diagram (though it is also semantically equivalent to UML's sequence diagram); both forms of diagram share the ideas of sequence numbers and using two dimensions to express collaborations. The BON and UML syntax for these diagrams is so similar that we omit examples, though some can be found in [13]. We point out that BON provides only one diagram for dynamic modeling, whereas UML provides several.

4 Design by contract and assertion languages

We now turn to the second major technique supported by BON (and supportable by UML), namely design by contract [10]. In doing so, we explain how the technique is used in BON and UML, and discuss the respective constraint and assertion languages.

The notion of design by contract is central to BON. Contracts are used to specify the behavior of features, of classes, and of class interactions. Each feature of a class may be given 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 never separated. This substantially aids readability and specification simplicity.

The contract of a feature places obligations on the clients of the feature (they must establish the precondition) and supplies benefits to the clients of the feature (they can rely that the feature will establish the postcondition). Both BON and UML offer constraint languages than can be used to precisely specify behavioral details about classes, features, and entire systems. BON has a simple assertion language based on first-order predicate logic; the method was designed around the use of the assertion language. By contrast, UML has its Object Constraint Language, which was added to UML in version 1.1, after much work on the modeling language had been completed.

4.1 Assertions in BON

Contracts, and thus class behavior in BON, are written in 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 propositional, predicate, and set theoretic operators and constructors, as well arithmetic operators and constants.

The assertion language can also be used to refer to the prestate in the postcondition of a routine. The **old** keyword, applied to any expression *expr*, refers to the value of *expr* before the routine 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, $count = \textbf{old}\ count + 1$ specifies that *count* is increased by one.

A formal semantics for contracts in BON, as well as a collection of re-engineered rules for reasoning about BON contracts, can be found in [12].

4.2 The Object Constraint Language

The Object Constraint Language (OCL) is roughly the equivalent of the BON assertion language in UML; a difference is that the OCL is not based on standard predicate logic. The OCL also fixes problems inherent in the UML metamodel. Requirements for the OCL include: precision; a declarative language; strong typing; and, being easy to write and read by non-mathematicians. As a result, OCL syntax is verbose, replacing common mathematical operators and terms with a more programming language-like syntax. To developers experienced with the use of a constraint language, the OCL will appear cumbersome and difficult to use—especially for reasoning.

4.3 Comparison

While the BON assertion language and OCL are similar in terms of how they are intended to be used, there are significant differences between the two languages.

The first difference is in terms of the rôle that the constraint languages play in the modeling language. The assertion language is fully integrated into BON; the graphical (and textual) notation and the process have been designed to use it. With UML, the constraint language is an add-on, and there are syntactic and semantic issues that remain to be considered after OCL's addition [5], such as connecting finite state modeling with constraints.

The BON assertion language provides both a familiar, concise, expressive mathematical notation – in its graphical form – as well as a textual form that may be preferable to inexperienced constraint language users. The graphical BON assertion language is far superior for reasoning, either with a tool or without, than the OCL; even simple proofs, e.g., the kind needed to show totality or satisfiability of a constraint, will be large and difficult to do with OCL's syntax. An example of using the BON assertion language for reasoning can be found in [12].

Another significant difference is that OCL is a three-valued logic; an *expression* may have the value *Undefined*. BON possesses a notion of *Void*, which reference types may take on. However, this is not the same as OCL's *Undefined*, as only a reference variable (and not, e.g., a *BOOLEAN* variable) can take on value *Void*. Three-valued logics need more extensive rules for reasoning than standard predicate calculus. A case for making the OCL a two-valued logic can be found in [5]. Techniques for reasoning about references can be found in [9].

BON also defines the effect of inheritance on constraints: they are all inherited, and may be refined by the child class. With OCL, this approach is suggested, but not required. It is not clear what value there is in not requiring the inheritance of contracts.

4.4 Contextual information

In BON, constraints are written in class interfaces and are never separated from the interface to which they apply, and therefore maintaining constraints and ensuring their consistency with respect to the attributes and queries of a class is straightforward.

With OCL, it is recommended that constraints not be included in the class diagrams [18], in part because doing so clutters the UML class diagram. Constraints are instead written textually, separate from the diagram. For example, to express that an attribute *age* of a class *Customer* is always at least 18, we would write

$$\underline{\texttt{Customer}}$$
$$\texttt{age} \geq 18$$

Since constraint and diagram are separate, there is increased likelihood of inconsistency, especially without tool support. Even with tool support, separating constraint and class can make it difficult to use existing constraints for further development. Part of the value of using constraints with classes is that we can use existing constraints when writing new ones. This is not easy to do when constraints are not kept in one place.

4.5 Software contracting with OCL

OCL support for software contracting comes in the form of class constraints (which are equivalent to BON's class invariant), and optional pre- and postconditions. These

contracts are not, by default, inherited by a child class, though they may be. Here are two example contracts in OCL. They are translated from the BON class *CITIZEN* in Fig. 1(a). On the left is an OCL contract for function *single*, and on the right is the contract for procedure *divorce*.

$\underline{\texttt{Citizen} :: \texttt{single}()}$	$\underline{\texttt{Citizen} :: \texttt{divorce}()}$
$\texttt{pre} : -- \texttt{none}$	$\texttt{pre} : \texttt{not single}()$
$\texttt{post} : \texttt{result} = (\texttt{spouse} = \texttt{NULL}??)$	$\texttt{post} : \texttt{single}() \texttt{ and spouse@pre.single}()$

In the BON contract for *single*, *spouse* is compared with the *Void* reference; a citizen is single if and only if their *spouse* attribute refers to the *Void* object. [18] makes no reference to a *Void* or *NULL* object that can be used with reference (or object) types. We use *NULL*?? here for illustration, but a careful consideration of object types, *Void* references, and their effect on the type system of OCL and UML, is necessary.

In the postcondition of *divorce*, the value of attribute *spouse* before *divorce* is called is referred to by using of the `@pre` notation. `@pre` can only be applied in postconditions to attributes or associations. This should be contrasted with **old**, which serves a similar purpose in BON. **old** can be applied to any expression. **old** makes specification of certain features very straightforward and convenient. There is no valid technical reason to restrict use of `@pre` to attributes and associations.

4.6 Using the constraint languages

OCL provides a number of built-in types, including basic types like integers, and collection types like bags, sequences, and sets. Methods of collection types (defined in [18]; examples include *collect*, *select*, *forAll* and *exists*) are accessed via the arrow notation $\rightarrow$; methods of basic types are accessed by the standard dot notation. It has been suggested that the arrow notation in OCL is counter-intuitive [2], in part because of its confusion with the pointer dereference syntax of C and implication of logic. A simplifying modification to OCL would be to obey the uniform access principle, and to use dot notation to access both methods and attributes.

The definition of OCL states that collections are flattened [18]; that is, collections cannot contain other collections. Nestings of collections are not permitted because they are considered to be complex to use and explain; however, they are a very useful modeling tool. Further, flattening makes formalization of a theory of collections difficult [5], requires non-standard reasoning about collections, and significantly reduces the modeling power of the notation. We agree with [5] that flattening collections is unnecessary, and it reduces the expressive power of the OCL significantly.

Consider the following illustration of the use of built-in types, taken from [18], that uses the OCL *forAll* operation. Suppose we have a collection (e.g., a set) of customers in a class *LoyaltyProgram* and want to specify that all customers are no more than 70 years old. In OCL, a constraint is

$$\begin{array}{l}\underline{\texttt{LoyaltyProgram}}\\ \texttt{self.customer} \rightarrow \texttt{forAll}(\texttt{c} : \texttt{Customer} \mid \texttt{c.age}() \leq 70)\end{array} \quad (1)$$

This specification is not very readable. It also contains many unnecessary elements: the $\rightarrow$, the empty parentheses, and the type of c. The corresponding BON specification is an invariant of class *LOYALTY_PROGRAM*, which possesses a set attribute *customer*. The constraint is

$$\forall\, c \in customer \bullet c.age \leq 70$$

It is difficult to argue that the OCL constraint (1) is easier to write and read than this BON constraint.

An alternative OCL specification of (1) is given in [18]. The alternative is, in fact, more concise, and is as follows.

$$\frac{\texttt{LoyaltyProgram}}{\texttt{self.customer} \rightarrow \texttt{forAll}(\texttt{age}() \leq 70)} \qquad (2)$$

This is easier to read than (1), but it introduces a new problem. `age()` is an operation of the class `Customer`. The constraint (2) belongs to `LoyaltyProgram`. The use of `age()` in (2) is untargeted; the object to which the call applies is not provided. The OO paradigm clearly states that all operation calls must be targeted, either implicitly to the current object `self` or to a specified object. Neither case applies to the use of `age()` in (2), so we must reject use of such constraints for OO modeling.

See [13] for a discussion of OCL's *allInstances* operation, and how the BON assertion language can be used for everything that it can do. A special *allInstances* feature is unnecessary if a single, expressive assertion language based on standard typed set theory and predicate logic is provided.

5 Limitations of BON and UML

In this section, we briefly discuss some limitations that we have identified, with both BON and UML.

5.1 Improvements to BON

Two inadequacies with BON were identified and discussed in detail in [12]: tool support and handling of real-time systems. There does not exist a wealth of tool support for BON; EiffelCase, a CASE tool from ISE, supports the static diagram and interface notation, as well as round-trip engineering and code generation. There is no analytic tool support, e.g., for reasoning about contracts and classes. Work is underway on providing such support, as detailed in [12]. Better tool support is needed for BON in general.

Currently, BON provides no support for real-time specification (concurrency can be expressed using object communication diagrams). UML, by comparison, has a real-time dialect. A long-term direction of research will be to study how to provide real-time features that integrate with BON's behavioral modeling techniques. This could go hand-in-hand with further study and development of dynamic modeling notations in BON. Any extensions to BON will have to maintain seamlessness and reversibility.

5.2 Improvements to UML

The UML has been constructively criticized by others, e.g., [4, 11, 15]. Our comparison of BON with UML has led us to the following suggestions for improvements with the UML.

- **Design by contract.** Design by contract can be supported in UML through the OCL, but it is not a core part of the modeling language. Full support for design by contract in UML would an excellent way to rationalize existing techniques for specifying constraints, and would significantly improve the UML's capabilities for building reliable, robust software. This, however, may be difficult: the visual modeling notation may require changes in order to better integrate design by contract capabilities, and the semantics, particularly with respect to state diagrams, may have to be changed to accommodate contracts.
- **OCL.** As it currently stands, we believe the OCL is too informal and too verbose for behavioral modeling and for reasoning about said models. A formal semantics for the OCL, as well as a less verbose syntax, needs to be developed. Work is underway along these lines [6]. A number of decisions in the design of OCL are also worth revisiting. As discussed earlier, and elsewhere [5], making the OCL a three-valued logic, and requiring the flattening of collections, are questionable decisions and impact on the modeling power of the notation.
  We question whether it is feasible to develop a constraint language that meets all the requirements placed on the OCL. The goals of precision and non-expert understandability seem to be mutually exclusive. A better approach, as is used in the formal methods application area, might be to use a formal contract language for modeling and specification, and to thereafter paraphrase it into natural language.
- **Rationalization.** With the UML, there are typically several ways to write a model. In part, this is an artifact of unification and the desire to make it as easy as possible for users of the unifying methods to move to UML. With the addition of the OCL, a number of modeling concepts, e.g., or-constraints, subset constraints, etc., can be considered redundant. Further rationalization could be done. Alternatively, restrictions of the UML could be examined, e.g., removing those graphical modeling concepts that become redundant upon addition of a precise constraint language. This is discussed more in [13].

6 Conclusions

BON and UML are languages that can be used to model object-oriented systems. BON is founded in behavioral modeling and emphasizes seamlessness, reversibility, and the use of design by contract. It is simple, easy to teach, and scales up to large systems. UML is a data modeling language that emphasizes use-cases, architectural modeling, and expressiveness. It is supported by a constraint language that is optional for developers to use. It is large, general-purpose, and extensible.

One of our goals in writing this paper was to better understand UML and BON, and to potentially identify limitations and aspects for improvement with each notation. With BON, we have identified limitations with respect to real-time specification and

tool support. With UML, our main conclusion is that its development is not complete. UML has unified three different approaches to modeling; that is a useful first step. A next step for UML development should be rationalization.

A second goal of this paper was to understand how UML supports, or fails to support seamlessness, reversibility, and software contracting. We believe that these are vital techniques for an OO modeling language to support. BON has been designed to support these techniques, but UML has not. If it is desired to use UML and to support the techniques of seamlessness, reversibility, and contracting, we suggest the following.

- **Seamlessness.** We can treat dynamic diagrams as rough sketches [8], and make contracts the fundamental specification element. State diagrams should be used minimally, and ideally as an automatically generated view for a class (e.g., as is done with SOMA [4]).
- **Reversibility.** Navigable associations should be used. Non-standard stereotypes should not be used. Contracting should be considered for use as a technique that further supports reverse-engineering.
- **Contracting.** The OCL should be carefully formalized, and a precise definition of the effect of contracts on inheritance should be specified. Collapsing of collections should not be carried out.

Suppose that the UML was used in this manner. It is still very questionable whether the UML applied in this way is the best approach for developing software seamlessly and reversibly, using design by contract. The most significant difference between BON and UML is that the former satisfies what we term the *single-model principle.* In BON, there is precisely one model for a class. All information associated with the class, e.g., contracts, invariants, signatures, is always kept in that single model. When we design, we add information to the class model, and as necessary we produce different views of the model. But these views are always based on the single model for the class.

UML does not satisfy the single model principle. Information about a class need not be kept in one place; its contracts and invariants are written in OCL, and are not part of the diagram. Information about attributes that are not ‘simple’ is kept outside of the class. The semantics of a class may be given using a state machine. There is no single model for a class written in UML, and this may lead to consistency and communication problems as the class is reused or maintained, and as the system evolves.

References

1. F. Brooks. *The Mythical Man Month*, Addison-Wesley, 1995.
2. D. D’Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.
3. I. Graham, J. Bischof, and B. Henderson-Sellers. Association considered a bad thing. *Journal of Object-oriented Programming* 9(9), February 1997.
4. I. Graham. *Requirements Engineering and Rapid Development*, Addison-Wesley, 1998.
5. A. Hamie, F. Civello, J. Howse, S. Kent, and R. Mitchell. Reflections on the Object Constraint Language. In *Proc. UML’98*, Springer, 1998.
6. A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proc. APSEC’98*, 1998.

7. C.A.R. Hoare. The Emperor's Old Clothes. Turing Award Lecture 1980. *ACM Turing Award Lectures*, ACM Press, 1987.
8. M. Jackson. *Software Requirements and Specifications*, Addison-Wesley, 1995.
9. S. Kent and I. Maung. Quantified Assertions in Eiffel. In *Proc. TOOLS Pacific 1995*, Prentice-Hall, 1995.
10. B. Meyer. *Object-Oriented Software Construction*, Second Edition, Prentice-Hall, 1997.
11. B. Meyer. UML: The Positive Spin. *American Programmer*, March 1997.
12. R.F. Paige and J.S. Ostroff. Developing BON as an Industrial-Strength Formal Method. In *Proc. World Congress on Formal Methods (FM'99)*, Springer-Verlag, September 1999.
13. R.F. Paige and J.S. Ostroff. A Comparison of BON and UML. Technical Report CS-1999-03, York University, www.cs.yorku.ca/techreports/1999/CS-1999-03.html, May 1999.
14. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual,* Addison-Wesley, 1999.
15. A. Simons and I. Graham. 37 Things that Don't Work in Object-Oriented Modeling with UML. In *Proc. ECOOP'98 Workshops*, TU-Munich Report 19813, 1998.
16. *Unified Modeling Language Specification.* Object Management Group, 1998. www.omg.org.
17. K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Development,* Prentice-Hall, 1995.
18. J. Warmer and A. Kleppe. *The Object Constraint Language*, Addison-Wesley, 1999.