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

Richard F. Paige and Jonathan S. Ostroff

Technical Report CS-1999-04

May 6, 1999

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

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, if not essential, techniques to be supported by object-oriented modeling languages and methods. These techniques are used to provide a framework for the comparison of two modeling languages, the Business Object Notation (BON)–which has been designed to support the techniques–and the Unified Modeling Language (UML). Elements of the UML and its constraint language, OCL, 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*.

As described by Brooks [2], 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, invisible, and highly changeable — it consists of interlocking data sets, relationships among data items, algorithms, and invocations of functions. This conceptual construct 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 notation 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 or renovation is as essential as having a blueprint for a bridge or a large building.

In 1995, there were between 20 and 50 such description languages and notations. Often users had to choose from among many similar modeling languages with minor differences in overall expressive power. But in a landmark meeting in Silicon Valley in 1995, methodologists and tools producers agreed that users needed a worldwide standard for metamodeling and notation. At that moment UML was born, and it has been embraced by leading software developers.

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 method or notation. Such a theory and consequent metric is not currently available and we must thus resort to a more qualitative, and hence more subjective, analysis.

When the C++ language first became widespread, a significant segment of the software industry converted to C++ from C, at a substantial cost in re-training. How solidly grounded was this decision to switch to C++? One case study [10], in which a product was developed by the same experienced team, once in C and again in C++, showed that the maintenance cost of the product written in C++ was significantly larger. This is counterintuitive to the standard intuition that OO (at least in C++) should significantly reduce maintenance costs. We may ask the same kind of question about the move to UML.

In the absence of a scientific theory of quality, our starting point will be the remarks of Hoare quoted at the beginning of this paper. Our main goal will be simplicity of method. But we will need a criteria which will allow us to reject a feature of a notation 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 exceptions outside of 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 obviously the key quality requirement. If the software does not work correctly and supply the required functionality it is unusable despite having other qualities such as a fancy GUI or blindingly fast execution. Maintainability is the other key requirement because maintenance often accounts for 70% or even more of the cost of the product.

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

– **Seamlessness.** Seamlessness allows the mapping of abstractions in the problem space to abstractions in the solution space without changing notation. BON was designed to work seamlessly with Eiffel, but it has also been used successfully with a number of other OO languages. In development using BON, systems are specified using classes giving interfaces of abstractions in the problem space; design introduces new solution space classes; and implementation introduces effective classes that implement all the deferred behavior. No translation is required in this process; progress occurs by adding more classes, or defining or implementing class behavior.
– **Reversibility.** Changes made during one stage of development can be automatically reflected back to earlier stages. So a modification made to, e.g., an Eiffel implementation class can be reflected in changes to a BON design class. CASE tools exist to support such reversibility for BON and Eiffel, namely, EiffelCase.

– **Design by contract.** Design by contract is a practical, efficient technique for producing reliable, maintainable, and reusable software. BON was designed to support contracting, 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 concepts that will allow us to describe the conceptual construct underlying our software. The following concepts will be rejected:

– Any concept that militates against contracting, seamlessness or reversibility.
– Any concept that duplicates a concept already in the notation.
– Any concept that is in the notation merely because a competing notation has it.

The notation summary for UML (version 1.3) is 161 pages, whereas the summary for BON is one page; see [16]. 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., datatypes and interfaces can both 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 this abets seamlessness.

There are three ways to defeat our arguments.

1. Develop a scientific theory of software quality, and do suitable studies 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 figure prominently).
3. Prove that UML does at least as good a job as BON/Eiffel 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.

1.1 Organization of the paper

Section 2 introduces BON, and gives an overview of its process and the main techniques that it embodies: seamlessness, reversibility, and design by contract. The focus of Section 3 is seamlessness and reversibility, and therein, we describe the static BON specification notations and how they support these techniques. These BON notations are contrasted with corresponding UML elements. We also touch on BON's small collection of dynamic notations, which we consider as *rough sketches* [14]. Section 4 turns to design by contract, and compares the assertion language of BON with the Object Constraint Language, along with the uses of these languages in several examples. Section 5 briefly touches on tool support, while Section 6 discusses improvements to BON and UML. Finally, in Section 7, we relate our experience in teaching with BON and UML, before concluding in Section 8, with discussion of language design and some general principles that may be useful for enhancing UML and BON.

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 specifying and describing object oriented systems. The notation provides mechanisms for specifying inheritance and client-supplier 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 3; 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 focus in data modeling approaches is to create public data representations that are manipulated by query languages; this has been considered to be against the tenets of information hiding [11].

BON is designed to support three main techniques: seamlessness, reversibility, and software contracting. It also supports the language design principles of *uniqueness*, *simplicity*, and *consistency* [12, 15, 31]. Uniqueness means that there is exactly one way to express each operation or concept of interest in BON; the designers avoid providing more than one means. Further, the notation is simple, based on a small number of powerful, expressive concepts. Finally, BON is consistent: there is a goal to its design—providing a seamless, reversible method for OO development that is founded on design by contract. Each of the concepts present in BON strictly aims at meeting this goal.

As a result of these requirements, BON provides only a small collection of powerful specification features that guarantee seamlessness and full reversibility on the static specification notations. The design of the notation, and the reliance on design by contract, makes the implementation of seamlessness and round-trip engineering straightforward for all static notations, and fully supportable by tools, e.g., EiffelCase [16].

2.1 BON is a method

Unlike UML, BON is a method, and thus possesses a recommended process. The process is representative of many earlier object-oriented methods. The basic steps of the recommended process are outlined as follows.

1. **Delineate the system borders:** identify what the system will include and will ignore. Determine the major subsystems. Prepare a glossary.
2. **List candidate classes.**
3. **Select classes and group:** organize classes into logical groups, and cluster.
4. **Define classes:** provide features and contracts for classes.
5. **Sketch system behavior.** A dynamic model, expressing message passing and scenarios, can be produced.
6. **Define public features:** give final class interfaces, taking into account visibility.
7. **Refine system:** implementation in an OO programming language.

Steps 1, 2, and 3 make use of the informal *chart* notation for documenting potential classes, collections of classes (called *clusters* in BON), and properties of classes. The chart notation is a version of CRC index cards [1], and is supported by a CASE tool [16]. We will not discuss chart notation further here, though we emphasize that it is an important part of the BON method.

Steps 4–6 rely on the BON static and dynamic specification notations, which we summarize in the following subsections. The last step involves mapping a BON specification into an OO programming language, like C++, Java, or Eiffel.

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 design by contract driven, but not use-case driven. BON does implicitly apply use-cases with its object communication diagrams (they are called ‘scenarios’ therein), but it is not an emphasized part of the method, as it is with UML-compatible methods.

2.2 What is not in BON?

BON contains only a small collection of modeling elements that guarantee seamlessness and reversibility, and that are based upon contracting. BON is also distinguished by the so-called standard modeling elements that it omits. BON omits modeling features that break seamlessness and reversibility, in particular, data modeling (e.g., via some variant of entity relationship diagrams) and finite state machines. With both techniques, their advantages are far outweighed by the advantages of seamlessness and reversibility.

It is clear that finite state machines introduce an impedence mismatch which requires translation 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. For this reason, BON includes only simple OO primitives for modeling relationships among classes, thus guaranteeing seamlessness and reversibility. It is claimed in [29] that using simple OO primitives, and not binary associations, for class relationships is sufficient for specifying all the interesting relationships between classes.

3 Seamlessness and Reversibility

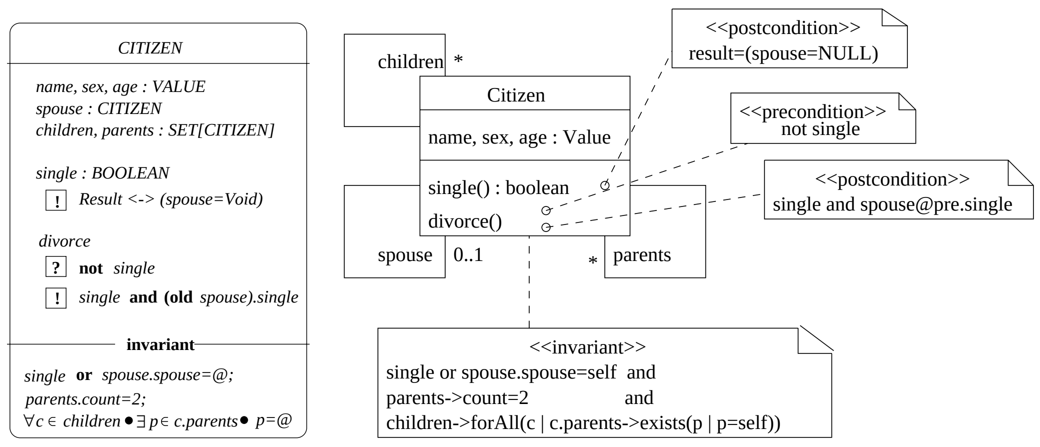
In this section, we outline the basic BON specification language, concentrating on those aspects of the language that support seamlessness and reversibility. As we shall see, all of the static diagramming elements of BON are designed to support seamlessness and reversibility. These elements will be compared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

3.1 Class interfaces

The fundamental specification construct in BON is the *class*; in UML terminology, the class is the only form of classifier available. This should be contrasted with the eight classifiers available in UML. In part, the proliferation of classifiers available in UML is because the designers did not unify the notions of type and module. In BON, a class is both a module and a type. In this sense, it is a possibly partial implementation of an abstract data type (ADT). This means that, when reasoning about BON classes, powerful and mature ADT theory can be used. With BON, a class is the only way to introduce new types.

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 class *CITIZEN*. Class features are in the middle section of the diagram (there may be an arbitrary number of sections, annotated with visibility tags, as discussed in Section 2.3.4). Features may optionally have behavioral specifications, written in the BON assertion language (discussed in Section 3) in a pre- and postcondition form. 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 query (which results in a *BOOLEAN*), while *divorce* is a parameterless command that changes the state of an object. 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

Classes can be self-referential; for example, *CITIZEN* has a query *spouse* of type *CITIZEN*. This can also be indicated using a self-directed client-supplier arrow, as we discuss in Section 2.5. Overloading of feature names is not permitted in BON; each feature must possess a unique name. Feature names may be tagged to indicate that they are *rename*d or *redefine*d; these tags are primarily used when combined with inheritance.

A UML class diagram for a citizen class is quite different; one is shown in Fig. 1(b). This diagram is drawn under the assumption that we want to represent all details shown in the BON interface in a UML class diagram; we discuss how the UML diagram can be simplified later, using standard techniques.

Let us discuss the differences between the diagrams. First, consider the types of the attributes. In UML, attributes correspond to small simple variables (such as integers and booleans). A citizen class thus is not used as a type of an attribute in a class. Thus, *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. We note that in BON any type 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 issue in detail in Section 3, but for now we note that pre- and postconditions of operations in UML class diagrams can be modeled as notes, using the precondition and postcondition stereotypes (and similarly for invariants). This clutters the diagram, as Fig. 1(b) shows. For this reason, behavioral details for classes are frequently omitted from diagrams and alternatively presented using a textual assertion language, such as the OCL, separate from the diagram. This introduces the potential for maintenance and consistency problems, as we discuss in detail later.

There is a minor stylistic difference between BON and UML: in BON, all class names are in italicized upper case; this makes class names easy to spot in specifications. In UML, class names are in roman font, except when they are abstract, where italics are used.

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; no special notation is used to access collections, or reference types. Identical syntax is therefore used to access attributes, and parameterless queries; this is the so-called *uniform access* principle [16], 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 as attributes under inheritance [15].

UML distinguishes between the concept of an *operation* and a *method*. An operation is a service that a class can provide; a method is an implementation of a service. An operation can possess many different methods (e.g., introduced in an inheritance hierarchy). It is usually specified as a name and associated types, e.g., for parameters. BON does not distinguish between operation and method; there are only features, which possess a name, optional parameters and results, and behavioral specifications. A feature may be *redefined* zero or more times in an inheritance hierarchy. In this sense, the BON notion of a feature definition corresponds to the UML notion of a method. We prefer the BON terminology: it corresponds directly to ADT terminology, which is the underlying theory of OO, and it avoids the use of the overloaded term ‘method’.

3.1.2 Compressed interfaces Often, specifiers do not want to include all the details of a class interface in a diagram. Instead, the view they want to present aims at showing the classes and their architectural interactions. 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 are shown in Fig. 2.

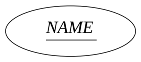
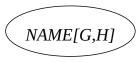
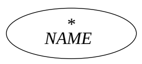
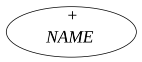
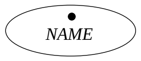
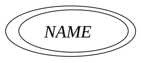
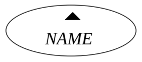
Graphical form	Explanation
	Class is reused from a library.
	Class is parameterized.
	Class is deferred. It has no instances, and is used for classification purposes.
	Class is implementing a deferred class, or reimplementing an ancestor class.
	Class is (potentially) persistent.
	Class is a root; instances may be created as separate processes.
	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 – the latter of which can be indicated by the constraint `{abstract}` or by writing the class name in italics. 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 most common uses of abstract class as well. There do not appear to be UML equivalents to BON's effective classes.

Features of a BON deferred class need not have implementations, though they may have behavioral specifications in the form of contracts, which can be used to constrain any potential implementations and interactions. We discuss contracts in in Section 3.

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 read-only). Visibility of individual features or collections of features 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. For example, a section prefix of `feature{A,B}` indicates that only classes `A` and `B` may access the features in the section.

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 (and directly maps to corresponding mechanisms in Eiffel and C++, the latter via use of `friend` classes). It is also useful for expressing Java's inner class mechanisms, and is very helpful in the design phase, when class communication and coupling is being developed; it is suggested [16, 29] that such a flexible visibility mechanism is a useful specification and design tool, not just an implementation tool. To express such visibility mechanisms in UML would require non-standard stereotypes, which introduce the usual potential for communication problems.

3.1.4 Textual dialect of BON Unlike UML, BON has a textual dialect for specification that is entirely equivalent to its graphical notation. Using the textual notation is convenient for reducing the syntactic gap between specifications and code. The BON *CITIZEN* class in Fig. 1(a) has the textual specification shown in Fig. 3 (readers familiar with Eiffel will note syntactic similarities).

```
class CITIZEN feature
    name, sex, age : VALUE
    spouse : CITIZEN
    children, parents : SET[CITIZEN]

    single : BOOLEAN
      ensure Result<->(spouse=Void) end

    divorce
      require not single
      ensure single and (old spouse).single
    end

    invariant
      single or spouse.spouse=Current;
      parents.count=2;
      for_all c member_of children it_holds
        (exists p member_of c.parents
         it_holds p=Current)
 end -- CITIZEN
```

Fig.3. Textual BON specification for class *CITIZEN*

Mathematical elements like ∀ and ∃ in the graphical class interface are written out in words, e.g., `for_all`, in the textual specification. The textual dialect for BON was designed to serve a purpose similar to that of the OCL, in reducing the difficulty that nonmathematicians may have in using constraint languages. A difference between textual BON and OCL is that standard mathematics is used with textual BON, and therefore so are all standard forms of writing and reasoning about constraints. These are techniques that are taught in all undergraduate CS programmes today. With OCL, students and developers must learn a new syntax, semantics, and reasoning rules.

The textual dialect of BON includes facilities for expressing inheritance (single and multiple), and for renaming and redefining features. See [29] for details.

3.2 Static architecture diagrams

As the design principles of uniqueness and simplicity suggest, 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, but also Java and C++. 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 form of inheritance relationship in BON; however, the effect of the inheritance relationship can be varied by changing the form of the parent classes (e.g., making parents deferred).
  In BON, renaming mechanisms can be used to resolve name clashes and repeated inheritance conflicts that arise when using multiple inheritance. In a child class, features inherited from a parent can be tagged as being renamed. Renaming supports *joining* of features and *replicating* of features. By contrast, UML provides no mechanism for resolving conflicts. According to [23], it is the responsibility of the designer to resolve class conflicts in multiple inheritance, for example, based on some provided programming language mechanism.
– **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. Both relationships are directed; there is no equivalent to the undirected association of UML in BON. These two relationships correspond to *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 be bidirectional, and self-directed; we provide examples later. A third relationship, shared association, is discussed in [29].
  There are only class, and not object, client-supplier relationships in BON, because of the requirement for seamlessness and reversibility. This is discussed more in Section 3.2.1.

Fig. 4 contains a non-trivial architectural diagram using BON, demonstrating examples of both inheritance and association. In the figure, classes are drawn in their compressed form. 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 ('has-a'). 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*; thus, *events* is a feature of *CLOCKCHART*. 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.

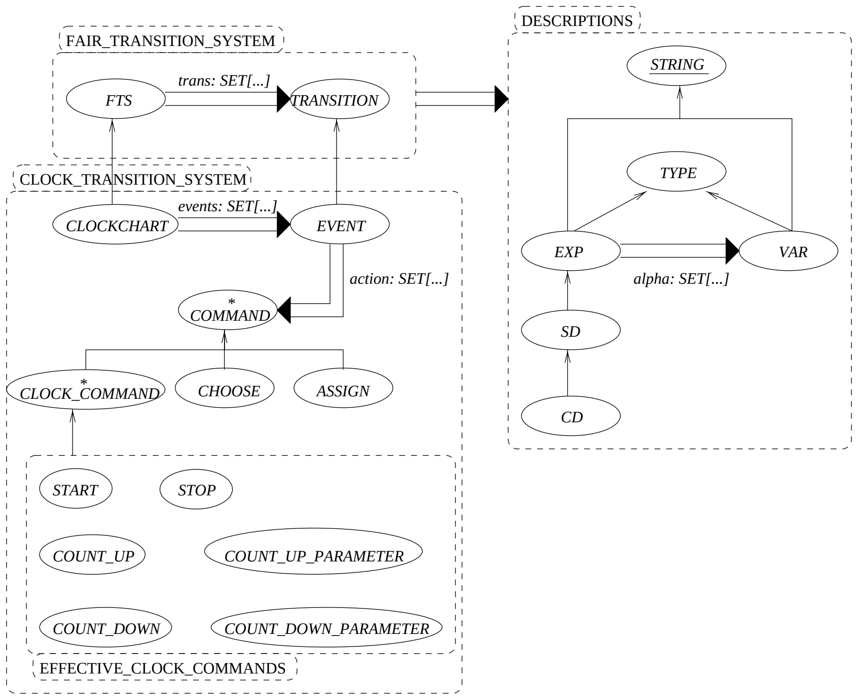


Fig.4. BON architectural diagram for fair transition systems

3.2.1 Client-supplier relationships Client-supplier relationships in BON are between *classes*, and constrain classes. This differs from UML, in which associations, compositions, and aggregations all constrain collections. The BON relationships can be mapped directly to attributes or queries in OO languages like 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. If BON also supported object multiplicities, it would either break the seamlessness of the approach (because there is no simple, general mapping from object multiplicities to programs), or it would provide no further information beyond what we obtain from class invariants. Using class invariants instead also helps to improve the readability of the specification.

Multiplicities in a relationship 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, including multiplicities. We note that with UML, multiplicities and the OCL can be used separately or together to constrain relationships; this redundancy can make it difficult to provide guidance on the use of the modeling language.

Client-supplier relationships in BON may have *class* multiplicities, indicating the *number of relationships* between a client and a supplier (e.g., a client may have two variables of supplier type). Multiplicity is indicated by placing a lozenge containing the number of relationships on the client-supplier arrow. This is not the same as UML's object multiplicity. A similar notation exists for expressing multiple repeated inheritance relationships.

Fig. 4 shows examples of associations. BON also provides a notion of *aggregation*, which is commonly used to represent the 'part-of' relationship. Unlike some notations aggregation has a clear and formal semantics in BON: it corresponds to the notion of *expanded* type, e.g., in Eiffel; that is, aggregations are mapped seamlessly to expanded types. A variable of expanded type is not a reference; thus, memory can be allocated on a run-time stack. An implication of this is that in BON, aggregates are created, exist, and are destroyed with the whole. This most closely corresponds with UML's notion of *composition*.

All client-supplier relationships in BON are directed; unlike UML, there is no undirected association. The reason for eliminating undirected relationships is reversibility; undirected associations would eliminate the possibility of developing a seamless method. The BON developers state that the gains of having undirected associations are outweighed by the gains of a seamless method. Further, the relationships that are expressible via undirected associations are typically not the most interesting to the modeler [29].

3.2.2 Clustering In Fig. 4, 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, e.g., based on user categories, subsystem functionality, license restriction vs. public domain components, etc. 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 extensions can be found in [29], but the basic ideas are simple. An inheritance relationship from a cluster to a class indicates that every element of the cluster inherits from the class. A client-supplier relationship from a class to a cluster means that the class *can* use some element of the cluster.

A similar relationship is defined for package dependencies in UML; the UML `<<imports>>` stereotype is similar in intent to BON's client-supplier relationship between classes. 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 everything 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 (specifically, public, private and protected visibility) on things inside a package. Packages can also explicitly import visible components of other packages, via the `<<import>>` stereotype. BON supports none of these features; visibility and accessibility is determined and specified by the modeler. Clusters provide no namespace control, visibility control, and import/export facilities. All of these features *are* provided at the class level (e.g., through renaming and feature sectioning), because of the requirement for seamlessness.

The limitation with the UML approach is that it makes it difficult to express fine-grained visibility of specific features of classes; we discuss this more below. On the other hand, UML's packages make it straightforward to model facades, i.e., visible and accessible elements of a package, something which can be done only by convention in BON. We also note that not all programming languages support packages in the form present in UML; this means that seamlessness can be lost.

In both UML and BON, the contents of a package or a cluster can be omitted from a diagram if they are not important to the specific view of the specification that we are presenting. CASE tools for both languages exist to support this 'hiding-of-detail'.

In BON, clustering, combined with the previously mentioned visibility mechanism in classes, can significantly simplify modeling. Suppose that we have two clusters, a control and a plant, each of which may contain other clusters and classes. A class *FOO* in the control cluster needs to use a class *BAR* in the plant cluster. To express this in BON, we simply draw a client-supplier relationship from the control to plant cluster; there is no need to indicate the specific classes involved. Then, in the interface of class *BAR* in the plant, we specify visibility controls; that is, there will be a section in *BAR* prefixed with the name of class *FOO*.

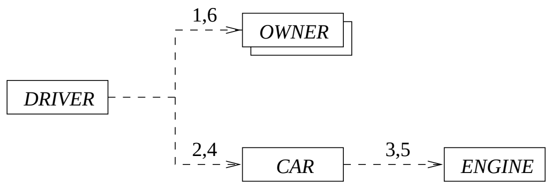
How could this be done with UML? Packages for the control and the plant would be created, and imports and exports would have to be added. But selective export of components of a package, i.e., that only a specific component of this package is accessible to another package, is not easy to express. New stereotypes would have to be introduced, at both the class and package level, to express that, e.g., only class *FOO* can access a particular strict subset of the features of class *BAR*.

3.3 Dynamic diagrams

The BON static diagrams are designed to be simple, expressive, and appropriate for supporting seamlessness and reversibility. BON also provides a simple, uniform expressive notation for specifying message passing and object interactions. This notation presents a complementary view to that of a static model, and is supplemented by a dynamic *chart* notation.

We view the BON dynamic notation as useful for producing *rough sketches* of system behaviour [14]. Rough sketches provide informal, possible vague ideas of how elements in a system interact, e.g., via message passing. They are to be used to provide a basis for later production of feature contracts. However, just because we view dynamic diagrams as rough sketches does not mean that the BON dynamic diagrams cause us to lose seamlessness and reversibility. We retain these capabilities because in the dynamic diagrams, the only message passing primitive is the feature call.

There are two categories of dynamic BON notations: the charts, and the object communication diagram. The charts are an informal CRC card-like notation — typically used early in the dynamic design stage — to describe incoming and outgoing system events (via event charts), communication scenarios (via scenario charts), and creation of instances (via object creation charts). The object communication diagram models objects, and the messages that are passed between objects during a computation; messages are just feature calls. Objects are represented as rectangles enclosing the name of their class, perhaps with an object name qualifier; multiple objects are drawn as overlapping boxes. 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*, which explains semantics. Semantics of calls can be expressed in natural language, or using the assertion language. This is depicted in Fig. 5. Sequence numbers may be nested to an arbitrary depth (e.g., 1.1, 1.1.1, 1.1.2), as in UML.



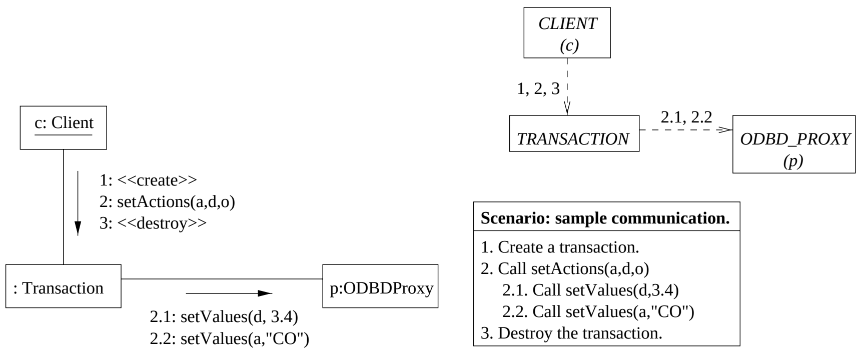
Scenario: borrow car and go for a drive.
1. Driver gets keys from owner. 2. Driver turns ignition. 3. Engine starts. 4. Driver removes key. 5. Engine stops. 6. Driver returns keys to owner.

Fig.5. Object communication diagram with scenario box

Messages in the diagram correspond to calling a feature of an object. Messages are always potential. Bidirectional messages (e.g., showing calls and returns) can be drawn, as can concurrent messages, where a message is sent to many objects simultaneously.

The object communication diagram corresponds most closely to UML's collaboration diagram; both forms of diagram share the ideas of sequence numbers and using two dimensions to express collaborations. Unlike UML, though, the BON diagram does not include stereotypes or function or procedure calls on messages; such details, when necessary, are included in the scenario box. These kinds of details can clutter a diagram if not used judiciously. Further, BON frees the time axis in the diagram; this can result in substantially smaller diagrams than are possible otherwise.

As an example, consider the UML collaboration diagram shown in Fig. 6(a); it is adapted from one in [23], omitting certain stereotype details on links. It could be expressed in BON as shown in Fig. 6(b).



(a) UML collaboration diagram

(b) BON object communication diagram

Fig.6. Examples of dynamic diagrams in UML and BON

Since UML's collaboration diagram is semantically equivalent to its sequence diagram, BON's object communication diagram can be used to express sequence diagram details as well. BON provides only a single diagram for modeling a concept where UML provides more than one.

BON provides an object grouping mechanism akin to the cluster; see [29]. Message relationships are recursively extended to groups, which can substantially simplify dynamic diagrams.

3.3.1 State machines BON does not provide its own dialect of state machines. If designers want to use state machines to, say, specify the reactive behavior of objects, they can use any of the existing dialects, e.g., Harel statecharts, or that used with UML. If contracts are used to specify feature behavior, a state machine can be inferred from the contracts, in much the same way as is provided by the *SOMATIK* tool [7].

In BON, more emphasis is placed on the use of design by contract, object communication diagrams, and the assertion language than on the use of state machines. This is because of the seamlessness and reversibility requirements for the language.

3.4 Further UML diagrams

There are a number of UML diagrams and concepts that have no direct equivalent in BON. Some of these concepts contravene the BON requirements for seamlessness and reversibility. Stereotypes, in particular, are not a part of BON for this reason, as are use-case diagrams. Components and component diagrams, as well as deployment diagrams, are not part of BON as well.

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 [16]. In doing so, we explain how the technique is used in BON and UML, and discuss the respective contracting or assertion languages.

The notion of design by contract is central to BON. It is 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 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). 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 standard first-order predicate logic; the method was designed around the use of the assertion language, via design by contract. 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 and arithmetic operators and constants. Expressions constructed from these simple elements are easy to immediately translate into executable code.

The BON predicate logic can be used when more expressive power is needed; the logic is sufficiently expressive to model any computation. The predicate logic introduces basic set operations (e.g., membership, union, et cetera), and universal and existential quantifiers. The basic assertion language elements in BON are given in Table 1, below. We show elements in both the graphical and the textual syntax. The graphical syntax is similar to that used in any recent logic or discrete maths textbook.

The BON assertion language can also be used to refer to a prestate in the postcondition of a routine. The **old** keyword, applied to an expression *expr*, refers to the value of

Assertion Elements		
Graphical BON	Textual BON	Explanation
Result	`Result`	query result
@	`Current`	current object
$+, -, /, *$		numeric operators
$=, \neq$	`=, /=`	equal, not equal
$\rightarrow$	`->`	semi-strict implies
and, or		logical and, or
$\neg$, $\leftrightarrow$	`not, <->`	not, equivalence
$\exists$	`exists`	there exists
$\forall$	`for_all`	for all
$\mid$	`such_that`	such that
$\bullet$	`it_holds`	it holds
$\in$	`member_of`	in set

Table 1. Examples of BON assertion elements

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* $=$ **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 [21].

4.2 The Object Constraint Language

The Object Constraint Language (OCL) is roughly the equivalent of the BON assertion language in UML. It can be used to write pre- and postconditions of methods and class invariants. Requirements for the OCL include: precision; a declarative language; strong typing; and, being easy to write and read by people who are not mathematicians or computer scientists. As a result, OCL syntax is verbose, replacing common mathematical operators and terms with a more programming language-like syntax. Such a syntax is proposed for being more attractive to developers unfamiliar with basic logic and set theory. However, to developers experienced with the use of a constraint language in software design, the OCL will appear cumbersome and difficult to use—especially for reasoning.

We–and others [7, 22]–question whether it is feasible to produce a usable constraint language that satisfies all the goals of the OCL. This is discussed more in Section 5.

4.3 Comparison

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

The first difference is really one in terms of the rôle or emphasis 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 with use of the assertion language in mind. With UML, the constraint language is an add-on, and there are syntactic and semantic issues that remain to be considered with the OCL's addition [8].

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. We suggest that 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 complex to do with OCL's syntax. An example of using the BON assertion language for reasoning can be found in [21].

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, or an expression) can take on value *Void*. Three-valued logics need more extensive rules for reasoning than standard predicate calculus. A full comparison of two- and three-valued logics is beyond the scope of this paper, but the interested reader can find a discussion in [3]. A case for making the OCL a two-valued logic can be found in [8].

BON defines the effect of inheritance on constraints: they are all inherited by the child class (where the child class may refine them in a formally defined way). With OCL, this approach is recommended, but not required. It is not clear what value there is in not requiring the inheritance of contracts.

The BON assertion language is not formal, in the sense of notations like Z [25], in that it has a precisely specified formal semantics, e.g., as a set of axioms. However, the language is based on typed set theory, and so producing a formal semantics will not be difficult; a formal semantics for contracts (in a sequential setting) can be found in [21]. The OCL is also not formal; work on providing it with a formal semantics has begun [9]. This work has already identified some limitations with the OCL, and some complications in terms of formalization, especially with respect to flattening of collections and the three-valued nature of OCL's logic.

We now discuss more specific features of BON and UML, with regards to the constraint languages and their use in modeling.

4.3.1 Contextual information In BON, constraints (preconditions, postconditions, and invariants) are written in class interfaces; they 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 [30], in part because doing so clutters the UML class diagram. Constraints are instead written textually, separate from the diagram, perhaps elsewhere in a specification document. For example, to express that an attribute *age* of a class *Customer* is always at least 18, we would write

$$\begin{array}{l}\underline{\mathtt{Customer}}\\ \mathtt{age} \geq \mathtt{18}\end{array}$$

Because constraint and class are written separately, it is necessary to indicate the class to which a constraint applies; the context is underlined and prefixes the constraint. Since constraint and diagram are separated, there is increased likelihood of inconsistency, especially without suitable tool support. Even with tool support, separating constraint and class can make it difficult for developers to use existing constraints to further develop the class, simply because the developers lose their focus. Part of the value of using constraints with classes is that when writing new constraints, we can use existing constraints, through functions of a class. This is not easy to do when constraints are not kept in one place.

4.4 Support for design by contract

The BON assertion language is used to specify pre- and postconditions of features as well as class invariants. Such behavioral specifications are inherited by child classes, and may be further refined (e.g., a precondition may be replaced by a weaker precondition in a child class). BON provides the **old** keyword, usable in postconditions, for referring to the value of an expression in a routine prestate.

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 of contracts in OCL. They are taken from the BON class *CITIZEN* in Fig. 1(a). First, we show a possible contract, in OCL, for the parameterless function *single*.

$$\begin{array}{l}\underline{\mathtt{Citizen} :: \mathtt{single}()}\\ \mathtt{pre} : -- \mathtt{none}\\ \mathtt{post} : \mathtt{result} = (\mathtt{spouse} = \mathtt{NULL})\end{array}$$

In the equivalent 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. [30] 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.

And the contract for the parameterless procedure, *divorce*.

$$\begin{array}{l}\underline{\mathtt{Citizen} :: \mathtt{divorce}()}\\ \mathtt{pre} : \mathtt{not\ single}()\\ \mathtt{post} : \mathtt{single}()\ \mathtt{and\ spouse@pre.single}()\end{array}$$

In the postcondition of *divorce*, the value of attribute *spouse* before *divorce* is called is referred to, by using of the `@pre` notation[1]. For any attribute *x*, `x@pre` is the value of

[1] `@pre` is evidently derived from the Z primed notation [25].

x in the prestate of a method. `@pre` can only be applied to attributes or associations (i.e., collections of some kind). This should be contrasted with **old**, which serves a similar purpose in BON, though therein **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, except to simplify the notation.

4.5 Language built-in types and operations

BON provides a library of built-in classes, such as *INTEGER*, *REAL*, *STRING*, and a number of generic container and traversal classes, e.g., $SET[G]$, $ARRAY[G]$, and $BAG[G]$. All class features are accessed in the same way, using the dot notation, whether the class is a collection or not.

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 [30]) 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 [6], and difficult to teach, in part because of its confusion with the similar pointer dereference syntax of C, and implication of logic. A simplifying consistency modification to OCL would be to obey the uniform access principle, and to use dot notation to access methods and attributes.

The definition of OCL states that collections are flattened [30]; 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 [8], can require non-standard reasoning about collections, and significantly reduces the modeling power of the notation. We agree with [8] that flattening collections is unnecessary, and it reduces the value of the OCL significantly.

OCL offers a selection of features on collections, such as *collect* (which produces a new collection based on computed values from an old collection), *select* (which chooses values from a collection), *forAll* (which specifies that a boolean expression must hold for all values), and *exists* [30]. Others are suggested in [8]. BON instead provides a general, and standard, set comprehension notation that encompasses all the OCL notations, and is, in our opinion, less prone to confusion and ambiguity, and easier to teach and provide methodological guidance as to its use. We note that there is redundancy with OCL's features: *collect* can be used to specify the other features. While this redundancy is not necessarily inappropriate in itself, better justification for including *collect* as well as the other features should be provided.

Consider an example, taken from [30], 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 specification is

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

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

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

It is difficult to argue that the OCL specification (1) is easier to write and read than the corresponding graphical BON specification, and even the textual BON specification:

```
for_all c member_of customer it_holds c.age<=70
```

In the example, the mathematical symbols involved in constraining the set are simple; overuse of such symbols may result in large, hard-to-read specifications. However, the OO setting can prevent overuse: therein, the constraints are used to write contracts and clauses of class invariants. Very often, such behavioral specifications are short and simple (and if they are not, very frequently the specifier is doing something wrong).

An alternative OCL specification of (1) is given in [30]. The alternative is, in fact, more concise, and is as follows.

$$\begin{array}{l}\underline{\texttt{LoyaltyProgram}}\\ \texttt{self.customer} \rightarrow \texttt{forAll}(\texttt{age}() \leq 70)\end{array} \tag{2}$$

This is clearly easier to read, 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 (2), so we must reject use of such constraints for OO modeling.

UML also provides the *allInstances* operation for producing the set of all instances of a modeling element as well as its subtypes. Here is an example, from [30].

$$\begin{array}{l}\underline{\texttt{Person}}\\ \texttt{Person.allInstances} \rightarrow \texttt{forAll}(\texttt{p} \mid \texttt{p.parents} \rightarrow \texttt{size} \leq 2)\end{array}$$

The constraint says that a person has no more than two parents. Note that the constraint is applied to the class *Person* directly. In BON, there is no equivalent to `allInstances`, and so the only alternative for expressing such a constraint is to constrain the invariant of class *Person*. In BON, the above constraint would be written as

$$\mathit{parents}.\mathit{size} \leq 2$$

This latter example of a constraint can also be written in OCL, and in fact is the recommended approach in [30]. [30] also discourages use of *allInstances* in this manner. *allInstances* is clearly unnecessary for writing constraints like the preceding, but it has been suggested as useful for specifying creation procedures [8]. In OCL, we can specify the creation of a new object (say, of type *Student*, being added to an existing set of students in a *Course* object) as follows. (For simplicity, we omit arguments that can be used to set attributes of the student being created.)

$$\begin{array}{l}\underline{\texttt{Course} :: \texttt{add\_new\_student}()}\\ \texttt{self.students} \rightarrow \texttt{exists}(\texttt{s} : \texttt{Student} \mid \\ \quad \texttt{Student.allInstances} - \texttt{Student.allInstances@pre} \rightarrow \texttt{includes}(\texttt{s}))\end{array}$$

We specify that *students* includes a new student who wasn't previously in the set of all known instances of type *Student*. From this example, it is implied that *allInstances* gives the set of all instances of a type at some point in time; this is confirmed in [8].

This notation is not convenient; Catalysis [6] introduces a `new` notation for object introduction. A similar notation could be added to OCL.

Expressions similar to *allInstances* can be written in BON. Suppose class *COURSE* has a set *students* and a feature *add_new_student*. This feature's postcondition will be

$$\exists\, s \in students \bullet s \notin \textbf{old}\ \{t : STUDENT\}$$

The propositional part of the postcondition says that *s*, now an element of *students*, wasn't in the set of all known *STUDENT*s before the operation was called.

A special *allInstances* feature is unnecessary if a single, expressive assertion language based on standard typed set theory and predicate logic is provided.

4.6 Contracts for resolving ambiguity

OCL was added to UML to provide a more precise basis to the modeling language. It also provides designers with a mechanism with which to resolve model ambiguities. An excellent example of the latter, from [30], arises with the or-constraint. Consider Fig. 7. Two interpretations are possible: one person has either a managed project or a performed project, but not both; and, one project has either a project leader or a project member, but not both.

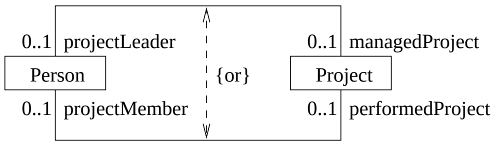


Fig.7. Ambiguous or-constraint

To precisely express which interpretation is desired, a constraint on either class *Person* or *Project* can be added; see [30] for examples. Adding an OCL constraint, in fact, removes the requirement for the UML or-element; the meaning of the association is captured unambiguously by the constraint.

This is exactly the approach taken to specifying such a model in BON. No new features are added to the language in order to model or-constraints; the constraint language is used to express the desired restrictions. Fig. 8 shows a BON diagram, with the first possible interpretation. The lozenge between *PERSON* and *PROJECT* indicates that there are two bidirectional relationships between the classes; the names of the features involved in the relationships are specified on the relationships.

While we can write a BON model similar to the UML one, we question whether object-oriented designs should be written in the style of Fig. 7 in the first place. A

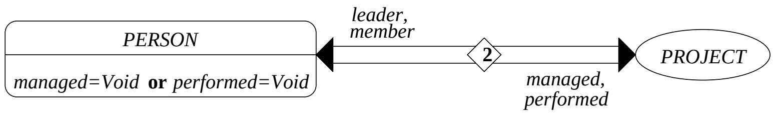


Fig.8. BON specification of or-constraint

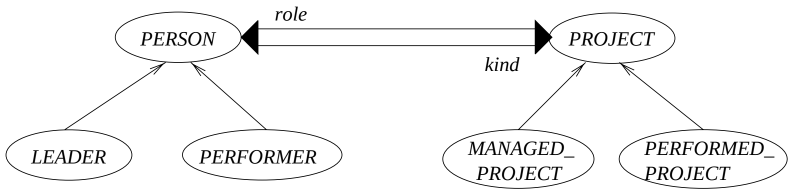


Fig.9. Inheritance solution to design of Fig. 8

more maintainable and understandable design for a similar system would make use of inheritance and polymorphism, as shown in Fig. 9.

Graham [7] makes observations similar to our own. Constraints should be added to subclasses of *PERSON* and *PROJECT* to restrict acceptable types of the features *role* and *kind*.

A precise, expressive, assertion language can dramatically simplify the requirements for a modeling language. It also makes it easier to give advice to users of the language on how to deal with various modeling problems. It can improve the readability of specifications, make specifications simpler, and can simplify the modeling language itself.

4.7 Accessing the meta-level

OCL provides facilities to access the meta-level of a model. This allows, for example, modelers to interrogate an object to determine its type. The operation *oclIsTypeOf* is used for this. OCL provides these operations, which are applicable to every type of OCL instance, as features of a class *OclAny*. Operations provided with this class also include *oclIsKindOf*, which is a boolean function used to check type conformance, and a number of equality and comparison operators.

OclAny is a supertype of all types in a model. The BON equivalent is called *ANY*, at the top of the BON classification lattice. It supplies equality, type conformance, and comparison operations, like *OclAny*. In theory, BON also has a *bottom* to its classification lattice: *NONE*, which inherits from every class. *NONE* has no instances.

Consider the example on page 52 of [30]. Child classes *ApplePie* and *PeacePie* inherit from *FruitPie*. An invariant on *ApplePie* is that its ingredients are only apples.

```
ApplePie
self.ingredient → forAll(oclIsKindOf(Apple))
```

The constraint says all elements of the ingredient collection are of type *Apple*. In BON, we would add an expression to the invariant of class *APPLE_PIE* as follows.

$$\forall\, i \in ingredient \bullet i : Apple$$

The colon operator ':' is used in BON constraints for type interrogation; it is a feature of class *ANY*. [29] contains a number of examples of its use.

5 Tool Support

Currently, there is a wealth of UML-compliant tools, e.g., Rose, COOL:Jex/ObjecTeam, and EiffelCase, to assist developers in producing UML models and in generating code automatically from the models. There is less tool support for BON. BON is primarily supported by EiffelCase, from ISE. EiffelCase supports production and browsing of static BON 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 subsets of specifications. As of revision 4.3 (released February 1999), EiffelCase also supports pseudo-UML.

UML currently has a significant advantage over BON in terms of tool support.

6 Limitations of BON and UML

In this section, we briefly discuss some limitations that we have identified, with both BON and UML. Our observations are based on those that arose during development of this paper, through extensive use of BON, and through teaching of both BON and UML.

6.1 Improvements to BON

Two inadequacies with BON were identified and discussed in detail in [21]: tool support and handling of real-time. 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 [21]. Better tool support is needed for BON in general, if it is to be considered industrial-strength.

Currently, BON provides no support for real-time specification (concurrency, of a certain degree, can be expressed using object communication diagrams). UML, by comparison, has real-time features, e.g., as detailed in [5]. A long-term direction of research will be to study how, and if it is valuable, 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. Currently, BON provides only object communication diagrams. [29] suggests that statecharts can be used in cooperation with BON as well. Consideration of other dynamic notations, e.g., those in UML or [4], as well as physical modeling notations, such as, deployment diagrams, to see if they have a role to play in BON, will be worthwhile as well. Any extensions to BON will have to maintain seamlessness and reversibility.

Unlike UML, BON does not currently have a precise diagrammatic meta-model; it does have a context-free grammar for its textual dialect (see [29]). Given the simplicity of BON and its grammar, producing a precise meta-model should be straightforward. Though the UML does possess a meta-model, it has been criticized for its lack of clarity and for omission of concepts. See [11], in particular, for a comparison of UML's meta-model with that of OPEN.

Finally, the BON assertion language can be criticized as being complex and inaccessible to non-domain experts. A semiformal natural language extension – akin in premise to OCL, though with a different syntax – might be worthwhile, when the method is to be used by non-experts. The textual BON assertion language would be a good starting point for the development of such a notation.

6.2 Improvements to UML

The UML has been constructively criticized by many others, e.g., [7, 17, 24]. 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. Further, tool support for UML would have to be augmented to fully support design by contract.
– **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, e.g., see [9] for a Larch LSL semantics for OCL. The interested reader might consult [20], which presents an integration of UML with a formal design calculus that can be used to write constraints and to reason about the constraints in a rigorous manner.
  A number of decisions in the design of OCL are also worth revisiting. As discussed earlier, and elsewhere [8], 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. Furthermore, there are modeling concepts in the OCL that are questionable value (e.g., *allInstances*) and there is redundancy; rationalization would therefore be useful.
  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 commonly

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; UML does not satisfy the principle of uniqueness, discussed earlier. 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 to be done in order to make the UML manageable and more teachable. 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 the conclusions.

7 Teaching with BON and UML

We have been teaching with BON at all levels of our undergraduate and graduate curriculum for two years now, and in this section we briefly discuss our rationales for using BON, and how we introduce students to UML and other modeling languages and methods as well.

7.1 How do we use BON?

We introduce BON, in a very simple form, to students in our first year CS1 and CS2 courses. Therein, BON is simply used to specify class interfaces, and to draw client-supplier relationships and inheritance relationships between classes. We do not use the full BON assertion language, nor dynamic diagrams. Contracts for methods are specified as simple propositional pre- and postconditions, which can be implemented directly in any programming language. The lab manual for these courses is supplemented by a short introduction to BON, prepared by one of the authors. These two courses use Java, and so the introduction contains a simple guide to mapping the subset of BON constructs that are used into Java code.

The use of contracts – even simple propositional ones, as used in our first-year courses – is emphasized and reinforced throughout the undergraduate curriculum. The value of contracts in terms of abetting debugging, testing, and coding, is emphasized in all programming-oriented courses.

We have two third-year undergraduate courses, one on object-oriented programming and another on software design. BON is introduced, at a high-level, in the first course, and is taught in detail in the second course. In the second course, BON is applied in a small specification assignment, and then in a large-scale group project. In the project, a BON specification is produced (typically, the specification is in the range of 15-25 pages). Then, in following project phases, the specification is implemented in an OO programming language. CASE tools are used to develop the specification, and to produce code templates for implementation.

We have also used BON in our advanced graduate-level software engineering course. Therein, BON is taught and students are required to apply it in requirements analysis and specification of a substantial software system.

7.2 Why did we choose BON?

Our choice of BON as a teaching method was influenced by our desire to educate students in the seamless production of reliable, robust, maintainable software systems. BON, through its design by contract and OO technologies, meets this need. Further, we also wanted to give students experience with state-of-the-art and industrial-strength CASE tools, compilers, and debuggers. Tools for BON also fit this need. Finally, we wanted to teach a method that would not get in the way of students applying software design principles. In our experience, and in the experience of others, complex methods, notations, and tools can prevent the students from understanding how to apply the design principles, e.g., related to class communication, taught in lecture. With BON, the notation and method do not get in the way of the students applying the techniques for producing reliable and maintainable software that are taught in lecture.

Another reason for teaching BON is that students can quickly and effectively start to apply it to non-trivial problems. We typically spend at most one fifty minute lecture teaching the syntax, semantics, and process of BON. We then spend a number of lectures on CASE studies illustrating BON's use, and on in-class examples where BON is used in the specification and design of systems. It is not surprising to us to see that after a single fifty minute lecture, the students are capable of applying BON to substantial problems. Because of this, and because of the minimal amount of time spent on teaching the method, we can give the students more complex and realistic projects in the course, and can devote more in-class time to discussing these projects, their design problems, and possible solutions. We doubt that we would be able to devote so much in-class time, nor could give such realistic projects to students were we to use an alternate method.

7.3 Teaching other methods

We emphasize the use of BON in our software design course, but we are careful to spend two weeks of the thirteen week term talking about other methods. In particular, we spend a week or so discussing UML and compatible processes, and another week on non-OO methods, including structured techniques and formal methods. We typically revisit problems that we attacked using BON, and produce UML models for comparison.

We find that after having spend ten or so weeks learning and applying BON to realistic problems, students have little difficulty learning the basic concepts of a language like UML.

8 Conclusions

BON and UML are languages that can be used to model object-oriented systems. BON, a graphical notation originally designed as a front-end for Eiffel, is founded in behavioral modeling and emphasizes seamlessness, reversibility, and the use of design by contract. It features a small collection of modeling concepts and diagrams, as well as an expressive assertion language. 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 motivations 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 clearly 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, to eliminate inconsistencies and overlap.

A second goal of this paper was to understand how UML supports, or fails to support seamlessness, reversibility, and software contracting. We, and others, 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. Table 2 summarizes key elements of UML that prevent full application of these techniques.

Technique	**UML Element**
Seamlessness	state diagrams,
	activity diagrams,
	interaction diagrams,
	non-standard stereotypes
Reversibility	undirected data modeling,
	multiplicities,
	non-standard stereotypes
Contracting	OCL and inheritance,
	collapsed collections

Table 2. Techniques and UML elements that hinder their support

If it is desired to use UML and to support the techniques of seamlessness, reversibility, and software contracting, we suggest the following.

– **Seamlessness.** We should treat dynamic diagrams as rough sketches [14], 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 [7]). Formal semantics should be provided for standard stereotypes, and non-standard stereotypes should not be used.
– **Reversibility.** Navigable associations should be used in class diagrams. 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.

Even with the UML used in this way, it is questionable whether it is the best approach for developing software via contracting seamlessly and reversibly. 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. this does not preclude multiple several models for a *system*.

UML does not satisfy the single model principle. Information about a class is not 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. And 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. K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. *ACM Sigplan Notices* 24(10), October 1989.
2. F. Brooks. *The Mythical Man Month*, Addison-Wesley, 1995.
3. J.H. Cheng and C.B. Jones. On the usability of logics which handle partial functions. *Proc. 3rd Refinement Workshop*, Springer-Verlag, 1991.
4. S. Cook and J. Daniels. *Designing Object Systems*, Prentice-Hall, 1994.
5. B. Douglass. *Real-Time UML*, Addison-Wesley, 1998.
6. D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.
7. I. Graham. *Requirements Engineering and Rapid Development*, Addison-Wesley, 1998.
8. A. Hamie, F. Civello, J. Howse, S. Kent, and R. Mitchell. Reflections on the Object Constraint Language. In *Proc. UML'98*, Springer, 1998.
9. A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proc. APSEC'98*, 1998.
10. L. Hatton. Does OO Synch with How We Think? *IEEE Software* 15(3), May/June 1998.
11. B. Henderson-Sellers and D. Firesmith. Comparing OPEN and UML: the two third-generation OO development approaches. *Information and Software Technology* **41**(3), 139-156, Feb. 1999.
12. C.A.R. Hoare. Hints on Programming Language Design. In *Proc. ACM Principles of Programming Languages 1973*, ACM Press, 1973.
13. C.A.R. Hoare. The Emperor's Old Clothes. Turing Award Lecture 1980. *ACM Turing Award Lectures*, ACM Press, 1987.
14. M. Jackson. *Software Requirements and Specifications*, Addison-Wesley, 1995.
15. B. Meyer. *Eiffel: The Language*, Prentice-Hall, 1992.
16. B. Meyer. *Object-Oriented Software Construction*, Second Edition, Prentice-Hall, 1997.
17. B. Meyer. UML: The Positive Spin. *American Programmer*, March 1997.
18. G.A. Miller. The magical number seven, plus or minus two. *Psychological Review*, 1956.

19. R.F. Paige. When are methods complementary? *Information and Software Technology* **41**(3), 157-162, Feb. 1999.
20. R.F. Paige. Integrating a Program Design Calculus with a Subset of UML. To appear in *The Computer Journal*, 1999.
21. R.F. Paige and J.S. Ostroff. Developing BON as an Industrial-Strength Formal Method. Submitted. 1999.
22. R. Pooley and P. Stevens. *Using UML,* Prentice-Hall, 1999.
23. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual,* Addison-Wesley, 1999.
24. A. Simons and I. Graham. 37 Things that Don't Work in Object-Oriented Modeling with UML. In *Proc. ECOOP'98 Workshop on Precise Behavioral Semantics*, TU-Munich Report 19813, 1998.
25. J.M. Spivey. *The Z Notation: A Reference Manual*, Prentice-Hall, 1989.
26. G.L. Steele. Growing a Language. In *Proc. OOPSLA'98,* ACM Press, 1998.
27. K. Surendram and B. Henderson-Sellers. OO Modeling Tools for Analysis and Design. In *Proc. Software Engineering Education and Practice '98,* IEEE Press, 1998.
28. H. Thimbleby. A critique of Java. Available at www.cs.mdx.ac.uk/harold/papers/javaspae.html
29. K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Development,* Prentice-Hall, 1995.
30. J. Warmer and A. Kleppe. *The Object Constraint Language,* Addison-Wesley, 1999.
31. N. Wirth. On the design of programming languages. In *IFIP World Congress 1974,* North-Holland, 1974.