Metamodelling and Conformance Checking with PVS

Richard F. Paige and Jonathan S. Ostroff

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

Abstract. A metamodel expresses the syntactic well-formedness constraints that all models written using the notation of a modelling language must obey. We formally capture the metamodel for an industrial-strength object-oriented modelling language, BON, using the PVS specification language. We discuss how the PVS system helped in debugging the metamodel, and show how to use the PVS theorem prover for conformance checking of models against the metamodel. We consider some of the benefits of using PVS's specification language, and discuss some lessons learned about formally specifying metamodels.

1 Introduction

Modelling languages such as UML [2], BON [7], and others have been used to capture requirements, describe designs, and to document software systems. Such languages are supported by tools, which aid in the construction of models, the generation of code from models, and the reverse engineering of models from code.

A modelling language consists of two parts: a *notation*, used to write models; and a *metamodel*, which expresses the syntactic well-formedness constraints that all valid models written using the notation must obey [2]. Metamodels serve several purposes that may be of interest to different modelling language users.

- **Modellers:** metamodels should be easy to understand by modellers. Thus, metamodels should be expressed so that their fundamental details can be easily explained to modellers, without requiring them to understand much formalism.
- **Tool Builders:** metamodels provide specifications for tool builders who are constructing applications to support the modelling language. Thus, metamodels should be precise and not open to misinterpretation.
- **Modelling Language Designers:** language designers have the responsibility to ensure that metamodels are consistent. Thus, metamodels should be expressed in a formalism so that automated reasoning about it can be carried out.

Different modelling language users have different goals, and therefore a metamodel must possess a collection of different characteristics. Metamodels must be understandable, to assist in the use of the language and its supporting tools. They should be unambiguous and not open to misinterpretation. A metamodel should be expressed in a form amenable to tool-based analysis, e.g., for consistency checking. And, to best deal with complexity and issues of scale, a metamodel should be well-structured.

In this paper, we present a formal specification of the metamodel of the BON object-oriented (OO) modelling language [7], written using the PVS specification language [3].

The PVS language has been designed for automated analysis, using the PVS system, which provides a theorem prover and typechecker. We construct the formal specification in two steps. We first specify the BON metamodel informally, using BON itself. In this manner, we use BON's structuring facilities to help understand the key abstractions used in the metamodel and their constraints, before writing a formal specification. From the BON version of the metamodel, we then construct a set of PVS theories that capture the metamodel. The PVS theories can then be used, in conjunction with the PVS system, to analyze and answer questions about the metamodel.

The specific contributions of this paper are as follows.

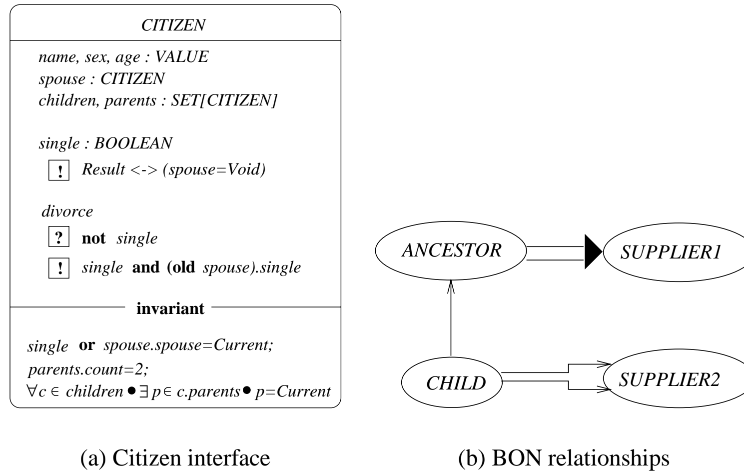
1. A formal specification of the syntactic well-formedness constraints for BON in the PVS language. To our knowledge, this is the first formal specification of the full metamodel of an OO modelling language in a form that is also amenable to automated analysis.
2. A description of how the PVS language can be used for metamodelling, and how the PVS system can be used to help debug and verify the metamodel.
3. Examples of how the PVS system can be used to reason about the metamodel. As a specific example, we show how to prove that BON models satisfy the metamodel.

2 An Overview of BON

BON is an OO method possessing a recommended process as well as a graphical language for specifying object-oriented systems. The fundamental construct in the BON language is the class. A class has a name, an optional class invariant, and zero or more features. A feature may be a *query* – which returns a value and does not change the system state – or a *command*, which does change system state but returns nothing. Fig. 1(a) contains an example of a BON model for the interface of a class *CITIZEN*. Features are in the middle section of the diagram (there may be an arbitrary number of feature sections, each annotated with names of classes that may access the features). Features may optionally have contracts, written in the BON assertion language, in a pre- and postcondition form. In postconditions, the keyword **old** can be used to refer to the value of an expression when the feature was called. Similarly, the implicitly declared variable *Result* can be used to constrain the value returned by a query. An optional class invariant is at the bottom of the diagram. The class invariant is an assertion that must be *true* whenever an instance of the class is used by another object. In the invariant, *Current* refers to the current object.

Classes and features can be tagged with a limited set of stereotypes. The root class contains a feature from which computation begins. A class name with a * next to it indicates the class is deferred, i.e., some features are unimplemented and thus the class cannot be instantiated; a + next to a class name indicates the class can be instantiated. A ++ next to a feature name indicates the feature is redefined, and its behaviour is changed from behaviour inherited from a parent.

In Fig. 1(a), class *CITIZEN* has seven queries and one command. For example, *single* is a *BOOLEAN* query while *divorce* is a parameterless command that changes the state of an object. $SET[G]$ is a generic predefined class with generic parameter G. $SET[CITIZEN]$ thus denotes a set of objects each of type *CITIZEN*.



(a) Citizen interface (b) BON relationships

Fig.1. BON syntax for interfaces and relationships

BON models usually consist of classes organized in *clusters* (drawn as dashed rounded rectangles – see Section 3), 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 Fig. 1(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.
- **Client-supplier:** there are two client-supplier relationships, association and aggregation. Both relationships are directed from a *client* class to a *supplier* class. 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 *SUPPLIER*2 in Fig. 1(b), whereas an association is drawn from *ANCESTOR* to class *SUPPLIER*1.

3 BON Specification of the Metamodel

We now present an informal specification of the BON metamodel, written in BON itself. This description is aimed at promoting an understanding of the fundamental abstractions and relationships that BON models use. We use BON to informally capture the metamodel, and the BON description will be used to help produce a formal specification of the metamodel in the PVS language.

Fig. 2 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.

Fig.2. The BON metamodel, abstract architecture

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 Fig. 3, which shows the interface for *MODEL*. We will not provide details of the individual clauses of the class invariant of *MODEL* (these can be found in [5]).

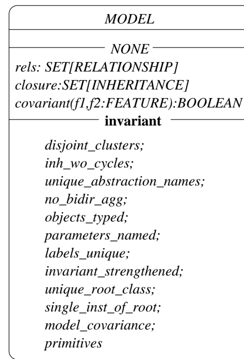


Fig.3. Interface of class *MODEL*

3.1 The relationships cluster

The relationships cluster describes the four basic BON relationships, as well as constraints on their use. The details of the cluster are shown in Fig. 4.

There are several important things to observe about Fig. 4.

- *Type redefinition:* BON allows types of features to be covariantly redefined [1] when they are inherited: a type in the signature of a feature can be replaced with a subtype. In class *RELATIONSHIP*, the attributes *source* and *target* are given types *ABSTRACTION*. In *INHERITANCE* and *CLIENT_SUPPLIER*, the types are redefined to *STATIC_ABSTRACTION*.
- *Aggregation:* an aggregation relationship cannot target its own source; this is precisely captured by the invariant on *AGGREGATION*. Associations, however, may target their source because they depict reference relationships.
- *Inheritance:* a class cannot inherit from itself. This is captured by the invariant of class *INHERITANCE*.

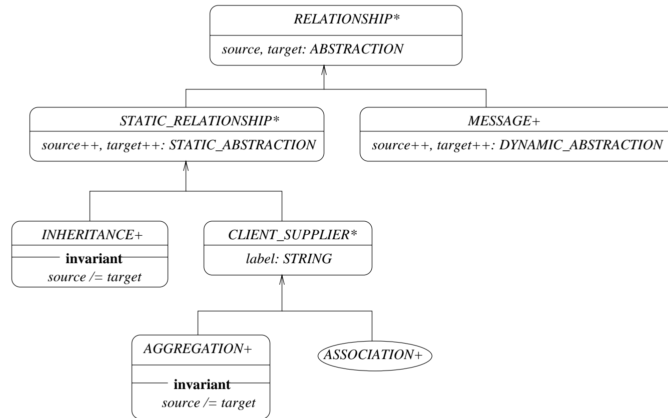


Fig.4. The relationships cluster

3.2 The abstractions cluster

The abstractions cluster describes the abstractions that may appear in a BON model. It is depicted in Fig. 5. Details of the invariant clauses of individual classes may be found in [5]; PVS specifications of several clauses will be presented in Section 4.

The features cluster of Fig. 5 will be described in Section 3.3. It contains the constraints relevant to features of classes. In particular, this cluster introduces the class *FEATURE* to represent the abstract notion of a feature belonging to a class.

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, i.e., static clusters contain only static abstractions.

3.3 The features cluster

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; the cluster that metamodels assertions can be found in [5]. Query calls may appear in assertions; the set of query calls that appear in an assertion must be modelled 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*, 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 parameterless queries that the feature may modify. Fig. 6 depicts the cluster.

4 PVS Specification of the Metamodel

In this section, we present a formal specification of the BON metamodel in the PVS specification language. We attempt to parallel the structure of the BON metamodel

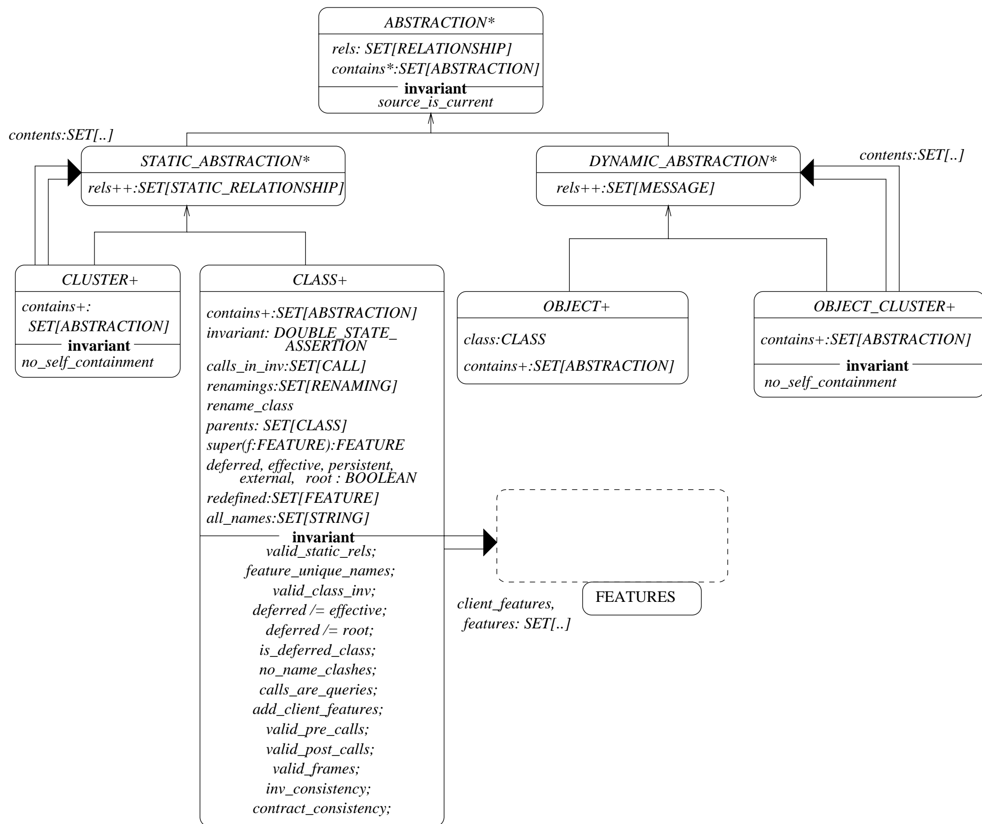


Fig.5. The abstractions cluster

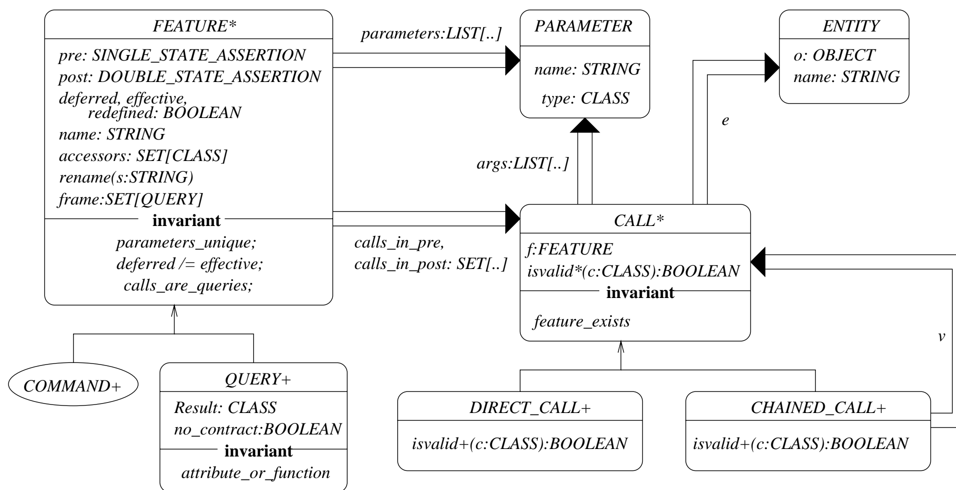


Fig.6. BON metamodel, features cluster

in the PVS language. We present the PVS version of the metamodel selectively, and attempt to give the flavour of using PVS for this purpose (the full metamodel can be found in [5]).

4.1 Theory of abstractions

A BON model can contain a number of abstractions, specifically classes, clusters, objects, and clusters of objects. To express this in PVS, we introduce a new non-empty type and a number of subtypes, effectively mimicking the inheritance hierarchy presented in Fig. 5. This information is declared in the PVS theory `abs_names.pvs`.

```
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 modelling 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).

```
abstractions: THEORY
  IMPORTING abs_names, rel_names

  % Features are queries or commands.
  FEATURE: TYPE+
  QUERY, COMMAND: TYPE+ FROM FEATURE
  ENTITY: TYPE+
```

We can now model feature calls (which may appear in an assertion associated with a feature). Parameters and arguments are modelled as functions from features to finite sequences of records. Calls are either direct (of the form $o.f$) or chained (of the form $o.p.q$).

```
  PARAM: TYPE = [# name:string, param_type: CLASS #]
  IMPORTING sequences[PARAM]
  parameters: [ FEATURE -> finite_sequence[PARAM] ]

  % Direct and chained calls.
  CALL: TYPE+
  DIRECT_CALL, CHAINED_CALL: TYPE+ FROM CALL
```

Primitive BON classes, e.g., *INTEGER*, are modelled as PVS constants: objects of `CLASS` type. We also define conversions so that the type checker can automatically convert BON primitives into PVS types.

```
bool_object, int_object, string_object, real_object, any_object: CLASS

% Example conversions that the PVS typechecker can automatically apply.
interp_bool:   [ bool_object -> bool ]
interp_int:    [ int_object -> int ]
CONVERSION interp_bool, interp_int
```

We must now describe constraints on abstractions. In the BON version of the meta-model, these took the form of features and class invariants. In PVS, the well-formedness constraints will appear as functions, predicate subtypes, and axioms. We start by defining a number of functions that will later be used to constrain the model.

```
% The class that an object belongs to, and the features of a class.
object_class: [ OBJECT -> CLASS ]
class_features: [ CLASS -> set[FEATURE] ]

% The contents of a cluster. Note that clusters may be nested.
cluster_contents: [ CLUSTER -> set[STATIC_ABS] ]
```

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.

```
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 need to be able to capture the concept of a legal set of calls. Consider an assertion in BON, e.g., a precondition. Such an assertion may call a query if the class owning the query has given permission to do so. To accomplish this, we introduce functions that give us all the calls associated with a precondition, postcondition, and invariant.

```
calls_in_pre, calls_in_post: [ FEATURE -> set[CALL] ]
calls_in_inv: [ CLASS -> set[CALL] ]
```

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))
```

A further axiom ensures that clusters do not contain themselves.

```
no_self_containment_cl: AXIOM
  (FORALL (cl:CLUSTER): not member(cl,cluster_contents(cl)))
```

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.

```
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)))))
```

Classes may possess stereotypes, e.g., they may be deferred or effective. Here is an example, showing that a class cannot be both deferred and effective.

```
deferred_effective_ax: AXIOM
  (FORALL (c:CLASS): (NOT (deferred_class(c) IFF effective_class(c))))
```

4.2 Theory of relationships

The theory of relationships defines the three basic relationships and the well-formedness constraints that exist in BON. To express the relationships in PVS, we introduce a new non-empty type and a number of subtypes. As with abstractions, we mimic the inheritance hierarchy that was presented in Fig. 4.

```
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 modelled 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. We give one example of relationship constraints: that inheritance relationships cannot be self-targetted.

```
% 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. Thus, further relationship constraints will be added in the next section, where we describe constraints on models themselves.

4.3 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` (described below) mimics the structure of the BON model in Fig. 2: a model consists of a set of abstractions.

```
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 first constraint we write ensures that inheritance hierarchies do not have cycles. 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.

```
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)))
```

Two further functions will be used in ensuring syntactic covariant redefinition of features. In BON, if a feature's signature is redefined when it is inherited, it can be changed to a subtype.

```
% is_subtype is true iff the second arg. is a descendent of the first
is_subtype: [ CLASS,CLASS -> bool ]

% The function covariant takes two features and results in true
% iff the second feature covariantly redefines the first.
covariant: [ FEATURE,FEATURE -> bool ]

covariant_ax1: AXIOM
  (FORALL (que1,que2:QUERY): covariant(que1,que2) IFF
    length(parameters(que1))=length(parameters(que2)) AND
    (FORALL (i:{j:nat|j<length(parameters(que1))}):
      is_subtype(param_type(parameters(que1)(i)),
                 param_type(parameters(que2)(i))) AND
      is_subtype(query_result(que1),query_result(que2)))
```

The primary purpose of introducing the `covariant` function is to ensure that redefined features obey the syntactic aspects of the covariant rule. The syntactic aspects of covariance are captured in the metamodel via the axiom `model_covariance`, which ensures that all feature redefinitions obey the covariance rule.

```
model_covariance: AXIOM
  (FORALL (c:CLASS): member(c,abst) IMPLIES
    (FORALL (f:FEATURE): member(f,redefined_features(c)) IMPLIES
                         covariant(f,super(c,f)))
```

We write an axiom demonstrating a well-formedness constraint on clusters: all clusters in a model are disjoint or nested.

```
% 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)))
```

A somewhat complicated axiom to formalize is to ensure that labels appearing on a client-supplier relationship do not clash with names appearing in the feature list of the relationship source, nor with any other client-supplier relationship from the same source. This is reasonably straightforward to formalize in the case where the source of the relationship is a class, but it becomes more complex when the source is a cluster. First we present the case where the source is a class.

```
labels_unique_ax1: AXIOM
  (FORALL (cs:C_S): (member(cs,rels) AND CLASS_pred(cs_source(cs))
    IMPLIES NOT member(cs_label(cs),
      { n:string | (EXISTS (f:FEATURE):
        member(f,class_features(cs_source(cs))) IMPLIES
        n=feature_name(f)) })  AND
        NOT (EXISTS (cs2:C_S): (member(cs2,rels) IMPLIES
         cs_source(cs2)=cs_source(cs) AND cs_label(cs)=cs_label(cs2))) ))
```

A second axiom is needed in the case where the source of the client-supplier relationship is a cluster. In this case, we must require that at least one class contained within the cluster does not have the name appearing on the relationship label.

```
labels_unique_ax2: AXIOM
  (FORALL (cs:C_S): (member(cs,rels) AND CLUSTER_pred(cs_source(cs)))
    IMPLIES (EXISTS (c:CLASS): member(c,all_classes(cs_source(cs))
      IMPLIES NOT member(cs_label(cs),all_names(c))))
```

It was only when typechecking the PVS theories that we discovered the need for `labels_unique_ax2`. Our original formulation considered only the case where the source of a client-supplier relationship is a class. The typechecker provided us with an obligation with the assumption `CLASS_pred(cs_source(cs))`, which is not true for all BON models, since client-supplier relationships may be from clusters as well as classes. Thus, PVS provided us with a counterexample to our original assumptions and thereby suggested extra constraints that needed to be formalized.

The complete metamodel typechecks without any user intervention. It can be found in [5].

5 Conformance Checking with the Metamodel

The metamodel presented in the previous section can be used to check that BON models, which are instances of the metamodel, obey the well-formedness constraints. Conformance checking is by proving PVS `CONJECTURE`s using the axioms of the metamodel. We present two examples to demonstrate the general approach. More examples and further discussion can be found in [5]. The two BON models in Fig. 7 will be used to demonstrate the process.

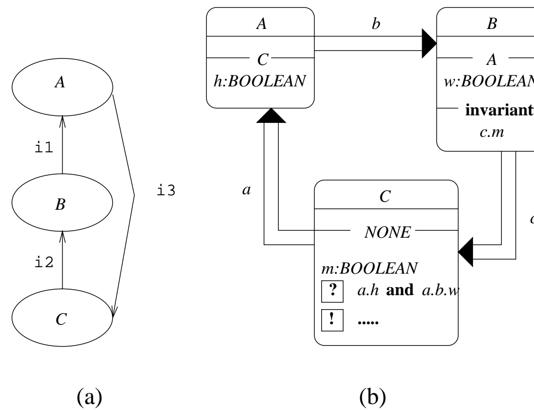


Fig.7. Models for conformance checking.

5.1 Inheritance cycles

We start by showing that a model that possesses cycles in its inheritance graph does not satisfy the metamodel. Consider Fig. 7(a) (labels are for reference only); this model is

not well-formed because of the cycle-introducing inheritance relationship from class *A* to class *C*. If we can describe this model in PVS, then we should be able to conject and prove that it is not well-formed. The conjecture is captured in the following theory.

```
use_metamodel2: THEORY
BEGIN
IMPORTING metamodel
    a,b,c: VAR CLASS
    i1,i2,i3: VAR INH

    no_inh_cycles: CONJECTURE
    (NOT (EXISTS (a,b,c:CLASS): member(a,abst) AND member(b,abst) AND
       member(c,abst) AND a/=b AND b/=c AND c/=a IMPLIES
      (EXISTS (i1,i2,i3:INH): (member(i1,rels) AND member(i2,rels) AND
        member(i3,rels) AND i1/=i2 AND i2/=i3 AND i3/=i1 IMPLIES
        member(i1,static_rels(b) AND member(i2,static_rels(c) AND
        member(i3,static_rels(a)) AND inh_source(i1)=b AND inh_source(i2)=c AND
        inh_source(i3)=a AND inh_target(i1)=a AND inh_target(i2)=b AND
        inh_target(i3)=c))))
END use_metamodel2
```

The conjecture can be explained as follows: there cannot exist a model consisting of the distinct classes *a*, *b*, and *c* with three inheritance relationships *i*1, *i*2, and *i*3 such that *i*1 is directed from *b* to *a*, *i*2 is directed from *c* to *b*, and *i*3 is directed from *a* to *c*. To prove the conjecture with PVS requires use of three axioms, two of which define the inheritance closure of a model, with the third being `inh_wo_cycles`. After instantiating the axioms with the abstractions contained in the model, the conjecture proves automatically using `(grind)`. See [5] for the full proof.

5.2 Obeying export policies of classes

As a second example, we show how to check that a model correctly obeys the export policies of all classes in the model. Consider the BON model in Fig. 7(b). 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 obey the constraints in 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 conjecting that the model in Fig. 7(b) cannot exist. The full conjecture contains a number of terms that are not relevant to the proof (they can be found in [5]) 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 invalid according to the well-formedness constraints of the metamodel.

6 Discussion and Conclusions

By producing a formal metamodel for BON, we have taken a step towards placing the modelling language on a solid mathematical basis. We have captured the well-formedness constraints that all BON models must obey, thus describing core information that is essential for all tool builders and modellers to understand.

We learned several things about PVS and metamodelling in carrying out this exercise. For one, we found the BON version of the metamodel extremely useful in constructing the PVS version. The BON version provided structuring information and indications as to how PVS theories might be related. We also determined several helpful heuristics that can be used, in general, to help model object-oriented concepts in PVS. Class hierarchies can be modelled using PVS types and subtypes. Class features can be described as functions that take a variable as an argument. Commands are PVS functions that take an invoking object as an argument and return a new object. We found the PVS `CONVERSION` facility ideal for transforming BON built-in primitives, e.g., *INTEGER*, into PVS types. Finally, we found that we could model BON's covariant redefinition of feature signatures via PVS's subtyping mechanism. In principle, a formal translation of BON models into PVS, and thereafter a tool, could be developed based on these heuristics.

We found the PVS type checker particularly helpful in debugging the metamodel. Our initial metamodel contained several errors and omissions – e.g., that a client-supplier relationship must always be from a class source, and that we erroneously required that a feature must have one or more parameters – that the checker caught automatically. This was used in updating the metamodel as it was being constructed.

By giving a PVS specification of the metamodel, we have the additional advantage of being able to use the PVS system to analyze the metamodel. The PVS system allowed us to carry out conformance checks of models against the metamodel, as demonstrated

in Section 5. The PVS specification allows us to do more than check models against the metamodel: it allows us to ask *questions* about the metamodel, in particular, about emergent properties of the metamodel. These properties are not explicitly described via the axioms of the metamodel itself; rather, they are logical consequences of the axioms. Thus, the PVS system can be used to help users of the metamodel answer pertinent questions they may have about the metamodel.

Much work remains to be done. We plan to carry out further examples of conformance checking, particularly concentrating on examples that require inductive proofs. We also plan to validate the BON metamodel itself – i.e., prove that the version of the metamodel described in Section 3 is a valid BON model; this will give us greater confidence in the validity of our work. Comparisons of our work with other formal specifications of metamodels will be worthwhile; preliminary efforts on this, for Alloy [6] and UML, can be found in [5]. We also intend to tie this work in with a refinement calculus that we have been creating for BON [4]. In this latter work, we have provided a formal semantics for much of BON in terms of predicates. Thus, we aim to define relationships between the BON metamodel – which captures syntactic constraints – and the formal semantics of abstractions and relationships described elsewhere.

References

1. B. Meyer. *Object-Oriented Software Construction*, Prentice-Hall, 1997.
2. *OMG Unified Modelling Language Specification 1.3*, OMG, June 1999.
3. S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. The PVS Language Reference Version 2.3, SRI International Technical Report, September 1999.
4. R. Paige and J. Ostroff. An Object-Oriented Refinement Calculus. Technical Report CS-1999-07, York University, December 1999.
5. R. Paige and J. Ostroff. Precise and Formal Metamodelling with the Business Object Notation and PVS. Technical Report CS-2000-03, York University, August 2000.
6. M. Vaziri and D. Jackson. Some Shortcomings of OCL, the Object Constraint Language of UML. Technical Report, MIT Laboratory for Computer Science, December 1999.
7. K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Development,* Prentice-Hall, 1995.