Metamodel-Based Model Conformance and Multiview Consistency Checking

RICHARD F. PAIGE
University of York, UK
PHILLIP J. BROOKE
University of Teesside, UK
and
JONATHAN S. OSTROFF
York University, Canada

Model-driven development, using languages such as UML and BON, often makes use of multiple diagrams (e.g., class and sequence diagrams) when modeling systems. These diagrams, presenting different views of a system of interest, may be inconsistent. A metamodel provides a unifying framework in which to ensure and check consistency, while at the same time providing the means to distinguish between valid and invalid models, that is, conformance. Two formal specifications of the metamodel for an object-oriented modeling language are presented, and it is shown how to use these specifications for model conformance and multiview consistency checking. Comparisons are made in terms of completeness and the level of automation each provide for checking multiview consistency and model conformance. The lessons learned from applying formal techniques to the problems of metamodeling, model conformance, and multiview consistency checking are summarized.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Metamodeling, multiview consistency, formal methods, automated verification

ACM Reference Format:

Paige, R. F., Brooke. P. J., and Ostroff, J. S. 2007. Metamodel-based model conformance and multiview consistency checking. ACM Trans. Softw. Engin. Method. 16, 3, Article 11 (July 2007), 49 pages. DOI = 10.1145/ 1243987.1243989 http://doi.acm.org/10.1145/1243987.1243989

This research was partially supported by the National Sciences and Engineering Research Council of Canada, and by the European Commission under IST # 51173.

Authors' addresses: R. F. Paige, Department of Computer Science, University of York, Heslington, York YO10 5DD, UK; email: paige@cs.york.ac.uk; P. J. Brooke, School of Computing, University of Teesside, Middlesbrough, TS1 3BA; email: p.j.brooke@tees.ac.uk; J. S. Ostroff, Department of Computer Science and Engineering, York University, Toronto, Ontario, Canada; email: jonathan@cs.yorku.ca.

1. INTRODUCTION

Modeling languages such as UML 2.0 [Object Management Group 2003b] and BON [Walden and Nerson 1995] form the basis of model-driven approaches to building software systems. The *Model-Driven Architecture* (MDA) initiative of the OMG [Object Management Group 2003a] makes modeling languages the primary tool of developers in all stages of the systems engineering process whether carrying out requirements engineering, system modeling, deployment, implementation, testing, or transformation. The modeling languages used during model-driven development must be carefully designed, supportable and supported by tools, and understandable and explainable to developers and domain experts.

A modeling language is typically described using one or more *metamodels* [Object Management Group 2003b], which are used to define the *syntax* of the language, the well-formedness constraints it must obey, and the semantics of the language's constructs. Most often, metamodels are constructed to represent abstract syntax and well-formedness constraints, though some efforts are beginning to appear on metamodels of language semantics [Xactium 2006] and concrete syntax [Fondement and Baar 2005].

Metamodeling is now accepted as a critical part of the design of modeling languages [Evans et al. 2005]: without a precise, consistent, and validated metamodel specification, it is difficult to explain a language, build tools to support it and produce consistent and unambiguous models. The importance of metamodeling is reflected in the literature, for example, the recent workshop on metamodeling for support of MDA [Evans et al. 2003], the substantial effort placed on reengineering the metamodel of UML for version 2.0 [Object Management Group 2004b], the emphasis on metamodeling in Eclipse via the Eclipse Modeling Framework [Budinsky et al. 2003] and the focus of the EU Integrated Project MODELWARE, wherein metamodeling pervades all aspects of the project. Metamodeling is also widely accepted as a nontrivial task especially for industrial-strength languages which have large and complex syntaxes and semantics and often make use of *multiple cross-cutting views*. A view is a description that represents a system from a particular perspective (e.g., system architecture, behavior, contract, deployment), and thus includes a subset of the system's elements (e.g., modules). A view is often represented using a separate diagram, for example, class diagrams for structure, communication diagrams for system behavior (though standards such as IEEE 1471 [IEEE 2000] do not require this). Consider, for example, the simple system model presented in Figure 1, which illustrates two commonly used views each represented using a separate diagram.

On the left is a class diagram in BON syntax and on the right is a dialect of UML 2.0 communication diagrams. These represent the same system but from different perspectives, the architectural and the behavioral. A third view, represented by *contracts*, is also included in the left diagram; these contracts represent additional information about the services provided by classes and the conditions under which they can be used. It is important for developers to know and to be able to check that these views do not contain any contradictory

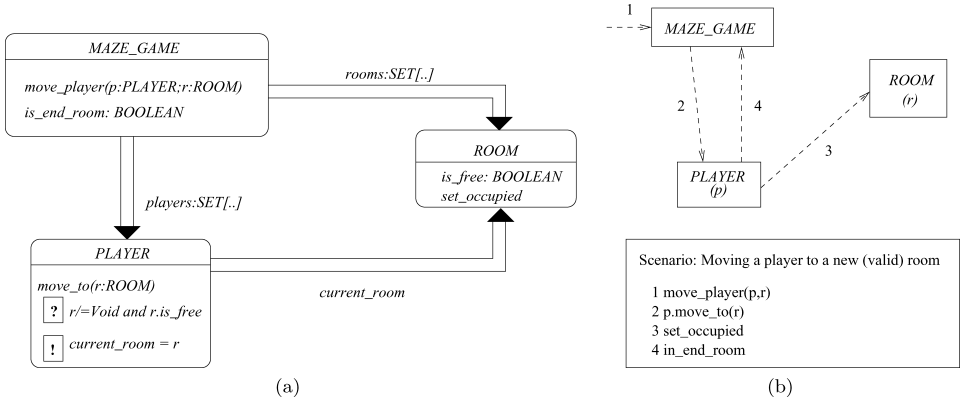


Fig. 1. Diagrams presenting a multiview model.

information before attempting to implement them. In this particular example, there are two inconsistencies that need to be detected: a routine that does not exist (*in_end_room*) is called, and a precondition is not true before a routine is called. We return to this example later in Section 4. More complex and realistic examples showing the value of multiview consistency checking are discussed in, MODELWARE [2005].

The contribution of this article is a comparison of two metamodeling approaches that can be used to detect inconsistencies like the ones contained in Figure 1. The approaches differ in terms of their completeness (i.e., the kinds of inconsistencies that can be detected) and the level of automation provided. We make this more precise in the next section.

1.1 Model Conformance and Multiview Consistency Checking

The presence of a clearly specified, understandable, tool-supported metamodel for a modeling language makes it feasible to carry out *model conformance* and *multiview consistency checking* (MVCC). A metamodel captures the syntax and semantics of all the modeling concepts in a language (e.g., concepts such as classes, objects, procedures, processes, documents, and services) and thus provides the context needed for expressing well-formedness constraints for models and on multiple views. With model conformance, a model is checked that it satisfies the constraints captured in the metamodel, that is, that the model is indeed a valid instance of the metamodel. With MVCC, it is shown that two or more diagrams, each presenting a different view, do not contradict each other according to a set of (metalevel) rules. As we shall see shortly, it is possible to unify the definitions of MVCC and model conformance checking.

The constraints encoded in a metamodel in turn lead us to a precise definition of model conformance and multiview consistency as follows.

Formally, model conformance, that is, checking that a model satisfies the well-formedness constraints of a language, and multiview consistency checking can be defined as follows. Let L be a modeling language, and MM a metamodel for L. Let M be a model expressed in the language L. MM consists of a set

of constraints which can conceptually be partitioned into those capturing the abstract syntax of L, and the semantics of L, that is, $MM \mathrel{\widehat{=}} SYNTAX \cup SEM$. The semantic constraints can conceptually be partitioned into those that define the semantics of a single view (e.g., class diagrams in UML) and those that define the consistency of multiple views.[1] Thus, if $v1$ and $v2$ are views expressible in language L, then

$$\begin{aligned} SEM \;\mathrel{\widehat{=}}\; & \forall v1 \in L \bullet SVC(v1) \wedge \\ & \forall v1, v2 \in L \mid v1 \neq v2 \bullet MVC(v1, v2), \end{aligned}$$

where $SVC(v1)$ is the set of semantic constraints on the view $v1$, and $MVC(v1, v2)$ is the set of semantic constraints on the two views $v1, v2$.

The notion of model M *conforming* to the metamodel MM is defined as follows.

$$conforms(M, MM) \mathrel{\widehat{=}} \forall c \in MM \bullet M \;\mathbf{sat}\; c,$$

that is, the model M satisfies each well-formedness constraint c of the metamodel MM. Note that if the model M consists of two or more views (e.g., a class diagram and a statechart in UML) then $conforms(M, MM)$ will establish that (a) M conforms to the metamodel and (b) the views are mutually consistent.

Of course, steps (a) and (b) can be separated, that is, the definitions of conformance and multiview consistency can be constructed independently, and this may be useful in specifying the behavior of separate conformance and consistency-checking tools. Our definition suggests that, for metamodel-based approaches, it is appropriate to unify the notions of model conformance and multiview consistency at least conceptually.[2] It also implies that mathematical techniques can play a substantial role in these processes.

In practice, a variety of different views are used in model-driven development [Object Management Group 2004b]. Most widespread are the views presented by a variety of UML 2.0 diagrams: class diagrams (i.e., a structural view), state charts (i.e., a behavioral view pertaining to the properties of a single class or object), communication diagrams (i.e., a behavioral view pertaining to the properties of more than one object), and executable code. Class contracts, for instance, pre and postconditions of routines of a class, can also be considered as views, though they are not usually represented using diagrams and are often integrated within other views (e.g., in Catalysis [D'Souza and Wills 1998] and in BON). As well, different versions of the same kind of diagram can be perceived as presenting different views of a system which must be consistent. Conceptually, there is no difference between multiview consistency checking when applied to different kinds of diagrams versus different versions of diagrams and similar techniques can be used for each, for example, see MODELWARE [2005]. However, additional model management problems exist when dealing

[1] In practice, separating single view and multiview consistency constraints may be challenging, but in principle it is possible, and, moreover, it offers a discipline for constructing languages.

[2] Additional techniques will likely be needed for non-MVCC, for example, for checking consistency between refinement steps in MDA [Object Management Group 2003a; Paige et al. 2005].

with different versions of models such as the need to carry out model merging. We do not discuss this related topic further in this article.

Most of the previous work on metamodel-based multiview consistency, discussed in Section 2, has focused on class diagrams and state charts and very little research has examined the use of *contracts*, that is, pre and postconditions on messages appearing in sequence diagrams, unlike the work in this article. Contracts are of increasing importance in model-driven development; they are supported by a number of languages and methodologies, such as UML 2.0 and Catalysis, as well as the recent Architecture Analysis and Design Language (AADL) standard [Society of Automotive Engineers 2005], which makes use of a contract annex for improving the precision of modeling and improving the capabilities for analysis. As such, it is important to address their impact on multiview consistency checking.

1.2 Aims of This Article

This article contributes a detailed comparison of using two different (tool-supported) approaches to metamodeling, model conformance, and multiview consistency checking, including contracts. The work applies to any modeling language, such as UML 2.0 with OCL and BON, that supports views presented by class diagrams, and communication diagrams and that also supports use of contracts for capturing detailed behavior of objects. In particular, we contribute techniques and comparisons of approaches used for handling the consistency issues raised by using contracts in the diagrams. By including contracts, we implicitly introduce a third view, that which describes the behavior of classes of objects.

We have focused on these views because they present more challenges—particularly when considering contracts—than other views and also because there is less related work. There is nothing in the approaches presented and compared in this article that prevent them from being extended to additional views (e.g., deployment, business rules). The extension process may be made easier by having a clear understanding of how to construct metamodels and carry out consistency checking for the structural and behavioral views discussed earlier.

We present this synthesis in order to suggest guidelines and recommend practices on the development of modeling languages. Our aim is to provide pragmatic advice to users and developers of modeling languages on useful ways in which to carry out metamodeling and metamodel-based conformance and multiview consistency checking in a practical, tool-supported way, based on the use of formal techniques. This advice in turn could influence decisions on the techniques used to construct and validate metamodels and language designs, in the future.

To this end, we compare and contrast two approaches to metamodeling for object-oriented languages: one approach uses the PVS specification language [Owre et al. 1999], while the second uses Eiffel [Meyer 1992] as a specification language, that is, we make use of Eiffel's *declarative specification* techniques, particularly agents (discussed in Section 2) for expressing metamodels. Both

Table I. Criteria for Comparison of Metamodeling and MVCC Approaches

—**Understandability** is the extent to which the descriptions used are understandable to a reasonably experienced software engineer who has modeling language but not necessarily formal methods experience.
—**Correctness** is the extent to which metamodel descriptions have been checked against their specifications.
—**Completeness** is the extent to which all features of the modeling language (including multiview consistency constraints) are supported in the metamodeling approach.
—**Maintainability** is the extent to which the descriptions support extension, refactoring, and wholesale modification.
—**Tool construction** is the extent to which tools are available to support producing the descriptions.
—**Tool-based V&V** is the extent to which tools are available to assist in verifying and validating the descriptions.
—**Automation in MVCC** is the extent to which multiview consistency checking can be carried out automatically.

approaches also support model conformance checking. The two languages used for metamodeling and checking in these approaches, theorem proving and object-oriented programming, respectively, are very different and have advantages and disadvantages which we attempt to draw out in our discussion. We choose PVS as it is representative of the state-of-the-art in theorem-proving technology. We choose Eiffel as it is representative of the state-of-the-art in executable object-oriented languages that support contracts; an alternative to Eiffel might be Spec# [Microsoft 2006], but it is still under development.

We use these metamodel specifications to carry out a comparison of two approaches to metamodel-based multiview consistency checking and describe the advantages and disadvantages of the approaches. We particularly focus on the lessons learned in terms of the *completeness* of the approaches, that is, does the approach support all elements of the modeling language in full—and in terms of the level of automation offered in checking the *conforms* relation, that is, for carrying out conformance and multiview consistency checking.

It is important to note that when comparing the two approaches to metamodeling, we are not attempting a like-for-like comparison, that is, attempting to use the same approach for metamodeling in both PVS and Eiffel. Instead, we will consider best practices with each technology, focusing on the typical way in which an engineer might use PVS or Eiffel for metamodeling, model conformance, and multiview consistency checking. Since the approaches offered are as one would expect quite different, we will need to identify criteria in order to have a common basis for comparison. These will be precisely specified in Sections 3 and 4; an informal summary of the criteria that will be used is in Table I. There is a level of subjectivity in comparisons involving these criteria; the aim of the comparison is to provide general guidance on how to construct metamodels, as well as how to select metamodeling languages and multiview consistency-checking approaches.

We are not aiming at a comprehensive overview of the field; in particular, we do not consider all aspects of consistency checking in the domain of the MDA. We omit refinement-based consistency in MDA, though many of the techniques in this article, we claim can be extended to handle this (see Paige et al. [2005] for

more details on this). Further, we are focusing on metamodel-based techniques since these are critical and widely used in the model-driven development community rather than fully considering the problems of model conformance and consistency.

The comparisons will be carried out using the BON modeling language [Walden and Nerson 1995] as an exemplar (see Section 2). BON is concise enough so as to convey and cover all of the language within this article. BON provides three views: a static view presented by class diagrams, a dynamic view presented by a dialect of collaboration diagrams, and a behavioral view presented using contracts (pre and postconditions of routines, plus invariants of classes). Contracts effectively subsume the information that is often presented using state diagrams (e.g., in UML 2.0) in some modeling languages. Contracts are widely used in object-oriented programming and are also used via OCL in UML 2.0. However, while contracts increase the expressiveness of a modeling language, they introduce additional challenges for metamodeling and multiview consistency checking. Integrating support for contracts into metamodeling and MVCC approaches is one of the key contributions of this article. While our arguments and presentation are framed in terms of BON and its diagrams, they can easily be generalized to apply to UML as well, as we discuss in Section 2.

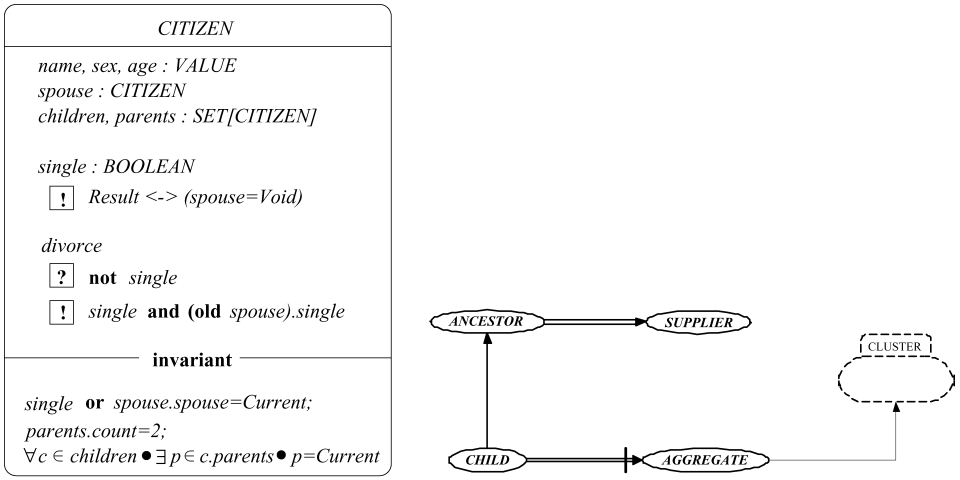
We start with an overview of BON, and also describe other related recent work in metamodeling and multiview consistency checking. As well, we relate BON to UML in order to broaden the scope of the work; we summarize the key differences between BON and UML. We then describe two approaches to metamodeling and model conformance for BON and compare and contrast the results. The metamodels are next used in two approaches to multiview consistency checking. At key points while presenting the metamodels and techniques, we briefly explain how the techniques could apply to UML as well. A summary of our findings and a discussion conclude the article.

2. BON AND RELATED WORK

2.1 BON

BON [Walden and Nerson 1995] is an object-oriented method, consisting of a modeling language and development process for building reliable systems. It is supported by a number of tools, including Visio and EiffelStudio. Its language has been designed to work seamlessly and reversibly with Eiffel: BON diagrams can automatically be produced from Eiffel code, and Eiffel code can be generated from BON diagrams. There are three main parts to BON's language. The first, class interfaces, is demonstrated with an example in Figure 2(a).

The name of the class is at the top of of Figure 2(a). The middle section is made up of *features* (attributes or routines). Routines are either functions (returning a value) or procedures (effecting state changes) and may have preconditions (denoted using a ? in a box, which are assertions that must be true before any call to the routine) and postconditions (denoted using a ! in a box, which are assertions that must be true after the routine body has executed). At the bottom



(a) Citizen interface (b) BON relationships

Fig. 2. BON syntax for interfaces and relationships.

of the class is its *invariant*, a set of assertions that must be true before and after each client call to a function or procedure. Assertions are written in BON's first-order dialect of predicate logic, which includes propositions, universal and existential quantifiers, type interrogation, and reference types (along with the usual arithmetic and boolean operators and values).

The second part of BON is the class diagram notation. Class diagrams consist of classes organized in *clusters* (drawn as a dashed rounded rectangle in Figure 2(b)), which interact via two kinds of relationships.

—*Inheritance.* Inheritance defines a subtype relationship between child and one or more parents. The inheritance relationship is drawn between classes *CHILD* and *ANCESTOR* in Figure 2(b) with the arrow directed from the child to the parent class. In this figure, classes have been drawn in their compressed form as ellipses with interface details hidden. Inheritance can also be drawn so that the source or target is a cluster (a set of one or more classes or other clusters), for instance, as drawn between *AGGREGATE* and *CLUSTER* in Figure 2(b). In this particular example, the inheritance arrow indicates that *AGGREGATE* inherits behavior from one or more classes in *CLUSTER*.

—*Client-Supplier.* There are two client-supplier relationships, association and aggregation. Both relationships are directed from a client class to a supplier class; the relationship has an optional label. With association, the client class has a reference to an object of the supplier class. With aggregation, the client class contains an object of the supplier class. The aggregation relationship is drawn between classes *CHILD* and *AGGREGATE* in Figure 2(b), whereas an association is drawn from *ANCESTOR* to class *SUPPLIER*.

The third part of BON's language is dynamic diagrams (similar to communication diagrams in UML 2.0). The syntax for these diagrams is very similar

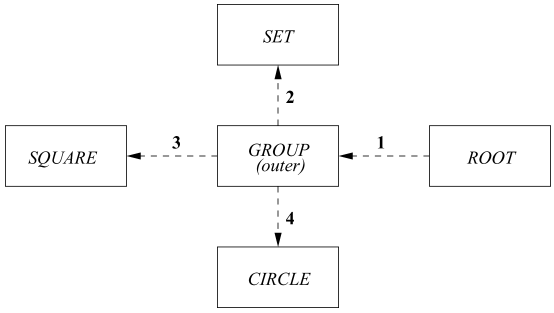


Fig. 3. BON dynamic diagram.

to UML; it is used to depict objects, messages sent between objects, and sequencing information, where a message corresponds to a potential routine call on a target object. An example is shown in Figure 3. Rectangles represent objects, while dashed arrows represent messages. Four objects are anonymous in Figure 3, whereas object *outer* is of class *GROUP*. Each message has a number that is cross-referenced to a scenario box; this box indicates the routine that is bound to the message and which is invoked when the message is received by the target. Messages are sent (and hence routines invoked) in the order indicated by the message numbers. When a message is received and before its corresponding routine is invoked, the precondition of this routine should be true; note that this is not guaranteed, but it must be checked (e.g., by the sender of the message). Typically such diagrams are not used to capture exceptional behavior (e.g., what a system does if a precondition fails on sending a message) but exceptions can be described in a similar way to other scenarios (e.g., by adding additional messages).

2.1.1 *BON, UML, and Domain-Specific Modeling.* BON was chosen as the language of discourse in this article because of its size and conciseness and also because of its integrated support for contracts. However, the lessons and observations made in this article apply to UML 2.0 as well and also to domain-specific modeling languages, such as the aforementioned AADL. To help clarify this, we now briefly contrast BON with UML and domain-specific modeling.

BON is equivalent to a profile of UML 2.0, consisting of a subset of class diagrams, the Object Constraint Language (OCL), and communication diagrams. BON purposely omits all other diagrams to maintain semantic coherence [Walden and Nerson 1995; Meyer 1997].

BON class diagrams are equivalent to UML class diagrams, omitting UML aggregation (which is inexpressible in BON). BON provides a single concept, the class, which subsumes UML's interface, abstract class, and class. BON uses stereotypes on classes to denote variants of the general-purpose class concept; this allows specification of interface-like concepts. BON supports only a small set of fixed stereotypes unlike UML. Aggregation is omitted because it cannot be mapped directly into a programming language [Walden and Nerson 1995]; UML composition in contrast, can be mapped into Eiffel's expanded types or C++'s variables and is supported in BON via its aggregation notation (this is

admittedly confusing, but the short of it is that BON aggregation is equivalent to UML composition).

BON dynamic diagrams are semantically equivalent to UML 2.0 communication diagrams, omitting guards. UML's concept of guards are subsumed in BON within the contracts of individual routines. Constraints on repetition of messages in UML are expressed either using routine contracts or by using multi-messages (i.e., messages that are sent to multiple objects simultaneously); the former is preferred. A syntactic novelty with BON dynamic diagrams is that it provides precise rules for flattening or expanding collections of messages using clusters; this is helpful for improving the readability of diagrams.

BON's contract language is very similar to OCL 2.0, though there are some differences in the type systems (BON does not provide direct equivalents to many of OCL's built-in types). BON is also based on two-valued predicate logic, whereas OCL supports three-valued logic. OCL also provides mechanisms for relating constraints with UML state charts. A detailed comparison of the contract languages is outside of the scope of this article, but the reader is referred to Paige and Ostroff [1999a] and Spencer [2005]. The former contrasts BON with UML and OCL in detail, while the latter implements a substantial subset of OCL using Eiffel's contract language (which is effectively a subset of BON).

BON can also be viewed as a domain-specific modeling language for Eiffel. BON has been designed (a) to be useful for visually representing substantial parts of Eiffel programs, and (b) so that each BON concept maps seamlessly and reversibly to an Eiffel concept (i.e., each concept in BON has a direct equivalent in Eiffel). This makes it straightforward to implement code generators and reverse engineering tools for BON, but it raises questions about using BON for analysis, as it omits a number of modeling constructs that UML provides. However, the BON contract language (a dialect of first-order predicate logic) is considered suitably rich and expressive to mitigate for the (visual) modeling constructs omitted from the language [Walden and Nerson 1995].

2.2 Eiffel

Eiffel is an object-oriented programming language [Meyer 1997]; it provides constructs typical of the object-oriented paradigm, including classes, objects, inheritance, associations, composite (expanded) types, generic (parameterised) types, polymorphism and dynamic binding, and automatic memory management. It has a comprehensive set of libraries, including data structures, GUI widgets, and database management system bindings, and the language is integrated with .NET.

A short example of an Eiffel class is shown in Figure 4. The class *CITIZEN* inherits from *PERSON* (thus defining a subtyping relationship). It provides several attributes, for example, *spouse*, *children* which are of reference type (*spouse* refers to an object of type *CITIZEN*, while *children* refers to an object of type *SET*[*CITIZEN*]); these features are publicly accessible (i.e., are exported to *ANY* client). Attributes are of reference type by default a reference attribute either points at an object on the heap, or is *void*. The class provides one expanded attribute, *blood_type*. Expanded attributes are also known as composite

```
class CITIZEN inherit PERSON
feature {ANY}

  spouse: CITIZEN
  children, parents: SET[CITIZEN]
  blood_type: expanded BLOOD_TYPE

  single: BOOLEAN is
    do Result := (spouse=Void)
    ensure Result = (spouse=Void)
    end

  set_spouse(s:CITIZEN) is do spouse := s end

feature {BIG_GOVERNMENT}

  divorce is
    require not single
    do spouse.set_spouse(Void); spouse := Void end
    ensure single and (old spouse).single
    end

invariant
  single or spouse.spouse = Current;
  parents.count <= 2;
  children.for_all((c:CITIZEN):BOOLEAN
    do Result := c.parents.has(Current) end);
end -- CITIZEN
```

Fig. 4. Eiffel class interface.

attributes; they are not references, and memory is allocated for expanded attributes when memory is allocated for the enclosing object.

The remaining features of the class are routines, that is, functions (like *single*, which returns *true* if and only if the citizen has no spouse) and procedures (like *divorce*, which changes the state of the object). These routines may have preconditions (*require* clauses) and postconditions (*ensure* clauses). Finally, the class has an invariant, specifying properties that must be true of all objects of the class at stable points in time, that is, before any valid client call on the object. In writing the invariant of *CITIZEN*, we have used Eiffel's *agent* notation. Agents are a way to encapsulate operations in objects; the operation can then be invoked on collections (e.g., a set or linked list) when necessary. Two built-in agents in Eiffel are `for_all` and `there_exists`, which can be used to implement quantifiers over finite data structures. In this example, one agent is used in the class invariant: `for_all` iterates over all elements of *children* and returns *true* if its body, applied to each element, returns *true*. The body is a boolean expression which returns *true* if and only if the current citizen is a child of one of its parents. In other words, the agent expression is true if and only if all children have links to their parents.

Fig. 5. The BON metamodel, abstract architecture.

Other facilities offered by Eiffel but not demonstrated here include dynamic dispatch, multiple inheritance, and static typing. We refer the reader to Meyer [1992] for full details on these and other features.

2.2.1 *Eiffel as a Formal Specification Language.* Eiffel is a wide-spectrum language [Meyer 1997; Paige and Ostroff 1999b, 2004] suitable for both programming and lightweight formal specification. It has been designed to support seamless development, where one language is used throughout the development process. Eiffel is not a small language, but a subset of it can be identified and applied for formal specification. An *Eiffel formal specification* is written using only the following constructs.

—Classes and class interfaces (containing routine signatures and attributes).
—Local variables of routines
—Boolean expressions (including agents)
—Routine calls of the form $o.f$, where o is a variable or attribute, and f a routine
—Assignment statements in routine bodies
—Sequential composition of routine calls and assignment statements
—Preconditions and postconditions of routines, and class invariants

All other Eiffel constructs are excluded. This subset is roughly similar to the subset identified in the Eiffel Refinement Calculus [Paige and Ostroff 2004], which allows Eiffel formal specifications to be refined to programs. A formal semantics for these constructs can be found in Paige and Ostroff [2004].

2.3 Overview of the BON Metamodel

The BON metamodel was specified in BON itself in Paige and Ostroff [2001], keeping with the approaches followed in the UML community. We give a summary of the metamodel in this section, focusing on the key components and packages. This metamodel will be used for comparing different approaches in later sections.

Figure 5 contains an abstract depiction of the BON metamodel. BON models are instances of the class *MODEL*. Each model has a set of abstractions. The two clusters, representing abstractions and relationships, will be detailed shortly.

The class *MODEL* possesses a number of features and invariant clauses that will be used to capture the well-formedness constraints of BON models. These features are depicted in Figure 6, which shows the interface for *MODEL*. We will not provide all the details of the individual clauses of the

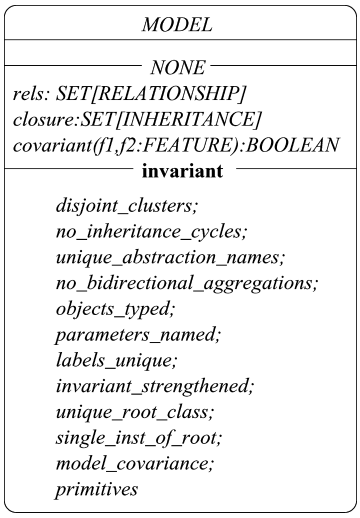


Fig. 6. Interface of class *MODEL*.

class invariant of *MODEL*, though examples will be considered in the following sections.

The relationships cluster describes the four basic BON relationships (including messages between objects) as well as constraints on their use. The details of the cluster are shown in Figure 7.

Covariant redefinition[3] is used in Figure 7 to capture well-formedness conditions, for instance, that inheritance can connect only static abstractions.

Invariants are written on classes representing inheritance and aggregation relationships to express that these relationships cannot be self-targeted.

The abstractions cluster describes the abstractions that may appear in a BON model. It is depicted in Figure 8.

ABSTRACTION is a deferred class: instances of *ABSTRACTION*s cannot be created. Classification is used to separate all abstractions into two subtypes, static and dynamic abstractions. Static abstractions are *CLASS*es and *CLUSTER*s. Dynamic abstractions are *OBJECT*s and *OBJECT_CLUSTER*s. Clusters may contain other abstractions according to their type, that is, static clusters contain only static abstractions.

The features cluster describes the notion of a feature that is possessed by a class. Features have optional parameters, an optional precondition and postcondition, and an optional frame. The pre and postcondition are assertions. Query calls may appear in assertions; the set of query calls that appear in an assertion must be modeled in order to ensure that the calls are valid according to the export policy of a class. Each feature will thus have a list of *accessors*,

[3]Covariant redefinition allows types to be replaced by descendents in a child class; this applies to both the types of parameters and the type of function result. This is permitted in both BON and Eiffel. It should be contrasted with no variance in Java, and partial covariance (for function results only) in C++.

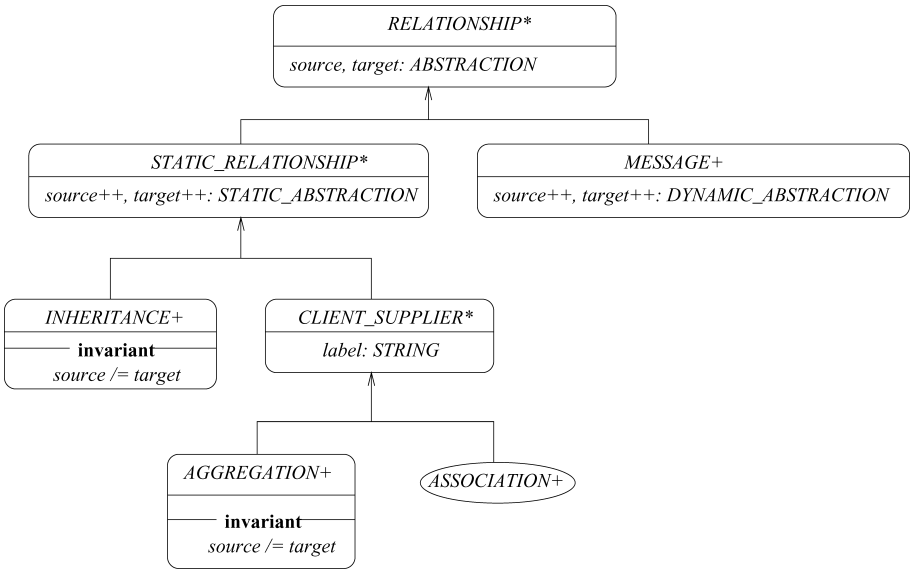


Fig. 7. The relationships cluster.

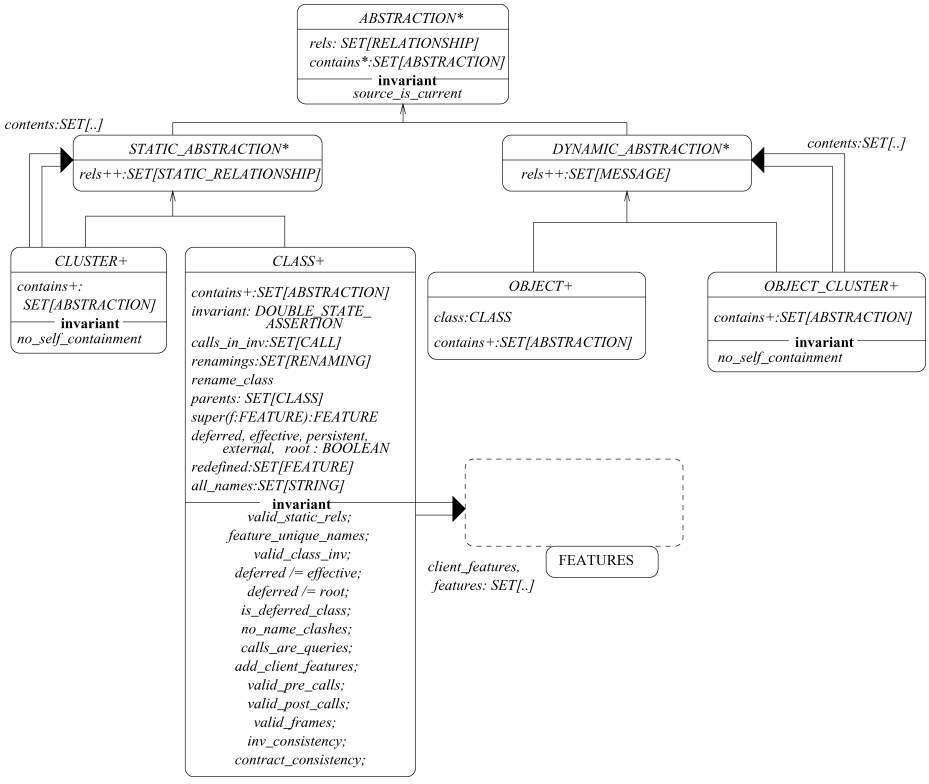


Fig. 8. The abstractions cluster.

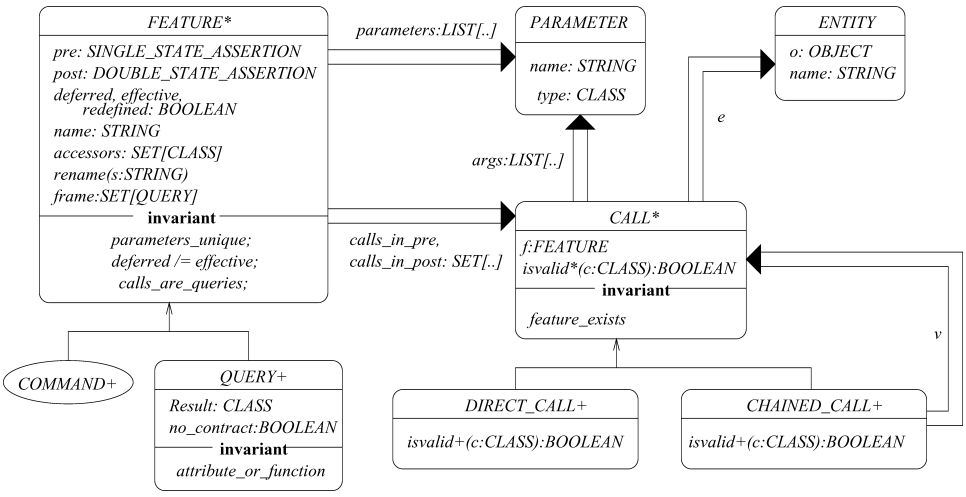


Fig. 9. BON metamodel, features cluster.

which are classes that may use the feature as a client. A call consists of an entity (the target of the call), a feature, and optional arguments. The frame is a set of attributes that the feature may modify. Figure 9 depicts the cluster.

2.3.1 *Summary.* This section has summarized the BON metamodel, primarily to illustrate that the BON language, while smaller than UML, still provides useful modeling constructs and is nontrivial from a metamodeling perspective. This (semiformal) summary attempts to promote understandability by diagrammatically expressing well-formedness constraints and by allowing projection of abstract views of the metamodel. In this sense, this mimics the approach taken in the OMG to metamodeling UML.

The main disadvantage with specifying the BON metamodel in BON directly is that the language's semantics is not fully and formally defined (parts of the language are formalized in Paige and Ostroff [1999b]), and there are as of yet no tools that can be used to directly check the correctness of the invariants and assertions contained within the classes. This criticism only partly applies to the UML 2.0 metamodel: while a full formal semantics for UML 2.0 does not yet exist, the fact that UML 2.0 is defined in terms of MOF, and there are tools that partly support MOF, means that partial checking of the UML metamodel can be carried out automatically.

Given the lack of formal semantics and the need to use translation to obtain verification and validation capabilities, the value of using BON to express its metamodel directly is circumscribed. A second, related disadvantage is that it is difficult to use the metamodel, as specified, to carry out multiview consistency checking because of a lack of formal semantics and automated tool support. A final disadvantage is that it is difficult to validate and verify the metamodel expressed directly in BON. The advantage of using BON (or other visual languages) to express metamodels is that the results are generally more easily understood than other representations.

2.4 Related Work on OO Metamodeling and Multiview Consistency Checking

Our focus in this article is specifically on metamodeling and consistency checking in the domain of object-oriented and systems modeling, and this guides our overview of related work. There has been much recent related work on OO metamodeling and on using these metamodels to carry out conformance checking and checking the consistency of multiple views. More general work on consistency, for instance, for requirements [Finkelstein et al. 1994], for handling and managing inconsistency using model checking [Chechik et al. 2003] and general XML-based tool support [Gryce et al. 2002] is discussed elsewhere and is outside our scope.

Work on OO metamodeling has been dominated by work on UML. This work can generally be categorized into that which presents semi-formal approaches to metamodeling—for example, using UML 2.0 itself [Object Management Group 2004b], the Meta-Object Facility [Object Management Group 2004a], or even other semiformal modeling languages such as BON [Paige and Ostroff 2001], and that based on formal approaches to metamodeling, for instance, using Object-Z to specify the UML metamodel [Kim and Carrington 2004], PVS [Paige et al. 2003b], OCL [Object Management Group 2004b], XMF-Mosaic [Xactium 2006], MML [Clark et al. 2001a], Executable UML [Mellor and Balcer 2002], and the template-based approaches to specification in Z [Amalio et al. 2004]. The semiformal modeling work tends to be more complete (in terms of supporting different diagrams) than the formal-based work. The formal-based work tends to focus on supporting specific views, for example, class diagrams and statecharts—or proves incomplete specifications of several views. The formal-based metamodeling work has a number of similarities to efforts made on formal modeling of object-oriented systems, for example, using Spec# [Microsoft 2006], JML [Leavens et al. 2005], or algebraic specification languages such as CASL [Bidoit and Mosses 2004], though these languages are generally used to reason about programs rather than metamodels and language definitions.

The general view in the UML community is that using a semiformal modeling language (such as UML, or MOF) to capture a metamodel is preferable to using formal techniques since more developers, particularly, tool builders, will be able to construct, interpret, and implement a metamodel expressed in UML as opposed to, for example, a formal language such as Z or PVS where there are substantial differences between the domains of discourse. Even so, it is generally more difficult to reason about and analyze metamodels that are specified using a semiformal language like UML. Some reasoning and analysis is possible with languages such as XMF-Mosaic and OCL, predominantly via simulation techniques, though the former is only supported by a single tool, and the latter is frequently only supported for a subset. The OCLE tool (discussed later) [Chiorean 2005] provides very rich support for OCL currently and supports the entire language. In general, for semiformal metamodels, it is difficult to verify and validate the specifications because of a lack of a formal semantics or because of limited simulation and execution mechanisms.

There is less related work on using OO metamodels for model conformance. In part, this is because many tools for OO modeling treat the metamodel as

a specification for user interface constraints, that is, the interface for the tool attempts to prevent illegal models from being constructed. However, tools such as Visio [Microsoft 2005], which allow rule checking to be turned off and on as desired, require an explicit encoding of well-formedness rules in some form. Some specific related work on OO metamodeling and model conformance is described in the following.

—The OCLE tool [Chiorean 2005] supports OCL 2.0 and UML 1.5 and provides the means to check UML models against the metamodel constraints and more general rules. It is a stand-alone modeling tool that can be interfaced with other tools via XML and XMI specifications. For practical use, the tool could be used as a backend to a full-featured modeling tool, providing rule-checking facilities via simulation. There may be challenges in terms of linking feedback from OCLE simulations to such a modeling tool.

—Similarly, the Kent Modeling Framework (KMF) [Akehurst et al. 2004] supports the metamodel for OCL 2.0 and provides parsing, simulation, and execution facilities. The framework aims to provide a full-featured, open-source package for modeling based on UML-compatible metamodels and conceptually can be used for conformance and consistency checking by specifying metamodel rules in OCL and executing them against models.

—XMF-Mosaic [Xactium 2006] is a self-contained metaobject programming environment that provides among other things an implementation of the UML metamodel. The tool can be instantiated with metamodels and thereafter supports the construction of models that conform to the metamodel. The tool supports an executable dialect of OCL—XOCL, and thus also provides the means for checking UML diagrams against the metamodel via simulation and execution. The emphasis with XMF is tool customization based on metamodeling techniques as well as expressing language transformations and supporting the QVT proposal.

—Similarly, MMT [Clark et al. 2001b], a stand-alone metamodeling tool for MOF-compliant languages, provides support for a subset of OCL and enables checking instances of the metamodel against one or more rules captured in the metamodel. MMT appears to have been deprecated by the development of XMF-Mosaic.

—The Dresden OCL toolkit [Hussman et al. 2000] supports compilation of OCL constraints; it has been integrated into the Argo/UML tool. In principle, it could be used to help validate the UML metamodel expressed in UML and OCL.

—The KeY project [Ahrendt et al. 2005] offers an extension of the Together ControlCentre case tool that provides theorem-proving technology for checking OCL constraints. As with the Dresden OCL toolkit, it could be applied to metamodel validation.

—The USE stand-alone tool [Richters and Gogolla 2000] is an OCL simulator that can be used to execute OCL constraints against model snapshots. Used in batch mode, it can be applied to carry out model validation and

verification. It has been recently updated to support substantial parts of OCL 2.0 and as such can be used for validating and testing parts of the UML metamodel. USE is distinctive as it is one of the few pieces of work that reports on how to check models against the metamodel using semiautomated tool support.

Work related to metamodeling and conformance checking arises in the domain of multiview consistency. The basic problem is to ensure that separate descriptions of an OO system are mutually consistent, that is, that information in one description does not contradict information contained in a separate description.

Many solutions have been presented for this problem. The recent series of workshops on consistency checking in the context of UML [Huzar et al. 2002, 2003, 2004; Kuzniarz et al. 2005] demonstrate techniques that consider different system views (particularly static and behavioral) and different lightweight and heavyweight techniques for implementing consistency checking. Some of these approaches require use of specialized tools such as xlinkit [Gryce et al. 2002], a general-purpose rule-based tool for consistency-checking documents based on XML—or mathematical languages, for example, [Marcano and Levy 2002]—for expressing the well-formedness constraints that establish multiview consistency.

Lightweight approaches to consistency checking are desirable and have been explored, but a limitation with most of these is that they are invariably incomplete, covering parts of a language (e.g., statecharts and class diagrams in UML). The work of Bhaduri and Venkatesh [2002] suggests the use of message sequence charts and model checking for multiview consistency checking. They express the semantics of the object life-cycle and scenario views as a labeled transition system, thus enabling the use of a model checker to identify inconsistencies. The advantage of this approach is that the model checking will be automatic, however, there are limitations on what can be expressed in terms of properties and models. In an alternative approach, UML model consistency is checked via the Sherlock tool [Sourrouille and Caplat 2002], wherein actions, models, and a knowledge base are combined. This approach is particularly promising as it also considers extensions to UML profiles that target specific problem domains. Krishnan [2000] considers OCL contracts in the context of UML and formalises multiview consistency in PVS. Krishnan [2000] requires users to formally unroll sequencing of method calls arising in sequence diagrams, which can be very awkward for large sequence and collaboration diagrams; an alternative approach is presented in Section 4. Work at IBM Haifa has studied metamodel extensions for consistency checking in the context of Rational's XDE tool Huzar et al. [2004]. This work is similar in scope to what we present in the next section.

A key limitation that can be identified from this work is that much of it does not consider contracts, that is, method pre and postconditions, when carrying out multiview consistency checking; Krishnan [2000] is an exception but is difficult to use. Both approaches to metamodeling and multiview consistency checking presented in the next sections consider contracts in different ways.

3. METAMODELING IN PVS AND EIFFEL

We now contrast two approaches to specifying the metamodel for BON; the metamodel was semiformally specified in an earlier section. We contrast two formal specifications: one written in PVS and one in Eiffel. Both these metamodels are formal as they are specified in languages with both formal syntax and semantics, and have tools to support checking models against the metamodel. What is interesting and different about the two formal metamodels is (a) their design; (b) the executability of the resulting spcifications, and (c) their completeness. Again, we are not attempting a like-for-like comparison of metamodel specifications: we will construct the metamodels using the best practices available for PVS and Eiffel. As such, some differences in terms of how well-formedness constraints are captured will arise.

There is, of course, an issue with producing the PVS and Eiffel descriptions of the BON metamodel, particularly, how do we show that these formal descriptions accurately capture the metamodels and well-formedness constraints? We answer this in two parts: first, by systematically presenting the PVS and Eiffel specifications and clearly relating them to the BON specifications in Section 2.3 (i.e., by making the relationships between descriptions clear and obvious), and second by analyzing the descriptions using the PVS theorem prover and type checker. Being able to successfully use the PVS and Eiffel metamodels for conformance and multiview consistency checking gives us greater confidence (though certainly no guarantee) of the accuracy of our descriptions. We intend, in the future, to implement the transformations from BON to PVS and Eiffel using standardized transformation tools, but again this is no guarantee of correctness. However, by following it we can reduce the likelihood of introducing errors.

3.1 Metamodel Specification in PVS

The disadvantages that were apparent with the metamodel specified in BON were primarily due to the lack of a formal semantics for the BON language. These disadvantages can be alleviated by using a different language with a formal semantics. In this section, we present a formal specification of the BON metamodel in the PVS specification language. We present the PVS version of the metamodel selectively, and attempt to give the flavor of using PVS for this purpose. We note that PVS is a general purpose theorem-proving environment, and as such it is not tailored for metamodeling.

3.1.1 *Theory of Abstractions.* To express the cluster of abstractions in PVS, we introduce a new nonempty type and a number of subtypes, effectively mimicking the inheritance hierarchy presented in Figure 8. A similar technique could be used for expressing the UML 2.0 concept of Classifier [Object Management Group 2004b] and its subtypes or implementations if one desired to use PVS to express the UML 2.0 metamodel. This information is declared in the PVS theory `abs_names`. Note that this specification includes both static language constructs (like classes) as well as dynamic constructs (like objects), thus supporting both class and dynamic diagrams from BON.

```
abs_names: THEORY
BEGIN
  ABS: TYPE+

  % Static and dynamic abstractions.
  STATIC_ABS, DYN_ABS: TYPE+ FROM ABS

  % Instantiable abstractions
  CLASS, CLUSTER: TYPE+ FROM STATIC_ABS
  OBJECT, OBJECT_CLUSTER: TYPE+ FROM DYN_ABS
END abs_names
```

The PVS theory `abstractions` then uses `abs_names` to introduce further modeling concepts as well as the constraints on abstractions that appear in models. Further concepts that we need to model include features and entities (that appear in calls), calls themselves, and parameters and arguments to routines. Primitive BON classes, for example, *INTEGER*, are modeled as PVS constants, objects of `CLASS` type. We also define conversions so that the type checker can automatically convert BON primitives into PVS types.

We must now describe constraints on abstractions. In the BON version of the metamodel, these took the form of features and class invariants. In the UML 2.0 metamodel [Object Management Group 2004b], these generally take the form of OCL constraints. In PVS, the well-formedness constraints will appear as functions, predicate subtypes, and axioms.

For example, a number of constraints will have to be written on features. To accomplish this, we introduce a number of functions that will let us acquire information about a feature such as its properties, precondition, and postcondition.[4]

```
feature_pre, feature_post: [ FEATURE -> bool ]

% Properties of a feature.
deferred_feature, effective_feature, redefined_feature: [ FEATURE -> bool ]

% The set of classes that can legally access a feature.
accessors: [ FEATURE -> set[CLASS] ]
```

We now provide examples of axioms which define the constraints on BON models. The first example ensures that all features of a class have unique names (BON does not permit overloading based on feature names or signatures).

```
feature_unique_names: AXIOM
 (FORALL (c:CLASS): (FORALL (f1,f2:FEATURE):
     (member(f1,class_features(c)) AND member(f2,class_features(c)))
     IMPLIES (feature_name(f1) = feature_name(f2)) IMPLIES f1=f2))
```

Here is an example of specifying that an assertion is valid according to the export policy used in a model. The axiom `valid_precondition_calls` ensures that (a) all calls in a precondition are legal (according to the accessor list for each feature) and (b) all calls in the precondition are queries.

[4]The approximate equivalent of this information in UML 2.0's metamodel is contained in the notion of behavioral feature, and as such a similar approach could be used to construct this part of the UML metamodel.

```
valid_precondition_calls: AXIOM
  (FORALL (c:CLASS): (FORALL (f:FEATURE): member(f, class_features(c)) IMPLIES
       (FORALL (call:CALL): member(call, calls_in_pre(f)) IMPLIES
         QUERY_pred(f(call)) AND call_isvalid(f(call)))))
```

3.1.2 *Theory of Relationships.* The theory of relationships mimics the relationships cluster in the BON version; it defines the three basic relationships and the well-formedness constraints that exist in BON.[5] To express the relationships in PVS, we introduce a new nonempty type and a number of subtypes. As with abstractions, we mimic the inheritance hierarchy that was presented in Figure 7.

```
rel_names: THEORY
BEGIN
  % The abstract concept of a relationship.
  REL: TYPE+

  % Instantiable relationships.
  INH, C_S, MESSAGE: TYPE+ FROM REL
  AGG, ASSOC: TYPE+ FROM C_S
END rel_names
```

The `rel_names` theory is then used by the `relationships` theory. In BON, all relationships are directed (or targetted). Thus, each relationship has a source and a target, and these concepts are modeled using PVS functions.

```
relationships: THEORY
BEGIN
  IMPORTING rel_names, abstractions

  % Examples of the source and target of a relationship.
  inh_source, inh_target: [ INH -> STATIC_ABS ]
  cs_source, cs_target:   [ C_S -> STATIC_ABS ]
```

Now we can express constraints on the functions; once again; these correspond to invariants in BON and well-formedness constraints in OCL. We give one example of relationship constraints that inheritance relationships cannot be self-targeted.

```
% Inheritance relationships cannot be directed from an abstraction to itself.
inh_ax: AXIOM  (FORALL (i:INH): NOT (inh_source(i)=inh_target(i)))
```

The theory of relationships is quite simple because many of the constraints on the use of relationships are global constraints that can only be specified when it is possible to discuss all abstractions in a model (e.g., that there are no circularities in a chain of inheritance relationships). Thus, further relationship constraints will be added in the next section where we describe constraints on models themselves.

3.1.3 *The Metamodel Theory.* The PVS theory `metamodel` uses the two previous theories of abstractions and relationships to describe the well-formedness constraints on all BON models. Effectively, the PVS theory `metamodel` mimics

[5]In UML 2.0 terms, this theory would be used to implement parts of the dependency package in the UML superstructure.

the structure of the BON model in Figure 5, and captures the invariants on class *MODEL*.

```
metamodel: THEORY
BEGIN
IMPORTING abstractions, relationships

% A BON model consists of a set of abstractions.
abs:  SET[ABS]
rels: SET[REL]
```

Now we must write constraints on how models can be formed from a set of abstractions. The complete details are in Paige and Ostroff [2001]. We present two examples. The first constraint we write ensures that inheritance hierarchies do not have cycles; a similar constraint, written in OCL, appears in the UML 2.0 metamodel in Object Management Group [2004b]. We express this by calculating the *inheritance closure*, the set of all inheritance relationships that are either explicitly written in a model or that arise due to the transitivity of inheritance in BON.

```
inh_closure: SET[INH]

% Closure axiom #1: any inheritance relationship in a model is also
% in the inheritance closure.
closure_ax1: AXIOM
  (FORALL (r:INH): member(r,rels) IMPLIES member(r,inh_closure))

% Closure axiom #2: all inheritance relationships that arise due to
% transitivity must also be in the inheritance closure.
closure_ax2: AXIOM
  (FORALL (r1,r2:INH):
     (member(r1,rels) AND member(r2,rels) AND inh_source(r1)=inh_target(r2))
     IMPLIES (EXISTS (r:INH): member(r,inh_closure) AND
        inh_source(r)=inh_source(r2) AND inh_target(r)=inh_target(r1)))

% Inheritance relationships must not generate cycles.
inh_wo_cycles: AXIOM
  (FORALL (i:INH): member(i,inh_closure) IMPLIES
    NOT (EXISTS (r1:INH): (member(r1,rels) AND i/=r1) IMPLIES
       inh_source(i)=inh_target(r1) AND inh_target(i)=inh_source(r1)))
```

The second example is an axiom demonstrating a well-formedness constraint on clusters: all clusters in a model are disjoint or nested (again, a similar constraint appears in the UML metamodel with respect to packages). The third example shows that bidirectional aggregation relationships are forbidden. This is equivalent to the UML metamodel constraint that bidirectional compositions are forbidden.

```
% All clusters in a model are disjoint.
disjoint_clusters: AXIOM
  (FORALL (c1,c2:CLUSTER): (member(c1,abst) AND member(c2,abst)) IMPLIES
  (c1=c2 OR empty?(intersection(cluster_contents(c1),cluster_contents(c2)))))

% No bidirectional aggregation relationships are allowed.
no_bidir_agg: AXIOM
  (NOT (EXISTS (r1,r2:AGG): (member(r1,rels) AND member(r2,rels))
  IMPLIES (cs_source(r1)=cs_target(r2) AND cs_target(r1)=cs_source(r2)))
```

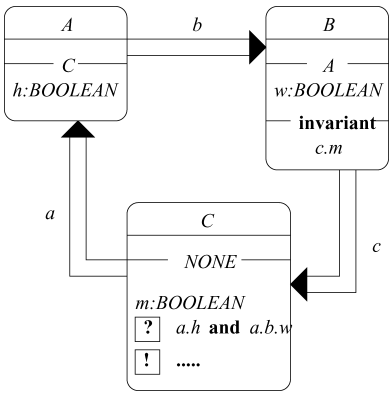


Fig. 10. Model conformance checking.

3.1.4 *Model Conformance Checking in PVS.* Model conformance, that is, checking the **sat** relation in Equation (1), resolves to proving a conjecture in PVS. The conjecture is that an encoding of a BON model (written as a set of PVS expressions) satisfies all axioms that capture metamodel well-formedness constraints. An identical approach could be applied directly to UML model conformance, that is, showing that a UML model satisfies the well-formedness constraints of its metamodel. This is illustrated by the following example. Consider the BON model in Figure 10.

We show how to check that this model does not conform to the metamodel of BON because there is a violation of the export policy of at least one class in the model. Note that m is a private feature of class C; thus the call $c.m$ in the invariant of B is illegal. Similarly, the call $a.b.w$ in class C is illegal in the precondition of m, because w is accessible only to the client A. We would like to show that this model does not conform to the BON metamodel. We will show that, as an example, the invariant of B is not well-formed. To prove that the model is not well-formed, we show that the class invariant for B is ill-formed by positing that the model in Figure 10 cannot exist. The full conjecture contains a number of terms that are not relevant to the proof (they can be found in Paige and Ostroff [2000]) but which would be included in a completely mechanical derivation of the conjecture. We only include terms relevant to the proof in this presentation due to space constraints.

```
info_hiding: THEORY
 BEGIN
  IMPORTING metamodel

  a, b, c: VAR CLASS
  h, w, m: VAR QUERY
  call1, call2, call3: VAR CALL

  test_info_hiding: CONJECTURE
    (NOT (EXISTS (a,b c: CLASS): EXISTS (h,w,m:QUERY):
     EXISTS (call1,call2,call3: CALL):
      member(c,accessors(h)) AND member(a,accessors(w)) AND
      empty?(accessors(m)) AND f(call1)=h AND f(call2)=w AND
      f(call3)=m AND member(call1,calls_in_pre(m)) AND
      member(call2,calls_in_pre(m)) AND member(call3,calls_in_inv(b))))
 END info_hiding
```

To prove the conjecture, we first skolemize three times, then flatten. We introduce the axiom `valid_class_invariant` and substitute class B and call *call*3 for the bound variables of this axiom. We use `typepred` to bring the type assumptions on m into the proof, and then one application of `grind` proves the conjecture automatically. The model is therefore invalid according to the well-formedness constraints of the metamodel.

Considerable expertise with PVS is necessary in discharging the conjecture, particularly in terms of selecting instantiation and skolem constants. However, strategies can be defined to automate parts of the process, though user guidance for selecting skolem constants is likely to always be necessary.

We should point out what happens with contracts when carrying out model conformance checking: all well-formedness constraints that are related to contracts are checked. These constraints effectively check the *static* well-formedness of contracts, for example, that the assertions are syntactically well-formed, functions that are called in contracts are accessible, etc. Nothing pertaining to *dynamic* (or behavioral) well-formedness is checked; this is one of the tasks for multiview consistency checking.

3.1.5 *Summary.* The advantages of expressing the BON metamodel in PVS are as follows.

—*Tool Support*. The PVS type checker and theorem prover can be used to verify and validate the metamodel, particularly for consistency, but also for correctness via proving that models satisfy the metamodel [Paige and Ostroff 2001].

—*Formality*. The semantics of the PVS specification of the metamodel is formally defined, and this semantics is implemented in the PVS toolkit. This provides stronger guarantees as to the soundness of the PVS specification and, therefore, the metamodel itself.

—*Completeness*. All aspects of the BON metamodel can be expressed in PVS.

The disadvantages are as follows.

—*Understandability*. The PVS specification is more difficult to understand than the BON specification of the metamodel. PVS is not object-oriented so it is more difficult to provide an architectural view of the metamodel that omits constraint detail. As well, the only structuring mechanism in PVS is the `imports` statement; thus, PVS specifications are generally flatter than object-oriented specifications since much of the hierarchical structure in the metamodel presented in the previous section is captured in PVS using type hierarchies which do not present themselves in the overall theory structure. This can substantially impede understanding of large specifications like the BON metamodel.

—*V&V*. Validating the metamodel in PVS corresponds to encoding BON models as PVS constants and proving that the models satisfy (or fail to satisfy) the axioms in the metamodel. This process can be partly automated, but it is certainly not fully automatic. Additional proofs in Paige and Ostroff [2001] demonstrate that in many instances user intervention and guidance

is needed. Further experiment on validation is continuing via the PVS ground evaluator which allows simulation.

In general, the PVS version of the metamodel should be preferred to the BON version in terms of available automated tool support for expressing the metamodel and validating it, and for its completeness. However, it falls short in understandability and general usability. As well, there are challenges in using the PVS version for multiview consistency checking, which we discuss in Section 4.

3.2 Metamodel Specification in Eiffel

A key advantage of the BON metamodel specified in PVS is that tools can be used to validate and verify it. However, these tools are complex, expensive to use, and typically require user interaction and intervention. We now present a formal specification of the BON metamodel in Eiffel. Eiffel, while a programming language, can also be used as a *declarative* specification language via its *agent* technology. We illustrate this technology throughout the following section. It is important to emphasise that we are not simply writing Eiffel programs to encode models, metamodels, and well-formedness constraints. Rather, we use Eiffel to declaratively specify (albeit in an executable form) these artifacts.

We do not attempt to reformulate the PVS version of the metamodel in Eiffel rather we will present an Eiffel specification that conforms to the best practices of using Eiffel and which attempts to exploit Eiffel's existing tool support (particularly compilers, debuggers, testing frameworks, and class libraries) as well as its language facilities.

Eiffel is an object-oriented language, and as such the structure of the metamodel specification in Eiffel will directly reflect the structure of the metamodel in BON. This is a result of the seamless integration of BON and Eiffel that has been designed into the two languages. As we did with PVS, we present the metamodel specification selectively, and attempt to give the flavor of applying Eiffel for this purpose.

3.2.1 *Expressing the Metamodel Structure.* Each class in the BON diagram in Section 3.1 is expressed directly as an Eiffel class; this is straightforward and produces a set of classes with routines, attributes, routine signatures. Each relationship in the BON diagram in Section 3.1 is expressed directly as an Eiffel relationship or as an attribute: BON inheritance is expressed as Eiffel inheritance, associations are expressed as Eiffel reference attributes, and aggregations are expressed as expanded attributes. Two additional elements need to be added: encoding BON assertions (pre and postconditions and class invariants) as Eiffel agents, and the second is *infrastructure* needed to initialize objects and provide setter and getter routines. The user of the metamodel in Eiffel need not know about this infrastructure—a single routine call is provided to set up the objects. This infrastructure is essential as Eiffel is a programming language and objects must have memory allocated for them; in contrast, no allocation is explicitly necessary in a specification language like PVS.

Much of this infrastructure is added as private features of class *MODEL*. The added routines are as follows.

—A constructor (creation procedure) *make* which allocates memory for the set-based data structures used to encode models. This constructor implements a Factory Method [Gamma et al. 1995] which calls default constructors on each attribute.

—Accessor and mutator routines for each attribute.

—A *prepare* routine which ensures that all data structures are initialized. This routine is invoked by the user after they have constructed their model, that is, when they wish to test the metamodel and check conformance. It effectively provides a facade for initializing the model, hiding the infrastructure required by Eiffel from the user. This routine in turn invokes a number of private routines that set up different data structures for use in conformance and consistency checking.

Additional attributes were added as private features in order to simplify some agent expressions; see Paige et al. [2004] for some details.

3.2.2 *Expressing Assertions.* This is the most challenging part of expressing the metamodel in Eiffel and requires capturing BON assertions using Eiffel agents.[6] BON's assertion language is more expressive than Eiffel (because in BON quantifiers can be written on arbitrary domains, whereas in Eiffel agents must be applied to finite data structures). In general, to translate a BON assertion into Eiffel, we can introduce a model of BON's *SET* type, following Meyer [2003]. However, given that the BON metamodel makes use of finite sets, and the metamodel assertions are computable, all assertions in the BON version of the metamodel can be translated to Eiffel. Assertions are expressed as follows.

—Each invariant clause in a class in the BON metamodel is translated to a boolean-valued function in Eiffel. For example, the clause *no_inh_cycles* in class *MODEL* in the BON diagram is translated to a boolean function in the Eiffel class model, and a call to the function appears in the invariant for *MODEL*.

—Each pre and postcondition is encoded as a *predicate agent*. Thus, the Eiffel class representing a *ROUTINE* includes two attributes, as follows.

$$pre : PREDICATE\ [ROUTINE, TUPLE\ [ANY]]$$
$$post : PREDICATE\ [ROUTINE, TUPLE\ [ANY, ANY]]$$

The precondition is encoded as a predicate agent that takes a tuple, consisting of the state of the agent as argument. The state includes arguments for the routine, as well as any attributes used by the routine. The postcondition takes a pre and poststate as arguments.

[6]Obviously we could encode assertions indirectly and programmatically, but we choose to use agents to make it easier to validate that the agent expressions correctly implement the original BON assertions.

```
no_inheritance_cycles: BOOLEAN is
do
  Result := closure.for_all((i1:INHERITANCE): BOOLEAN  -- iterate over closure once
  do
   Result := closure.for_all((i2: INHERITANCE): BOOLEAN -- iterate over closure twice
   do
     -- return true iff two selected inheritance relationships do not
     -- form a cycle
     Result := not (i1.source=i2.target and i1.target=i2.source)
   end)
  end)
end
```

Fig. 11. Invariant: no cycles in the inheritance graph.

—Each occurrence of the quantifiers ∀ or ∃ in BON is translated to the Eiffel agent `for_all` or `there_exists`, respectively. (In general, each BON assertion must be computable to translate to Eiffel.)

—Bound variables introduced in postconditions and invariants to represent intermediate state are instead represented using local variables (we show an example shortly). These local variables are local to an agent, and as such are not accessible or visible to clients of a routine, thus preserving encapsulation.

We provide three examples of metamodel constraints: the constraint establishing cycle-free inheritance graphs in a class diagram, the constraint expressing that there are no bidirectional aggregations, and the constraint establishing model covariance, that is, that any redefined routines that appear in the class diagram modify the routine signatures covariantly (as required by Eiffel and BON). The first two constraints appear in the UML metamodel (as discussed in Section 3.1), but the third does not and is distinctive to BON.

3.2.3 *No Cycles in the Inheritance Graph.* In the BON metamodel, this was captured by defining a private attribute, *closure*, which contains the transitive closure of the graph defined by inheritance relationships in the class diagram. The same approach is used in Eiffel. The invariant in Eiffel class *MODEL* includes a call to the boolean routine shown in Figure 11. This should be contrasted with the PVS specification of the same constraint in Section 3.2.

In this invariant, there are two agents iterating over the closure set. The final *Result* line in the second (inner) agent says that it is not the case that there are two inheritance relationships where the source of one is the target of the other (and vice versa). If this condition is violated for any pair of classes in the closure, there is a cycle. This is a declarative rather than operational specification of the cycle-free property.

The closure set is calculated by an invocation of the *prepare* routine. In this routine, an additional agent iterates across the set of all inheritance relationships and adds any implicit inheritance relationships that arise due to transitivity to the closure set.

3.2.4 *No Bidirectional Aggregations.* This constraint is similar to the one found in UML on part-of relationships, that is, that an object cannot recursively contain itself. The specification in Eiffel is as shown in Figure 12.

```
no_bidirectional_aggregations: BOOLEAN is
do
 -- iterate over aggregations in model
 Result := not aggregations.there_exists((a1: AGGREGATION): BOOLEAN
 do
  -- iterate over aggregations in model again
  Result := aggregations.there_exists((a2: AGGREGATION): BOOLEAN
  do
   -- return true iff the two selected aggregations are different
   -- but form a loop
   Result := (a1/=a2) implies
        (a1.source=a2.target and a1.target=a2.source)
  end)
 end)
end
```

Fig. 12. Invariant: no bidirectional aggregations.

This constraint has a structure similar to that in Figure 11. It requires two agents iterating over the set of all aggregations, this time applying a function to check if there are any cycles in the aggregation graph. Note that this time a transitive closure operation does not need to be applied since aggregations in BON are not transitive.

3.2.5 *Model Covariance.* The last example is more complex and shows how to capture the covariant redefinition concept in Eiffel using agents. Recall covariant redefinition from Meyer [1992]. Consider the following routine in a class A:

$$r(x : X) : Y$$

and suppose r is redefined (overridden) in class B which is a descendent of A. The new signature of r in B is

$$r(x : U) : W$$

For the redefinition to be valid in BON, class U must be a (not necessarily proper) descendent of X, and W must be a (not necessarily proper) descendent of Y. We capture this as follows in Eiffel. First, we define a routine *subtype* which, given two classes, returns true if and only if the second is a subtype (a descendent) of the first. This can be easily expressed using an agent iterating over the transitive closure of the inheritance graph. We can then define a routine *covariant* as in Figure 13. The routine returns true if and only if the second argument covariantly redefines the first.

The definition of *covariant* appears to be complex but can be easily explained. The first line generates an anonymous integer sequence and iterates across it using an agent. The agent checks that each pair of parameters in the feature arguments are covariant, and then, if the feature is also a query (tested by the assignment attempts), it is tested to determine whether the result type of the functions is covariantly redefined.

This function can then be used in a class invariant of *MODEL* which checks that each redefined feature of each class is covariantly redefined. Note that this must be a property of the *MODEL* and not the metaclass *CLASS*, as it requires

```
covariant(f1: FEATURE; f2:FEATURE): BOOLEAN is
require f1/=Void and f2/=Void;
local q1, q2: QUERY
do
  -- iterate over all parameters of the feature
  Result := (1|..|f1.parameters.count).for_all((i:INTEGER): BOOLEAN
    local c1, c2: CLASS
    do
      -- select the i-th parameters from the two features
      c1 ?= f1.parameters.i_th(i).item(2);
      c2 ?= f2.parameters.i_th(i).item(2);
      if(c1/=Void and c2/=Void) then
        -- check that the second parameter is a subtype of the first
        Result := subtype(c1,c2)
      end
    end);

  -- if the features are functions (with return types) then
  -- check that the return types are covariantly redefined.
  q1 ?= f1;
  q2 ?= f2;
  if(q1/=Void and q2/=Void) then
    Result := Result and subtype(q1.q_result,q2.q_result)
  end
end
```

Fig. 13. Routine: covariant redefinition specified in Eiffel.

access to the inheritance graph: a routine may covariantly redefine another routine from an indirect parent. To express this invariant, we use an agent that iterates across the set of all classes in *MODEL*, pulling out each redefined feature. It then checks that the redefinition obeys the covariance specification in Figure 13.

3.2.6 *Model Conformance in Eiffel.* The obvious way to carry out model conformance checking in Eiffel is to simulate a model against the metamodel encoding previously presented. Thus, a BON model is encoded in Eiffel as a *reference structure* (consisting of objects and references between objects), and, once constructed, the metamodel rules are executed to determine whether the reference structure is a valid instantiation of the metamodel. If a rule evaluates to false, then the runtime system of Eiffel will inform the user as to which well-formedness rule has failed.

It is most convenient, though not required, to encode the model using a unit test and unit testing framework. This allows multiple models to be encoded simultaneously and run automatically. The testing framework that we use, ETest, also provides documentation facilities indicating which well-formedness rules have failed (if any) and where. It is important to note that the model conformance-checking process is fully automated in Eiffel.

Consider Figure 14, which shows a class diagram possessing a cycle in its inheritance graph (labels are for reference only). This model is not well-formed and should violate the invariant *no_inheritance_cycles* of class *MODEL*.

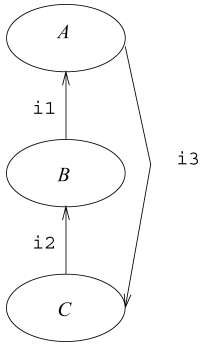


Fig. 14. Model conformance example: cycles in an inheritance graph.

```
CLASS VALIDATE_METAMODEL inherit UNIT_TEST creation make
feature ANY
  make is
  do -- initialise and prepare report end

  no_inh_cycles: BOOLEAN is
  local a,b,c: E_CLASS; i1, i2, i3: INHERITANCE; m: MODEL
  do
    create a.make("A"); create b.make("B"); create c.make("C");
    -- generate additional model elements here
    m.add_class(a); m.add_class(b); m.add_class(c);
    m.add_inheritance_rel(i1); m.add_inheritance_rel(i2);
    m.add_inheritance_rel(i3);

    -- initialise infrastructure and data structures
    m.prepare; Result := true;
  end
end -- VALIDATE_METAMODEL
```

Fig. 15. Unit test for metamodel validation.

By encoding the model as a unit test, we can easily simulate the model against the metamodel to show that it is not well-formed. This will give as a fully automated way to check the **sat** relation in Equation (1). An excerpt from the unit test that can be used to simulate the BON model in Figure 14 against the metamodel is shown in Figure 15.

The unit test in `no_inh_cycles` can be explained as follows. First, entities for each element of the model are declared and created; relevant attributes of these entities are then populated, for example, the name of each class, the source and target of each inheritance relationship. These elements are all added to the model m, and a call to *m.prepare* constructs the infrastructure for the model. On this call, the clause *no_inheritance_cycles* fails with an object configuration showing (e.g.) that the transitive closure of the graph contains an inheritance relationship from B to A and also from A to B.

Executing the unit test generates a runtime exception which is caught by the exception handling facilities built into ETest. This result is documented in the HTML report generated by ETest.

Once again, static well-formedness constraints related to contracts (e.g., that assertions are syntactically well-formed) are checked during model conformance checking. No behavioral well-formedness is checked at this stage. We discuss this in more detail in Section 4.

3.2.7 *Summary.* The advantages of using Eiffel for expressing the BON metamodel are as follows.

—*Understandability*. The structure of the Eiffel system is directly mapped from the semiformal specification of the metamodel in BON, that is, it is class-based. Eiffel tools can be used to project different views of the Eiffel system, for example, to hide class interface details so that only the structure of the metamodel, that is, classes and relationships, can be viewed.

—*Tool Support*. Standard programming tools can be used to construct, execute, verify, and validate the metamodel, for instance, compilers, IDEs, class libraries, debuggers, and unit testing frameworks. The feedback provided during construction and verification is familiar to programmers, and as such a new conceptual model of development does not need to be learned.

—*Testing*. The metamodel is easy to test, using standard unit, regression, and system-level techniques. In this particular case, the ETest unit testing framework was applied. This framework, tailored to testing in Eiffel, made it possible to target specific well-formedness constraints in the metamodel for testing and identified failures of specific constraints during the testing process.

The disadvantages to using Eiffel are listed below.

—*Implementation Detail*. It is necessary to include implementation detail in the Eiffel version, specifically for initializing data structures (i.e., object allocation and cloning), all of which can be omitted in the BON and PVS versions. Fortunately, much of this detail can be hidden using Eiffel's information hiding facilities, but it is necessary to include it: someone using the metamodel can ignore this implementation detail, but someone wanting to change the metamodel (or develop a new metamodel) will need to use it.

—*Completeness*. All well-formedness constraints in the BON metamodel can be expressed in Eiffel, though one tiny subset of them—consisting of four constraints—must be slightly restricted. The restriction arises with capturing contracts, that is, pre and postconditions of routines. The Eiffel version of the metamodel does capture contracts, using predicate agents to encode each contract, but these are restricted to the expressiveness constraints of Eiffel itself, that is, the contracts must be computable and expressible within Eiffel's boolean expression syntax. Thus, for example, BON quantifiers over arbitrary types cannot be expressed in Eiffel. In general, this is not a substantial restriction for metamodeling, because agents can be used to capture quantified expressions over finite structures, but it does mean that the PVS version of the metamodel expresses more.

This raises the question of whether a language used for metamodeling should be more expressive than the languages being metamodeled. Clearly there are trade-offs: while PVS is more expressive than BON (and Eiffel)

Table II. Informal Comparison of Metamodeling Approaches (• = least, ••• = most)

Quality Factor	Metamodeling Approach		
	BON	PVS	Eiffel
Understandability	•••	•	•••
Correctness	•	•••	••
Completeness	•••	•••	••
Maintainability	•••	•	•••
Construct via tools	•••	•	••
V&V via tools	•	•••	•••

and can capture all constraints, it does not provide fully automated model conformance checking or type checking, where Eiffel does. The question must be answered in the context of how the metamodel is intended to be used.

We will return to the issue of encoding contracts in the next section when we discuss multiview consistency checking. It is here that one of the key distinctions between using PVS and Eiffel becomes apparent.

3.3 Overall Comparison

While, it is difficult to objectively compare all aspects of the metamodels just presented, we attempt a comparison in terms of the factors described and present a relative comparison. The results are summarized in the Table II; we include the specification of the BON metamodel from Section 2.3 for completeness and interest.

The comparisons are relative, and a substantial degree of subjectivity is inherent in the matrix. However, the table is meant to provide a coarse-grained comparison of the approaches, so as to give metamodelers some approximate guidance. The quality factors for comparison are, in more detail, as follows.

—*Understandability* is the degree to which the metamodel specification is comprehensible to a suitable, experienced tool-builder and metamodel user. Understandability assumes a reasonable amount of expertise in modeling, not formal methods or metamodeling. Clearly, the BON version of the metamodel will be understandable to the largest number of developers (engineers familiar with UML will have no difficulty learning BON). Similarly, the Eiffel version of metamodel will also be understandable to many developers as it is written in a programming language; admittedly, Eiffel's agent syntax does require some adjustment, but it is effectively just an iterator, and as such is compatible with the standard object-oriented programmer's toolkit. The PVS version of the metamodel requires substantial formal methods expertise and is the most challenging to understand of the three.

—*Correctness* is the degree to which correctness of the metamodel specification can and has been checked, either automatically or semiautomatically. Note that a score of ••• does not imply that the specification is unchecked, merely that less checking is feasible or has been carried out than with the other approaches. The PVS version of the metamodel has been partly checked via proof, as reported in Paige and Ostroff [2001], and this process is repeatable.

The Eiffel version has been partially tested and additional testing can be carried out. The BON version has been constructed via BON-compatible CASE tools, but beyond the syntax and partial lightweight semantic checking that these tools support, no additional correctness checking has been carried out.

—*Completeness* is the degree to which the metamodel description captures all the well-formedness constraints of the BON language. We claim that the BON and PVS descriptions are complete (though this has not been proved). The Eiffel description is incomplete since Eiffel's predicate agent syntax is restricted to the use of boolean expressions that can be written in Eiffel.

Though we claim that both the BON and PVS descriptions are complete, we prefer the PVS description in this respect because it is possible to use the PVS theorem prover to detect incompleteness. An example of this was reported in Paige and Ostroff [2001], where, because a requirement was not captured, a conjecture expressed in PVS turned out to be impossible to prove (though the PVS prover itself did not indicate that the conjecture was unprovable). Of course, there are other reasons to prefer the BON (or Eiffel) descriptions over the PVS version.

—*Maintainability* is the degree to which the metamodel specification supports extension, refactoring, and wholesale modification. The BON and Eiffel versions have an advantage due to their component-based style of specification and their use of patterns. However, even using good object-oriented design practice may be insufficient for promoting extensibility and wholesale modification, for instance, as suggested with the pUML proposal for UML 2.0 [Evans et al. 2005], which recommended adding templates and frameworks as first-class language concepts.

—*Constructed via Tools* is the degree to which tools are available to assist in constructing the specification (e.g., diagramming tools, smart editors with syntax checking and highlighting, autocode generation). The BON and Eiffel versions were constructed using CASE tools and integrated development environments, whereas the PVS version was constructed using a text editor.

—*V&V via Tools* is the degree to which tools are available to assist in verifying and validating the correctness of the specification. With PVS this can be carried out via theorem proving, whereas unit testing can be used for Eiffel. No such support exists for BON directly. Note that while both Eiffel and PVS support verification and validation, the PVS version should be considered more powerful since it makes use of proof as opposed to testing, while also supporting simulation via its ground evaluator.

4. METAMODEL-BASED MULTIVIEW CONSISTENCY CHECKING

Metamodels are intended to be used indirectly by modelers applying the modeling language, and directly, by tool builders constructing diagramming tools for modelers. The tool builder, in particular, will need to be able to understand the metamodel, and he guaranteed that it is correct (since they will not want to substantially modify their tools due to corrections made to the metamodel). They will also need extensibility capabilities so that language-revision processes can be accommodated and directly linked to tool-revision processes. The relative

importance of these metamodel quality factors can be used in a specific context to decide the best metamodeling approach to use for a specific language.

A metamodel can form the basis of any approach to multiview consistency checking: a metamodel captures the essential concepts in a model and defines (or can be used to define) the rules that establish that different views of the model contain no contradictions. The rules for multiview consistency checking (which are expressed within a metamodel) must be correct, understandable, and extensible as well, for them to be of value to tool builders.

One particular aspect of a tool builder's remit is to establish capabilities in their tool for multiview consistency, either by providing consistency-checking capabilities, or model-synchronization capabilities. In a language like UML, where a large number of diagrams representing the same system from different perspectives can be produced, multiview consistency is a critical and challenging problem. There are several approaches that can be taken for metamodel-based view consistency checking, but they can be contrasted in terms of two desirable criteria.

—*Automation* is the extent to which the views can be checked for consistency by automated measures. Clearly, the more views that are present in the metamodel, the more challenging it is to provide an automated scheme. All UML tools and approaches to multiview consistency checking are only semiautomatic. The Eiffel-based approach we present here is fully automatic.

—*Completeness* is the degree to which all multiview consistency constraints can be captured, based on the metamodel specification available. For example, we will see that the Eiffel-based metamodeling approach suffers from incompleteness in its provisions for view consistency, while providing a fully automatic approach. All UML tools are incomplete in terms of their support for multiview consistency checking since they all support only a limited set of views (e.g., class diagrams and statecharts, like the Executable UML tools [Mellor and Balcer 2002]), or do not fully consider the metamodel constraints, particularly those involving contracts (discussed in more detail later).

Based on the metamodel specifications from the previous section, we now present, compare, and contrast two distinct approaches to multiview consistency checking in terms of the factors just discussed, in the context of BON's class and dynamic diagrams. A novelty of the two approaches presented here is that they include support for checking contracts for consistency (albeit in quite different ways).

Consider the BON dynamic diagram shown earlier in Figure 3. Suppose that each of messages 1–4 have been bound to routines in a BON class diagram. In order to check that the dynamic diagram is consistent with a class diagram, the following four constraints must be checked (there are additional constraints in MVCC, but these are essential ones).[7]

(1) *Object-Class*. Each object appearing in the dynamic diagram must have a corresponding class in a class diagram which acts as the object's type.

[7]Many of these constraints appear in consistency checking in UML [MODELWARE 2005].

This constraint links the abstractions appearing in the two different BON views. Note that without this constraint, an object may have a type that does not appear in a system model (thus making it impossible to compile and implement the model). Such a constraint also appears in UML 2.0, linking class and communication diagrams.

(2) *Message-Feature*. Each message in the dynamic diagram is bound to a routine, and a call to that routine is permitted based on the list of accessors provided with each routine. (To paraphrase, if a message in a dynamic diagram corresponds to a call *or* for object o and routine r, then r must be exported to the client class, that is, the class of the object that sends the message to o.) A similar but not identical constraint appears in UML 2.0: messages may be bound to operations in UML.

(3) *Message-Class*. Each routine bound to a message must actually belong to the target class of the message (i.e., routines that are called must exist). This ensures that if a message is sent from one object to another, there is a connection between the two objects. According to Gao [2004], connections can arise due to direct associations between the objects' classes, can be due to indirect associations (i.e., a chain of more than one association), or can be due to compositions of inheritance relationships and associations. In this article, we consider only direct associations and direct inheritance relationships; see Gao [2004] for extensions to indirect associations and more complex relationships. This constraint also appears in UML 2.0 for intramodel consistency checking.

(4) *Contract-Consistency*. The constraint in (2) establishes that each message in a dynamic diagram corresponds to a routine call. The routines that are called must be enabled (i.e., their preconditions must be true) for the dynamic diagram to be consistent with a class diagram. A precondition, can only be true if the sequence of previous calls to routines established a system state that satisfies the precondition, that is, the postconditions of previous calls combine to enable the current precondition of interest. To check this, an initial state, *init*, must be provided (by the developer), and *init* must enable the first message in the dynamic diagram. Successive messages must be enabled by the sequence of message calls that precede it. This constraint does not appear in the UML 2.0 metamodel, but it is discussed in Krishnan [2000] in the context of OCL and UML.

These four constraints turn out to be fundamental in establishing that the class diagram views and the dynamic diagram views are consistent; the constraints will need to be implemented by the multiview consistency approaches that we now present. Once again, we will present approaches to checking these constraints that follow the best practices of using PVS and Eiffel; we will discuss alternative approaches as appropriate.

4.1 A PVS Approach to Multiview Consistency

Building on the PVS specification of the BON metamodel in Paige and Ostroff [2001], the PVS theories were extended in Paige et al. [2002, 2003b] in three

directions in order to support multiview consistency and the four constraints discussed previously:

—to include additional features to project different views of a system. While this information was present in the original metamodel from Paige and Ostroff [2001], it was not explicitly easily accessible;

—to include constraints to establish that projected views were consistent;

—to include the semantics of routines in order to carry out MVCC checking involving pre and postconditions.

Referring to the rules discussed in the previous section, (1)–(3) are reasonably straightforward to capture in PVS once the aforementioned projection functions are defined. These functions are not difficult to define [Paige et al. 2003b] for details. Formalizing the notion of routine specification and the corresponding consistency constraint for (4) is much more challenging. The complication does not arise in expressing a routine specification directly, but in combining routine specifications: PVS requires explicit specification of a function's domain (possibly using an uninterpreted type) in order to support type checking, that is, a routine specification's state must be formally specified. The formulation of routine specifications is therefore aimed at being able to (sequentially) compose them. The formalization of specifications of a routine requires a new type, `SPECTYPE`, which is a record containing the initial and final state variables of a specification along with the value of the specification; initial and final state are sets of entities. The functions `oldstate` and `newstate` produce the entities associated with a routine (given the class in which the routine arises), specifically the parameters, local variables, and accessible attributes. It is also necessary to introduce a new type for specifications so that the frame of a specification can be expressed.

```
SPECTYPE: TYPE+ =
 [# old_state: set[ENTITY], new_state: set[ENTITY],
    value: [ set[ENTITY], set[ENTITY] -> bool ]      #]

 oldstate, newstate: [ ROUTINE, CLASS -> set[ENTITY] ]
```

A specification can now be defined in terms of the new type.

```
spec: [ ROUTINE, set[ENTITY], set[ENTITY] -> SPECTYPE ]

spec_ax: AXIOM
(FORALL (rou1:ROUTINE): (FORALL (c:CLASS):
  (member(rou1,class_features(c)) IMPLIES
    (spec(rou1,oldstate(rou1,c),newstate(rou1,c)) =
       (# old_state := oldstate(rou1,c), new_state := newstate(rou1,c),
          value := (LAMBDA (o:{p1:set[ENTITY] | p1=oldstate(rou1,c)}),
                    (n:{p2:set[ENTITY] | p2=newstate(rou1,c)}):
                    feature_pre(rou1,o) IMPLIES feature_post(rou1,o,n)) #)))))
```

The `spec_ax` axiom states that, for a routine, the prestate and poststate of a specification are those of the routine, and the value of the specification is a function from pre and poststate to a boolean where the boolean is true if and only if the precondition implies the postcondition.

We can now express the view-consistency constraint (4) in PVS; this is challenging and has two parts. The first part, enabling the first message in the dynamic diagram, can be done as follows. *init* is translated to a function mapping a model and a class (which should be the class from which execution begins) to a boolean. The enabling of the first message is formalized as an axiom.

```
init: [ MODEL, CLASS -> bool ]

views_consistent_ax1: AXIOM
 (FORALL (mod1:MODEL): FORALL (c:CLASS):
  LET
   loc_spec:SPECTYPE = (spec(init(mod1)(c),oldstate(init(mod1)(c)),
                             newstate(init(mod1)(c)))  IN
  value(loc_spec)(old_state(loc_spec),new_state(loc_spec)) IMPLIES
  feature_pre(calls_model(mod1)(0),
              oldstate(calls_model(mod1)(0),
              object_class(msg_target(sequence_model(mod1)(0))))) )
```

A local variable is declared, constructing a specification for the initializing predicate `init`. Then, it is stated that the initial state must imply the prestate of the first message.

The second part is even more challenging. The complexity lies in formalizing the definition of sequential composition: an explicit specification of the state of a routine is required so as to capture the frame of each specification and to be able to define an intermediate state. Sequential composition can be formalized in PVS as follows, using function `seqspecs`. It takes as argument two variables of type `SPECTYPE` and returns a `SPECTYPE` result, representing the sequential composition of the arguments.

```
 seqspecs(s1,s2:SPECTYPE): SPECTYPE =
    (# old_state := old_state(s1),
       new_state := new_state(s2),
       value := (LAMBDA (o:{p1:set[ENTITY] | p1=old_state(s1)}),
                         (n:{p2:set[ENTITY] | p2=new_state(s2)}):
           (EXISTS (i: set[ENTITY]): value(s1)(o,i) AND value(s2)(i,n)))
    #)
```

`seqspecs` must be lifted to apply to a finite sequence of specifications in order to formalize constraint (4). This is expressed as recursive function `seqspecsn`. A `MEASURE` must be provided in order to generate proof obligations for ensuring termination of recursive calls.

```
seqspecsn(seq1:{f:finseq[SPECTYPE]|length(f)>=1}): RECURSIVE SPECTYPE =
  IF length(seq1)=1 THEN seq1(0)
  ELSIF length(seq1)=2 THEN seqspecs(seq1(0),seq1(1))
  ELSE seqspecs(seq1(0),seqspecsn(^(seq1,(1,length(seq1))))) ENDIF
  MEASURE
   (LAMBDA (seq1:{f:finseq[SPECTYPE]|length(f)>=1}): length(seq1))
```

To complete the PVS formalization of constraint (4), it is helpful to define a function to convert a sequence of messages into a finite sequence of `SPECTYPE`s. This function, `convert`, extracts the routines from the messages and produces specifications from them by repeated application of function `spec`. Its details can be found in Paige et al. [2002].

Now the remaining view-consistency constraint can be formally expressed in PVS.

```
views_consistent_ax2: AXIOM
(FORALL (mod1:MODEL): FORALL (c:CLASS):
  (FORALL (i:{j:nat|0<j & j<length(calls_model(mod1))}):
  LET
  loc_spec:SPECTYPE =
    seq(spec(init(mod1)(c),oldstate(init(mod1)(c)),
             newstate(init(mod1)(c)),
         (seqspecsn(convert(sequence_model(mod1)^(0,i-1))))
  IN
  (value(loc_spec)(old_state(loc_spec),new_state(loc_spec)) IMPLIES
  feature_pre(calls_model(mod1)(i),
              oldstate(calls_model(mod1)(i),
              object_class(msg_target(sequence_model(mod1)(i))))))))
```

The structure of this axiom is similar to the axiom establishing that the first message is enabled by the initial state. This axiom first declares a local variable, `loc_spec`, which is the result of sequentially composing the first i specifications in messages in the model. This specification must then imply the precondition of the routine of message $i + 1$ in the model.

4.1.1 *Using the PVS Theories.* To use the PVS theories for proving view consistency, a BON model can be specified as a PVS conjecture, following the approach presented in Paige and Ostroff [2001]. These conjectures effectively posit that the model can exist. They must therefore satisfy the multiview consistency constraints as specified in the metamodel. PVS can then be used to import the view-consistency axiom previously presented, and one can then attempt to prove or disprove that the axiom is satisfied by the models. This is identical to the approach used in demonstrating conformance, that is, that Equation (1) is satisfied.

In general, it is typically easier to attempt to prove that a model does not satisfy the view-consistency constraint. This is because it means that the BON models can be expressed as a nonexistence conjecture, thus allowing automatic skolemization to be used in simplifying the conjecture.

A lengthy example demonstrating this approach is in Paige et al. [2003b] where a model consisting of a class and a dynamic diagram is shown to be consistent against the metamodel and multiview consistency constraints. The PVS proof is long and laborious (which is why we omit it) and requires a clever selection of skolem constants in order to succeed. Other examples and case studies that we have carried out suggest to us that a general strategy for using PVS to carry out view consistency checking is possible, but, in the most nontrivial cases, it will be extremely difficult to automatically choose skolem (or instantiation) constants as part of this process, and it seems that user intervention is unavoidable.

Succeeding with multiview consistency checking in PVS requires substantial expertise in not only using the PVS theorem prover, but also in interpreting the proof obligations and subgoals generated by PVS. These are typically difficult to interpret, in part because of PVS's syntax, but, in this particular application domain, they appear to be more difficult to understand because they refer to metalevel constraints.

4.2 An Eiffel Approach to Multiview Consistency

In Section 3, the BON metamodel was specified in Eiffel, using the programming language's agent technology. This resulted in a declarative and executable specification of the well-formedness constraints for BON. The metamodel encompassed both BON class diagrams (with their concomitant features and relationships) as well as BON dynamic diagrams (i.e., objects, clusters of objects, and messages). Based on the metamodel, three of the four multiview consistency constraints can be specified and directly expressed in Eiffel; the fourth is best handled indirectly (though we comment on handling it entirely within the Eiffel metamodel in the sequel).

Unlike the PVS version of the metamodel, the multiview consistency constraints in Eiffel are distributed over the specification. That is, constraints are expressed as invariants of classes; in the PVS version, the constraints were axioms belonging to the metamodel theory. Some constraints in the Eiffel version will therefore apply directly to classes, messages, and objects, while others will apply to the *MODEL* class as a whole.

An example of a consistency constraint that applies directly to a message is constraint *message-feature*, constraint (2), described earlier. This multiview consistency constraint is captured in the invariant of class *MESSAGE* as follows. First, the invariant of *MESSAGE* includes the following boolean expression.

```
bound_routine /= Void implies access_granted
```

which states that if the message is bound to a routine then access must be granted to that routine to the invoking class. The definition of `access_granted` is given below.

```
access_granted: BOOLEAN is
  local o, p: E_OBJECT; ec: E_OBJECTCLUSTER;
  do
    o ?= source
    if (o/=Void) then
      p ?= target
      if (p/=Void) then
        Result := bound_routine.accessors.has(o.static_type) or
           bound_routine.accessors.has(bound_routine.any_accessor)
      else
        ec ?= target
        Result := ec.contains.there_exists( agent ecce(?) )
      end
    end
  end
```

The routine extracts the source and target of the message, using Eiffel's assignment attempt operator (similar to casting in Java). If the target is an object, then it ensures that the class of that object has given permission to the invoking class to call the routine. If the target is a multiobject (i.e., a cluster of objects), then an agent is invoked to check that some object in this set provides access.

Multiview consistency constraints (1) and (3) can also be specified in Eiffel in a similar way. Moreover, checking that constraints (1)–(3) hold resolves

to model conformance checking; an example in Section 4.2.2 demonstrates this.

Constraint (4) is challenging, as it involves routine pre and postconditions. The Eiffel metamodel already captures routine pre and postconditions using predicate agents (see Section 3.2.2). There are several approaches for checking this constraint.

(1) *Mimic the PVS approach,* that is, simulate theorem proving in Eiffel. Such an encoding would make use of predicate agents and provide a way to evaluate sequential compositions of agents. Of course, the limitation with this is that only those predicates that can be expressed in Eiffel's agent syntax can be captured.

    We have experimented with this approach. Encoding sequential composition of agents can be done but it is nontrivial to write and is particularly difficult to use. Users of the metamodel must reexpress contracts at the metalevel, making the contracts difficult to write, debug, and understand. The metalevel encoding of contracts, in particular, means that users of the metamodel must explicitly understand and use the metalevel encoding of BON variables, objects, pointer dereferencing, and intermediate states that are expressed in Eiffel.

    Given this encoding, contract consistency can be checked but not fully automatically. Moreover, the results are less informative than with PVS. What is produced with this approach is a sequence of agents which can then be invoked on an initial state (conceptually identical to the initial state provided in the PVS approach). This produces a *true* or *false* result, whereas with PVS one also obtains a set of undischarged proof obligations when the constraint cannot be proven. This suggests to us that, while the approach provides partial automation, easier-to-use approaches should be developed. This is not the best way to use Eiffel.

(2) *Reflection.* Make use of reflection techniques to extract pre and postconditions from the Eiffel metamodel and then make use of an external theorem prover in much the same manner as in Section 4.1. This approximate approach motivates the work in Taligheni [2004].

(3) *Exploit Eiffel's executability.* The PVS approach trades off completeness with automation: MVCC is not fully automatic, but all constraints can be checked. One of Eiffel's strengths is its executability while still maintaining abstract specification capabilities. Thus, an alternative to the first option is to include routine implementations which can be invoked as messages are sent. Thus, contracts are again encoded as agents, but the metaclass *ROUTINE* also includes a new feature called *implementation*. This feature is invoked during the MVCC. MVCC in this case resolves to *generating code* from the metalevel encoding of the BON class diagram, generating an implementation of a dynamic diagram that instantiates objects and calls routines and running the dynamic diagram against the class diagram.

We should also refer to related work on the BON Design Tool (BDT) [Taligheni and Ostroff 2003] which provides simulation facilities for BON contracts. This

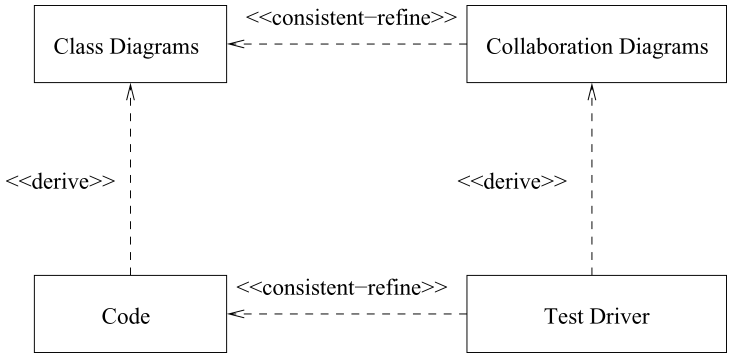


Fig. 16. Refinement structure for contract-consistency scheme.

technology integrated with the Eiffel metamodel would be an alternative to the Eiffel encoding of contracts previously discussed. BDT relies on fully automated theorem proving behind the scenes in order to carry out simulation. Efforts are continuing on integrating the metamodel in Eiffel with BDT.

The last approach is the most compatible with Eiffel's best practices and is the one that we describe in more detail now.

4.2.1 *A Transformational Approach to Multiview Consistency Checking.* The basic approach that we are advocating is to generate *unit tests* from dynamic diagrams based on the metamodel encoding discussed earlier. Executable Eiffel code can be generated from the metamodel encoding of class diagrams as well. The unit tests can then be executed against the generated code to check for consistency. If the test drivers run successfully, we infer that the code and the unit tests, and hence the dynamic diagrams and class diagrams, are consistent.[8]

We remind the reader that this approach is only used for checking constraint (4), that is, contractual consistency. The remaining multiview constraints can be handled directly by conformance checking and do not require any transformation technology to be applied.

This is an indirect consistency check: effectively, the unit tests are viewed as a refinement of dynamic diagrams and the executable code as a refinement of class diagrams. This is depicted as in Figure 16. The stereotype <<**derive**>> on the dependencies indicates that the source of the dependency can be automatically derived (and is therefore automatically consistent) from the target. The stereotype <<**consistent-refine**>> indicates that a consistency-checking process must take place. Note that the dependency between the executable work products (the source code and the unit test) is a refinement of dependency between the models; the implementation of the consistency checking algorithm must guarantee this. The dependency between class diagrams and code is standard code generation; the dependency between unit tests and dynamic diagrams is to be presented shortly.

Generating Eiffel code from the metalevel encoding of class diagrams is straightforward. Generating unit tests from dynamic diagrams is slightly more

[8]Assuming the correctness of the code and test generation algorithms.

complex. The simplifying assumption that we made earlier in article, that we handle only direct associations and inheritance relationships, will allow a generic and reasonably straightforward generation algorithm. Gao [2004] considers the general problem.

Given that a dynamic diagram typically refines a scenario of use in an object-oriented model, it is useful to be able to specify conditions that should be true when a scenario ends. We thus add a *final state* to the metamodel: when the sequence of message calls in the dynamic diagram ends, the final state should be reached. We already include an initial state for a dynamic diagram (see the PVS version). We currently require that the initial and final state specifications be in the Eiffel assertion language, and thus that they are machine checkable and executable. They will therefore be expressed as **check** statements (similar to C's `assert`) in Eiffel. Given the expressive power of Eiffel, this is a not unreasonable restriction.

We have considered two (generally equivalent) approaches to transforming dynamic diagrams into unit tests. We describe one in detail, and briefly outline the second. The first approach uses a syntax-directed algorithm to generate a single Eiffel class, *CONSISTENCY_TEST*, which implements a unit test. This class is inherited from the Eiffel unit-testing framework, ETest. The class possesses a **creation** routine, *make*, that is executed when *CONSISTENCY_TEST* is instantiated. The creation routine executes a sequence of feature calls generated according to the sequence of messages appearing in the class diagram. If guards appear on messages, the feature calls will be prefixed with suitable **if-then-else** structures, or **loop-end** structures, in the case where an iterative multiplicity constraint is provided.

Two challenges arise with the refinement process of dynamic diagrams into an Eiffel unit test.

(1) *New Messages*. These indicate the creation of a new object of the type of the recipient of the message. Each object in the dynamic diagram is mapped into an entity in Eiffel. However Eiffel classes may have many constructors/creation routines, and unlike languages such as C++ and Java, constructors can have any name. Even when dealing with a language like C++ and Java, which force constructors to have the same name as the class, will require dealing with a choice of multiple constructors in order to distinguish argument types. Thus, user assistance will be necessary in general to select the appropriate constructor to execute as a result of a new message. In the test-driver generation algorithm presented shortly, user assistance is obtained through the use of select features, for example, `select_create_feature`. This user assistance simply takes the form of selecting a feature from a specified, automatically generated list.

(2) *Underspecified Messages*. A message in the dynamic diagram may be annotated with the name of the feature that should be called in response. In this case, the feature call is added directly to the test driver. In general though, a message can be underspecified: names or types of arguments may not be provided, or messages may be overloaded, or the specific target of the message may not be precisely constrained. The last case arises when

```
class GENERATOR feature
  generate_test_driver(c: COLLABORATION_DIAGRAM) is
  local
     i: INTEGER
     m: MESSAGE
     f: E_FEATURE
  do
     generate_driver_header;
     generate_declarations(c.objects);
     generate_check_statement(c.initial);
     from i := 1
     until i>c.messages.length
     loop
       m := messages.item(i);
       if m.has_guard then
         generate_ifthen(m.guard)
       end
       if m.has_multiplicity then
         generate_loop(m.multiplicity)
       end
       if m.feature.is_create then
         -- return specific creation routine
         f := select_create_feature(m.target);
       else
         -- return specific feature with arguments
         f := select_feature(m.target);
       end
       generate_feature_call(m.target,f);
       generate_close_branches;
       i := i+1;
     end
     generate_check_statement(s.final);
     generate_driver_footer;
   end
   ...
end -- GENERATOR
```

Fig. 17. Algorithm for generating unit tests from the dynamic diagram.

a multiobject appears in the collaboration diagram. In all of these cases, it is best to ask for assistance from the user to indicate which feature to call in response to a message or to specify the name of an object that should be the recipient of the message. This is the purpose of `select_feature` in Figure 17 which provides the user with a choice of features from which one must be selected.

It is important to point out that the generated unit test is not sufficient for full test coverage of the system—it is to be used for checking multiview consistency. For test coverage, the dynamic diagram would have to be refined to include exceptional conditions. Providing that modelers are able to extend their collaboration diagrams to include exceptional cases, the approach can be

used to generate tests for functional and integration testing as well. However, we suggest that in practice this approach is not sensible since the collaboration diagrams will quickly become large, cumbersome, and difficult to understand. A pragmatic approach might be to extend the generated unit tests to carry out more detailed functional and integration testing, that is, to modify the unit tests directly.

We can now use the metamodel encoding of BON class and dynamic diagrams to express the unit-test generation algorithm. This appears as part of the definition of class *GENERATOR*.

The algorithm works informally as follows. First, declaration header information is generated for the unit test (i.e., name of test class and standard Eiffel syntax), then declarations for objects in the dynamic diagram. A **check** statement (i.e., an `assert`) is generated to test the validity of the initial state of the system. Then, the messages in the diagram are looped over. Each message is tested to see if it has a guard or multiplicity constraints, and suitable Eiffel **if-then-else** statements or **loop** statements are generated. Then a feature call is generated; this may require consulting the user to select either a default **create** statement or one of several possible **create** routines. Finally, all branches in the generated code (i.e., loops or selections) are closed, and a final check is generated on the final state of the system. The algorithm is interactive in the general case, but if users have specified features for each message (i.e., they have refined away nondeterminism) then the unit test generation is automatic. Any interactions would be limited to selecting a method from a drop-down list of options.

Running the unit test, and hence carrying out the MVCC, is a simple matter of compiling and executing the generated code, including the unit test, using the unit test as the root class [Meyer 1997] of the system. The root class contains a method from which execution must start. Examples of MVCC using this approach can be found in Paige et al. [2003a] and Gao [2004].

An alternative approach to transformation is demonstrated in Paige et al. [2005] where the Atlas Transformation Language (ATL) [Bezivin et al. 2003] is applied. This approach allows transformations, defined in terms of a set of rules, to be defined with explicit reference to a pair of metamodels. As a result, the rules are concise and abstract. It is easier to maintain and modify the ATL version of the transformation than the Eiffel algorithm in Figure 17, and thus future extensions of the approach will likely be based on the ATL transformation.

4.2.2 *Example: Using Eiffel for Multiview Consistency Checking.* We now demonstrate how to use Eiffel for MVCC with an example, which shows how to check constraints (1)–(3), that is, all but contract consistency. Using Eiffel for MVCC is very similar to how we use the language for model-conformance checking. This is not surprising since the multiview consistency rules are encoded at the metalevel. A short example illustrates the process.

Consider once again the diagrams shown earlier in Figure 1. These diagrams present a model of part of a simple maze game with the class diagram presenting a structural view, and the dynamic diagram presenting a behavioral view. These

```
views_consistent: BOOLEAN is
local
  maze_game, room, player, set_room, set_player, user: E_CLASS;
  m1, m2, m3, m4: MESSAGE; ....; m: MODEL;
do
  -- Initialise class diagram: classes, attributes, routines.
  create maze_game.make("MAZE_GAME"); create room.make("ROOM");
  create player.make("PLAYER"); create set_room.make("SET[ROOM]");
  ...
  -- Initialise dynamic diagram: objects and messages.
  create mg.make("mg",maze_game); create p.make("p", player);
  ...
  -- Create the model
  create m.make; m.add_class(maze_game); m.add_class(player); m.add_class(room);
  m.add_message(m1); m.add_message(m2); m.add_message(m3); m.add_message(m4);
  m.prepare; Result:=true;
end
```

Fig. 18. Unit test for multiview consistency.

Table III. Comparison of View Consistency Approaches (• = least, ••• = most)

Quality Factor	MVCC approach	
	PVS	Eiffel
Automation	•	•••
Completeness	•••	••

views are inconsistent according to the metamodel of BON. We test this by writing the unit test that is excerpted in Figure 18. This unit test encodes both the class diagram and the dynamic diagram as Eiffel reference structures. The test is then executed in order to check the well-formedness rules.

The routine `views_consistent` is a boolean-valued function that belongs to the same class as the previous example; thus, the results of running this unit test will appear in the HTML tables that are generated by ETest. As it turns out, the unit test fails, because it atempts to invoke a routine, *in_end_room* that does not exist in class *MAZE_GAME*. This is easy to repair. After repairing this, we can experiment and introduce an additional inconsistency by modifying the export policy of routine *is_end_room*, that is, by exporting this routine only to class *ROOM*. In this case, the unit test fails again since the constraint *message-feature* will not hold.

The dynamic diagram in Figure 1(b) assumes that routines have been associated with messages. In general, dynamic diagrams in BON can be used informally, for example, to refine use case scenarios, and such an association cannot be assumed.

4.3 Comparison

Table III summarises the view consistency approaches in terms of the quality factors discussed earlier.

Once again, the comparisons are subjective and are based on experience and careful analysis of the approaches. As we would perhaps expect, the PVS approach offers the most complete solution to multiview consistency checking. Consistency constraints can be specified in PVS as axioms (or even conjectures), and the theorem prover can be used to verify that a model obeys the axioms. However, the PVS approach is lacking in terms of automation, invariably user intervention is required to help discharge a proof (particularly for choosing instantiations of quantified variables). The Eiffel approach to view consistency checking is fully automatic but incomplete—some consistency constraints, particularly those BON contracts that cannot be translated to Eiffel—cannot be checked using the Eiffel approach in the same way as the PVS approach.

The comparison in terms of completeness is a representation of the current state-of-play. We expect to make progress on moving the completeness of the Eiffel approach closer to PVS by building the Eiffel metamodel as a .NET application and using its reflection capabilities to extract predicates which can then be passed along to a theorem prover for external verification.

5. SUMMARY AND CONCLUSIONS

We have presented and contrasted two different approaches to metamodeling the BON language. Comparisons of the approaches were based on a number of identified quality factors that are of interest to metamodelers and users of metamodels. The metamodel specifications were thereafter used in presenting approaches to multiview consistency checking, and these were in turn contrasted in terms of their completeness and level of automation.

An obvious conclusion of this work is that no one approach to metamodeling, and hence multiview consistency checking, is sufficient: a trade-off between levels of automation and completeness will have to be made. As well, issues of understandability, usability, and scalability must also be taken into account. A further concern is maintainability. Given the substantial efforts put into revising UML over the past few years, with its corresponding changes in the underlying metamodel and supporting tools, it is highly desirable for a metamodel to be maintainable, amenable to change. To this end, it will be useful if metamodeling can be done in an agile way, accepting that change to language specifications is inherent in the process and that the supporting tools that we use to write and verify and validate metamodels support agile development as well. We have explored these issues further in Paige et al. [2004] where we applied an agile process for building a small metamodel.

Another observation from this work is that, for metamodeling, we should not always prefer a more expressive metamodeling language to a less expressive one. PVS is more expressive than both BON and Eiffel and can capture all well-formedness rules; however, what it offers in completeness, it loses in terms of automation and ease-of-use. Eiffel, as a metamodeling language, is incomplete and yet it can be used to capture and check almost all well-formedness rules automatically. Moreover (as we discuss in the next paragraph), we have preliminary evidence to suggest that the Eiffel approach is more scalable. Whether we

should prefer a more expressive language to a less expressive one will depend on how we want to use the metamodel in other tasks.

Our aim in this article was to provide guidelines and recommendations to metamodelers and language designers in terms of the factors that they should consider when constructing metamodels and view consistency-checking facilities. It would be useful to broaden the comparison to include additional quality factors and a more quantitative set of comparisons. For example, some measure of maintainability or extensibility in terms of change metrics would be useful to include as would measures of the size of models and metamodels that are feasible to check using the PVS and Eiffel approaches. While we do not yet have conclusive data about scalability of either approach, our initial indications are that the Eiffel approach scales easily and maintains the ability to automatically check conformance and multiview consistency (up to the limitations noted earlier). In part, this is due to the object-oriented characteristics of Eiffel, but is also because of its executability. We plan to carry out further case studies with PVS, particularly to produce proof strategies, to assess the scalability of the theorem-proving approach.

Our future work is taking an agile approach to extending the Eiffel specification of the metamodel to more detailed consistency checking; in this sense, we are building up a relatively simple method of consistency checking to more complex tasks. There are two directions to this research: supporting contracts in more detail (as discussed in Section 4), and supporting additional views. We plan to first add a statechart view to the metamodel, basing this work on the new Event library available with Eiffel. In this manner, an event-driven model of concurrency and distribution will underpin the multiview metamodel.

REFERENCES

AHRENDT, W., BAAR, T., BECKERT, B., BUBEL, R., GIESE, M., HAHNLE, R., MENZEL, W., MOSTOWSKI, W., ROTH, A., SCHLAGER, S., AND SCHMITT, P. 2005. The KeY tool. *J. Softw. Syst. Model. 4*, 1.

AKEHURST, D., PATRASCOUI, O., AND SMITH, R. 2004. The Kent modelling framework user guide. http://www.cs.kent.ac.uk/projects/kmf.

AMALIO, N., STEPNEY, S., AND POLACK, F. 2004. Modular UML semantics: Interpretations in Z based on templates and generics. In *Formal Aspects of Component Software, International Workshop (FACS'03)*. UNU/IIST.

BEZIVIN, J., DUPE, G., JOUAULT, F., PITETTE, J., AND ROUGUI, J. 2003. First experiments with the ATL model transformation language. In *Workshop on Generative Techniques in the Context of MDA*. http://www.softmetaware.com/oopsla2003/mda-workshop.html.

BHADURI, P. AND VENKATESH, R. 2002. Formal consistency of models in multi-view modeling. In *Workshop on Consistency Problems in UML-Based Software Development.*

BIDOIT, M. AND MOSSES, P. D. 2004. CASL user manual. Lecture Notes in Computer Science, vol. 2900 (IFIP Series). Springer-Verlag.

BUDINSKY, F., STEINBERG, D., MERKS, E., ELLERSICK, R., AND GROSE, T. 2003. *The Eclipse Modelling Framework*. Addison-Wesley.

CHECHIK, M., DEVEREAUX, B., EASTERBROOK, S., AND GARFINKEL, A. 2003. Multi-valued symbolic model checking. *ACM Trans. Softw. Engin. Method. 12*, 4.

CHIOREAN, D. 2005. OCLE 2.0 User Manual. http://lci.cs.ubbcluj.ro/ocle/.

CLARK, T., EVANS, A., AND KENT, S. 2001a. The metamodelling language calculus: Foundation semantics for UML. In *Proceedings of the Fundamental Aspects of Software Engineering*. Lecture Notes in Computer Science, vol. 2029, Springer-Verlag.

CLARK, T., EVANS, A., AND KENT, S. 2001b. MMT programmer's guide. www.dcs.kcl.ac.uk/staff/tony/docs/ProgrammersGuideToMMT.pdf.

D'SOUZA, D. AND WILLS, A. 1998. *Objects, Components and Frameworks with UML*. Addison-Wesley.

EVANS, A., MASKERI, G., MOORE, A., SAMMUT, P., AND WILLANS, J. 2005. A unified superstructure for UML. *J. Object Techn. 4*, 1.

EVANS, A., SAMMUT, P., AND WILLANS, J. 2003. Proceedings of the metamodelling for MDA workshop. Tech. rep., University of York.

FINKELSTEIN, A., GABBAY, D., HUNTER, A., KRAMER, J., AND NUSEIBEH, B. 1994. Inconsistency handling in multi-perspective specification. *IEEE Trans. Softw. Engin. 20*, 8.

FONDEMENT, F. AND BAAR, T. 2005. Making metamodels aware of concrete syntax. In *European Conference on MDA (ECMDA'05)*. Lecture Notes in Computer Science, vol. 3748, Springer-Verlag.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns*. Addison-Wesley.

GAO, Y. July 2004. Multi-view consistency checking of BON software description diagrams. MSC thesis, York University, Canada.

GRYCE, C., FINKELSTEIN, A., AND NENTWICH, C. 2002. Xlinkit: Lightweight consistency checking for the UML. In *Workshop on Consistency Problems in UML-based Software Development*.

HUSSMAN, H., DEMUTH, B., AND FINGER, F. 2000. Modular architecture for a toolset supporting OCL. In *Proceedings of UML 2000*. Lecture Notes in Computer Science, vol. 1939, Springer-Verlag.

HUZAR, Z., KUZNIARZ, L., REGGIO, G., AND SOURROUILLE, J. 2002. *Workshop on Consistency Problems in UML-Based Software Development*. http://www.ipd.bth.se/UML2002.

HUZAR, Z., KUZNIARZ, L., REGGIO, G., AND SOURROUILLE, J. 2003. *Workshop on Consistency Problems in UML-Based Software Development*. http://www.ipd.bth.se/consistencyUML/UML2003Workshop.asp.

HUZAR, Z., KUZNIARZ, L., REGGIO, G., AND SOURROUILLE, J. 2004. *Workshop on Consistency Problems in UML-Based Software Development*. http://uml04.ci.pwr.wroc.pl/.

IEEE. 2000. IEEE Std. 1471-2000 Recommended Practice for Architectural Description of Software Intensive Systems. standards.ieee.org.

KIM, S. AND CARRINGTON, D. 2004. Using integrated metamodeling to define OO design patterns with Object-Z and UML. In *Proceedings of the Asia-Pacific Software Engineering Conference*. IEEE, 257–264.

KRISHNAN, P. 2000. Consistency Checks for UML. In *Proceedings of the 7th Asia-Pacific Software Engineering Conference*. IEEE, 162–169.

KUZNIARZ, L., REGGIO, G., AND SOURROUILLE, J. 2005. *Workshop on Consistency in Model-Driven Engineering*. http://www.ipd.bth.se/consistencyUML/CoMoDE

LEAVENS, G., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D., MUELLER, P., AND KINIRY, J. 2005. JML Reference Manual. http://www.cs.iastate.edu/ leavens/JML/jmlrefman/.

MARCANO, R. AND LEVY, N. 2002. Using B formal specification for analysis and verification of UML/OCL models. In *Workshop on Consistency Problems in UML-Based Software Development*.

MELLOR, S. AND BALCER, M. 2002. *Executable UML*. Addison-Wesley.

MEYER, B. 1992. *Eiffel—The Language* 2nd ed. Prentice Hall.

MEYER, B. 1997. *Object Oriented Software Construction* 2nd ed. Prentice Hall.

MEYER, B. 2003. Towards practical proofs of class correctness. In *Proceedings of the 3rd International Conference of B and Z Users (ZB'03)*. Lecture Notes in Computer Science, vol. 2651, Springer-Verlag.

MICROSOFT. 2005. Microsoft Visio Web resource. http://office.microsoft.com/en-gb/FX010857981033.aspx.

MICROSOFT. 2006. Spec# programming system. http://research.microsoft.com/specsharp/.

MODELWARE. 2005. D1.5: Model composition definition—consistency rules. http://www.modelware-ist.org.

OBJECT MANAGEMENT GROUP. 2003a. MDA guide version 1.0.1.

OBJECT MANAGEMENT GROUP. 2003b. UML Standard Guide 1.5.

OBJECT MANAGEMENT GROUP. 2004a. MOF Meta-Object Facility Specification 1.4.

OBJECT MANAGEMENT GROUP. July 2004b. UML 2.0 working documents. www.omg.org.

OWRE, S., SHANKAR, N., RUSHBY, J., AND STRINGER-CALVERT, D. 1999. PVS Language Reference. pvs.csl.sri.com.

PAIGE, R., BROOKE, P., AND OSTROFF, J. 2004. Specification-driven development of an executable metamodel. In *Proceedings of Workshop in Software Model Engineering*. http://www.metamodel.com/wisme-2004/present/6.pdf.

PAIGE, R., BROOKE, P., AND OSTROFF, J. 2005. Lightweight metamodeling, conformance, and view consistency checking. http://www.cs.york.ac.uk/~paige/trlwmm.pdf.

PAIGE, R., KOLOVOS, D., AND POLACK, F. 2005. Refinement via Consistency Checking in MDA. In *Proceeding of Refinenet Workshop, ENTCS*.

PAIGE, R. AND OSTROFF, J. 1999a. A Comparison of BON and UML. In *Proceedings of Unified Modeling Languages (UML'99)*. Lecture Notes in Computer Science, vol. 1723, Springer-Verlag.

PAIGE, R. AND OSTROFF, J. 1999b. Developing BON as an industrial-strength formal method. In *Proceedings of World Congress on Formal Methods*. Lecture Notes in Computer Science, vol. 1708, Springer-Verlag.

PAIGE, R. AND OSTROFF, J. 2001. Metamodelling and conformance checking with PVS. In *Proceedings of Fundamental Aspects of Software Engineering*. Lecture Notes in Computer Science, vol. 2029, Springer-Verlag.

PAIGE, R. AND OSTROFF, J. 2004. ERC: An object-oriented refinement calculus for Eiffel. *Formal Aspects Comput. 16*, 1.

PAIGE, R. AND OSTROFF, J. 2000. Precise and formal metamodeling with BON and PVS. Tech. rep. 2000-03, York University.

PAIGE, R., OSTROFF, J., AND BROOKE, P. 2002. Checking the consistency of class and collaboration diagrams using PVS. In *Proceedings of Rigorous Object-Oriented Methods 4 (ROOM4)*. British Computer Society.

PAIGE, R., OSTROFF, J., AND BROOKE, P. 2003a. A test-based and agile approach to checking the consistency of class and collaboration diagrams. In *Proceedings of UK Software Testing Research Workshop*.

PAIGE, R., OSTROFF, J., AND BROOKE, P. 2003b. Theorem proving support for view consistency checking. *L'Objet 9*, 4.

RICHTERS, M. AND GOGOLLA, M. 2000. Validating UML models and OCL constraints. In *Proceedings of Unified Modeling Languages (UML'00)*. Lecture Notes in Computer Science, vol. 1939, Springer-Verlag.

SOCIETY OF AUTOMOTIVE ENGINEERS. 2005. Architectural Analysis and Design Language (AADL) Standard. http://www.aadl.info.

SOURROUILLE, J. AND CAPLAT, G. 2002. A pragmatic view about consistency checking of UML models. In *Workshop on Consistency Problems in UML-Based Software Development*.

SPENCER, G. 2005. OCL to Eiffel. MSC Thesis. http://www.cs.york.ac.uk/library/.

TALIGHENI, A. 2004. Contractual consistency between BON static and dynamic diagrams. MSC Thesis, York University, Canada.

TALIGHENI, A. AND OSTROFF, J. 2003. The BON Development Tool. In *Proceedings of Eclipse Technology Exchange*.

WALDEN, K. AND NERSON, J.-M. 1995. *Seamless Object Oriented Software Architecture*. Prentice Hall.

XACTIUM. 2006. XMF-Mosaic User Guide (prerelease version 0.1.) www.xactium.com.

Received July 2005; revised May 2006; accepted December 2006