Precise and Formal Metamodeling with the Business Object Notation and PVS

Richard F. Paige and Jonathan S. Ostroff
Department of Computer Science
York University
Toronto, Ontario M3J 1P3, CANADA.
{paige,jonathan}@cs.yorku.ca

November 14, 2000

Contents

Abstract

A modeling language consists of both a notation and a *metamodel*, the latter of which captures the syntactic well-formedness constraints that all valid models must obey. We present two versions of a metamodel for an industrial-strength object-oriented modeling language, BON. The first version of the metamodel, written in BON itself, is intended to give a precise and understandable description of the syntactic well-formedness constraints. The second version, written in the PVS specification language, is intended to give a formal description of the constraints, in a form that is amenable to automated checking and analysis. We demonstrate how the PVS version can be used for conformance checking, i.e., showing that BON models satisfy or fail to satisfy the metamodel. A process is defined for using the metamodel in carrying out conformance testing of models. We also contrast the BON metamodel with the metamodel of the Unified Modelling Language (UML), both in terms of the precise metamodel presented in the UML Semantics Reference, and a formalization of the metamodel presented in the Alloy specification language. We discuss lessons learned in constructing metamodels, particularly in terms of using the PVS specification language for metamodeling.

NOTE: the PVS theories discussed in this paper are available from the first author. A web page will be made available at some point that will contain the PVS theories and the most up-to-date version of this document.

1 Introduction

Modeling languages are starting to see increased use in software development. UML [11], BON [14], Z [12], and others have been used to capture requirements, describe designs, and for documentation purposes. Such languages are frequently supported by tools, which abet the expression and construction of models, the forward generation of code, and the reverse engineering of models from code.

A modeling language, according to [11], 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. The combination of notation and metamodel for a graphical modeling language serves roughly the same purpose as a context-free grammar does in a textual or ASCII-based modeling language – the combination explains how to form models, and describes the constraints on using the notation.

Metamodels serve several purposes that may be of interest to different modeling language clients.

- **Modelers:** metamodels explain to users of the modeling language what can legally be written and what cannot be written. Thus, metamodels should be easy to explain and understand, particularly by users of the language. Ideally, the metamodel should be expressed in such a way so that the fundamental details, such as the legal ways to connect modeling constructs, can be quickly and intuitively explained to users, without the need to understand much formalism and complexity. Formal details should be accessible to the users, if they are needed.

- **Tool Writers:** metamodels provide documentation for the tool writers who are building applications to support the modeling language, and who must construct their applications to obey the constraints of the metamodel. A metamodel can be viewed as a *specification* for a conformance checker that underlies tools for the modeling language. Thus, metamodels should not be open to misinterpretation, and should be unambiguous.

- **Modeling Language Designers:** metamodels provide a basis for the designers of the modeling language who have the responsibility of proving that their metamodel is consistent, i.e., that when combined the set of syntactic well-formedness constraints which are a part of the metamodel do not produce a contradiction. Thus, metamodels should also have a mathematical foundation so that proofs of consistency can be carried out. Of course, users of the language, and people developing tools for the language, may not be interested in the details of such proofs (though they may want to know that the proofs have been successfully carried out).

Because metamodels will be used by different people, with different goals, who require different characteristics of the metamodel, different versions of a metamodel are useful. On one hand, we will need precise versions of a metamodel that are easy for users of the modeling language to understand and explain to each other. On the other hand we will need formal versions that are amenable to automated reasoning, proof, and mathematical manipulation so that the modeling language can be shown to be consistent and unambiguous.

In this paper, we study an industrial-strength object-oriented modeling language, BON [14], and present its metamodel in two ways: as a precise model written in BON itself; and as a formal specification in the PVS specification language [5]. The precise description, written in BON, is aimed at promoting *understanding* of the high-level details of the metamodel (the abstractions and their connections). It should be possible for users of BON to quickly and informally understand the metamodel and some of the well-formedness constraints that it possesses, perhaps without necessarily understanding all of the technical details. On the other hand, the PVS specification is aimed at providing a mathematical description of the metamodel, one specifically that is amenable to automated manipulation and reasoning using the PVS theorem prover.

We suggest that both forms of description are useful in describing a metamodel. A precise description is useful to convey understanding; a formal description is useful for rigorous proof, automated reasoning, and consistency checking. We will also attempt to demonstrate the utility of constructing a precise metamodel before a formal one: the BON version of the BON metamodel will be constructed first, and will be used to assist in the production of the PVS version. Specifically, type information and structuring information that is present in the BON version will be used to help structure the PVS version of the metamodel.

1.1 Choice of BON and PVS

In writing a model, the choice of description language is important, in order to best produce a readable, understandable, clear, and consistent description. We chose to write two separate versions of the BON metamodel, because (a) we had two separate sets of requirements for the metamodel; (b) we were constructing the BON metamodel from scratch; and (c) no one modeling language provided all the needed features to satisfy all these requirements.

We selected BON for writing the precise metamodel because it satisfied several requirements.

- *Simplicity:* the BON modeling language is simple (possessing only three classifiers, as opposed to seven in UML [11]), which abets understandability and clarity [9]. The basic concepts in BON can be explained in several minutes to even novice OO modelers.
- *Expressiveness:* BON is expressive enough to precisely capture all the well-formedness constraints required, due to its assertion language being a dialect of first-order predicate logic.
- *Graphical:* BON possesses both a graphical and an ASCII-based syntax. The graphical syntax will be used to present the basic details of the metamodel; formulae will be used to present well-formedness constraints. A graphical description can convey complex details quickly and concisely; formal details are usually best presented using mathematical equations.

There are limitations with using a modeling language to describe its own metamodel: in order to understand the BON version of the metamodel and its well-formedness constraints, it is necessary, to a certain extent, to understand the BON metamodel. Even with this metacircularity, it is plausible and practical to understand the BON version of the metamodel, because BON is concise and its informal semantics is clearly and carefully defined.

We also chose to write the BON version of the metamodel before the PVS version, in part because of our experience with object-oriented modeling, but also because by using BON we could postpone dealing with issues of mathematical specification until we were certain that we had captured the appropriate abstractions. This is discussed more in Section 3.

We selected PVS for writing the formal metamodel because it satisfied several necessary requirements different from BON.

- *Automation:* the PVS theorem prover is industrial-strength, robust, well-documented, and proven in practice. Correspondingly, BON does not possess automated tools for reasoning.
- *Formality:* the PVS specification language has a formal semantics; correspondingly, BON does not (though see [8] for a partial semantics).
- *Expressiveness:* the PVS specification language, based on typed set theory, is expressive enough to capture the well-formedness constraints of the BON language.

Together, BON and PVS satisfy our requirements, and thus we will produce two versions of the metamodel, one expressed in BON, the other in PVS. The BON version will be constructed first, then the PVS version; thus, we will effectively describe a process that can be used for constructing formal metamodels of OO notations. We will strive to maintain consistency between the two versions, but since BON does not have a complete formal semantics, it will be impossible to prove that the two versions are consistent.

Both versions of the metamodel will be based on the context-free grammar given for the textual BON dialect in [14], as well as the informal text used to describe the semantics of abstractions in the same book.

1.2 Organization

We commence the paper with a brief overview of BON, focusing on its assertion language, the notation for class interfaces, and the notation for relationships. We also briefly describe the PVS system. We then informally summarize the basic concepts and terminology that we will be using in both versions of the metamodel. Then we present the BON version of the metamodel, starting with an abstract view – concentrating on subsystems and prominent abstractions – and then filling in details regarding the separate subsystems. Afterwards, we present the PVS specification language version of the metamodel. In doing so, we attempt to retain the structure of the BON version of the model, in order to abet readability and comprehension – ideally, a reader who has understood the structure of the BON metamodel, as well as some of the informal constraints of the BON version, should be able to decipher most of the PVS version of the metamodel with little difficulty.

We show how to use the PVS version of the metamodel, in combination with the PVS theorem prover, to carry out conformance checking of BON models against the metamodel. Several examples demonstrate the technique.

Finally, we conclude with a discussion, including an overview of some of the lessons that we learned, how what we have learned can be applied to other modeling languages (such as UML), and contrast our work with other formal descriptions of metamodels in the literature.

2 Background

In this section, we describe the BON object-oriented modeling language, focusing on its graphical syntax. We also briefly describe the PVS system, focusing on its specification language. Finally, we summarize some terminology and definitions that will be used throughout the remainder of the paper.

2.1 An overview of BON

BON is an object-oriented method possessing a recommended process as well as a graphical notation for specifying object oriented systems. The notation provides mechanisms for specifying inheritance and client-supplier relationships between classes, and has a small collection of techniques for expressing dynamic relationships. The notation also includes an *assertion language*; the method is predicated on the use of this assertion language for specifying contracts of routines and invariants of classes.

BON is designed to support three main techniques: seamlessness, reversibility, and design-by-contract. It provides a small collection of powerful specification features that guarantee seamlessness and full reversibility on the static specification notations. The design of the notation, and the reliance on design by contract, makes the implementation of seamlessness and round-trip engineering straightforward for all static notations, and supportable by tools, e.g., EiffelCase [3].

(In UML terms, we can view BON as equivalent to a UML profile for the Eiffel programming language. The profile includes only those aspects of UML that support seamless and reversible development.)

The fundamental specification construct in BON is the *class*. In BON, a class is both a module and a type. A BON class has a name, an optional class invariant, and a collection of features. A feature may be a *query*—which returns a value and does not change the system state—or a *command*, which does change system state but returns nothing. BON does not include a separate notion of *attribute*. Conceptually, an attribute should be viewed as a query returning the value of some hidden state information. All features are typed; routines may have (optional) domains and (in the case of queries) ranges. In general, types may be *reference* or *expanded*. Where a reference type is expected, an address is to be provided; where an expanded type is expected, an object of that type is expected.

Fig. 1 contains a short example of a BON graphical specification of the interface of a class $CITIZEN$. Class features are in the middle section of the diagram (there may be an arbitrary number of sections, annotated with visibility tags). Routines may optionally have behavioral specifications, 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 routine 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 (conjoined terms are separated by semicolons) that must be $true$ whenever an instance of the class is used by another object. In the invariant, the symbol $Current$ refers to the current object; it corresponds to `this` in C++ and Java.

The basic form of a BON assertion is

$$\forall\, x : T \mid R \bullet P$$

where variable x of type T is the bound variable, R is the domain restriction, and P is the propositional part.

In Fig. 1, class $CITIZEN$ has seven queries and one command. For example, $single$ is a query (which results in a $BOOLEAN$), while $divorce$ is a parameterless command that changes the state of an object. Class $SET[G]$ is a generic predefined class with generic parameter G and the usual operators (e.g., $\in$, add). The class $SET[CITIZEN]$ thus denotes a set of objects each of type $CITIZEN$.

Short forms of assertions are permitted. For example, consider a query $children : SET[CITIZEN]$. Then $\forall\, c \in children \bullet P$ is an abbreviation of $\forall\, c : SET[CITIZEN] \mid c \in children \bullet P$. The “it holds” operator $\bullet$ is right associative. The last invariant of Fig. 1 thus asserts that each child of a citizen must have that citizen as one of its parents. The first invariant asserts that if you are a citizen then you are either single or married to somebody who is married to you. The second invariant asserts that a citizen has exactly two parents. An advantage of the object-oriented BON style of specification is that as the class is enriched with new features (e.g., $children$ and $single$), the new features become part of the specification language that can be used for writing contracts.

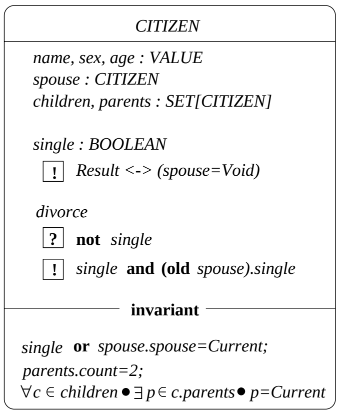
CITIZEN
name, sex, age : VALUE
spouse : CITIZEN
children, parents : SET[CITIZEN]
single : BOOLEAN
! Result <-> (spouse=Void)
divorce
? not single
! single and (old spouse).single
invariant
single or spouse.spouse=Current;
parents.count=2;
∀c ∈ children • ∃ p ∈ c.parents • p=Current

Figure 1: A citizen class in BON

BON specifications normally consist of multiple classes that interact via two kinds of different relationships.

- **Inheritance:** a child class inherits behavior from one or more parent classes. Inheritance is the subtyping relationship: everywhere an instance of a parent class is expected, an instance of a child class can appear. There is only one form of inheritance relationship in BON; however, the effect of the inheritance relationship on subtypes can be varied by changing the form of the parent classes; see [14]. The inheritance relationship is drawn between classes $ANCESTOR$ and $CHILD$ in Fig. 2, with the arrow directed from the child to the parent class. In this figure, we have drawn classes as ellipses – this is the compressed view of a class, used when no interface details need to be presented.

  Renaming mechanisms can be used to resolve name clashes and repeated inheritance conflicts that arise when using multiple inheritance. In a child class, features inherited from a parent can be tagged as being renamed. Renaming supports *joining* of features and *replicating* of features.

- **Client-supplier:** there are two basic client-supplier relationships, association and aggregation, which are used to specify the *has-a* or *part-of* relationships between classes, respectively. Both relationships are directed, from a *client* class to a *supplier* class. With association, the client class has an attribute that is a reference to an object of the supplier class. With aggregation, the deletion of an instance of a client class also implies the deletion of the corresponding instance of the supplier class; thus, the client class has an attribute that is an object of the supplier class. Aggregation corresponds semantically with the notion of *expanded type* discussed earlier. The aggregation relationship is drawn between classes $CHILD$ and $SUPPLIER1$ in Fig. 2, whereas the association relationship is drawn from class $CHILD$ to class $SUPPLIER2$. We say that $CHILD$ is a client of both $SUPPLIER1$ and $SUPPLIER2$.

2.2 A brief description of PVS

The PVS system [5] combines an expressive specification language with an interactive theorem prover and proof checker. The specification language is founded on classical typed higher-order logic, with typical base types `bool`, `nat`, `int`, `real`, et cetera. It also provides function constructors of the form `[A->B]`. The type system of PVS is augmented with dependent types and abstract data types, as well as predicate subtypes, the latter of which are a distinguishing feature of the specification language. The subtype `{x:A|P(x)}` consists of those elements of type `A` that satisfy the predicate `P`.

Predicates in PVS are elements of type `bool`; `pred[A]` is syntactic sugar for the function type `[A->bool]`. Sets are identified with their characteristic predicates. In general, type checking with predicate subtypes is undecidable. Thus, the PVS type checker generates proof obligations (called *type-correctness conditions (TCCs)*) in those cases

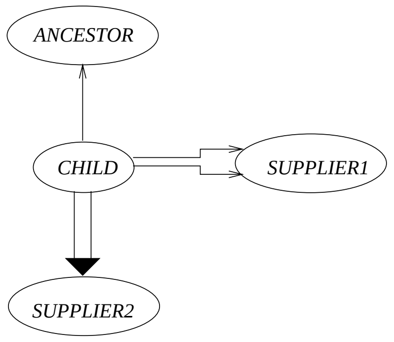

Figure 2: Class relationships in BON

where type conflicts cannot be resolved automatically. Frequently, a large number of TCCs are discharged by special proof strategies; a PVS expression is not fully type checked until all generated TCCs have been proved.

A built-in prelude and loadable libraries provide standard specification features (e.g., sets, sequences) and proved facts for a large number of theories. See [5] for further details.

2.3 Basic concepts and terminology

We briefly define some of the terminology and notions that will be used in both presentations of the BON metamodel.

A BON model is made up of *abstractions* and *relationships*. There are exactly four kinds of abstractions in a BON model:

1. Classes, which are also types. Each class has a name, a collection of *features*, and an optional *invariant* (which is a predicate).

2. Clusters, which contain sets of classes and other (nested) clusters. A cluster is a packaging construct, used to simplify and abstract away details in complicated models. Clusters have names. Clusters and classes combine to form the static abstractions that can appear in a BON model.

3. Objects, which are instances of classes. Objects have names, and have a type.

4. Object clusters, which contain collections of objects and other object clusters. Object clusters have names, and together with objects form the allowable dynamic abstractions.

Classes have a collection of features. In BON, a feature is either a query – a value – or a command (which is used to change the state of an invoking object). Queries have no side-effects; commands cannot return a value. Features have names and signatures, listing optional parameters and, in the case of queries, a type for the associated value.

Abstractions are connected via relationships. There are four kinds of relationships that may be present in a model.

1. Inheritance, which connects static abstractions. Semantically, inheritance is the subtyping relationship. An inheritance relationship is from a source (child) abstraction to a target (parent) abstraction.

2. Associations, which connect static abstractions. Semantically, inheritance is the *has-a* relationship. Associations are directed from a source (client) abstraction to a target (supplier) abstraction.

3. Aggregations, which also connect static abstractions. Semantically, aggregation is the *part-of* relationship. Aggregations are directed from a source (client) abstraction to a target (supplier) abstraction.

4. Messages, which connect dynamic abstractions. Semantically, messages correspond to method or feature calls. They are directed from a source abstraction to a target abstraction (which receives the message).

3 Specifying the Metamodel in BON

We specify the BON metamodel precisely, using BON. This is not the same as a formal specification of the metamodel: BON does not have a formal semantics. Thus, we do not intend the BON metamodel written in BON to be used for formal mathematical reasoning. The goals of presenting the BON metamodel in BON itself are as follows.

- To understand the metamodel, its important abstractions, and its structure, without having to be concerned with significant technical details, as would necessarily be the case if we were to specify the metamodel immediately in a formal notation like Z or the PVS specification language.
- To construct a preliminary draft of the BON metamodel that will be used in constructing a formal version of the metamodel.
- To construct an understandable, readable, and explainable version of the metamodel that may be useful for all users: tool writers, methodologists, and metamodelers.

Figure 3 contains a BON diagram depicting the basic metamodel. The two clusters, $RELATIONSHIPS$ and $ABSTRACTIONS$, will be further detailed in later subsections. The basic idea is that a model in BON consists of a set of relationships and a set of abstractions.

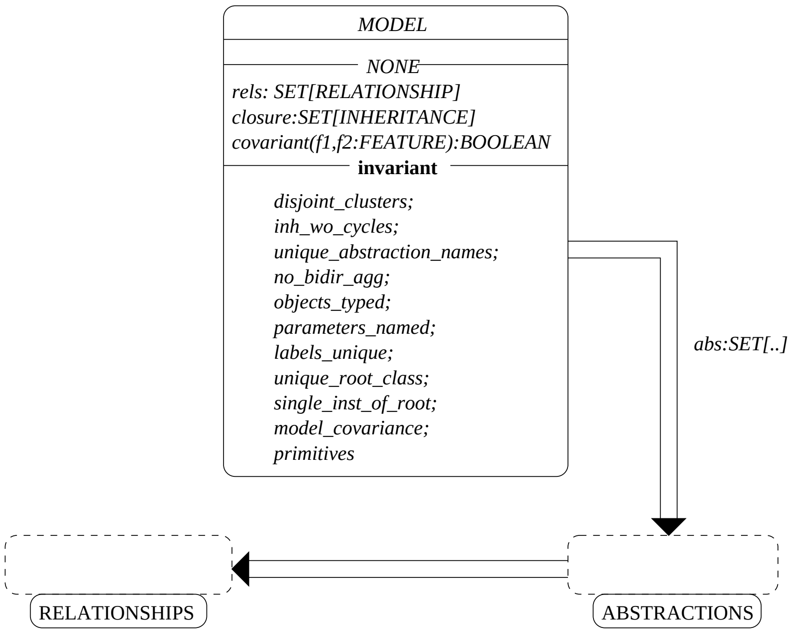

Figure 3: The BON metamodel, high-level view

Class $MODEL$, instances of which are BON models, contains a number of clauses in its invariant that constrain acceptable models. In order to understand each constraint in the invariant properly, we must first explain the clusters $RELATIONSHIPS$ and $ABSTRACTIONS$. Afterwards, we will return to the invariant and features of $MODEL$.

3.1 The relationships cluster

The $RELATIONSHIPS$ cluster contains details about the four basic BON relationships, as well as the type constraints on their use. The cluster is depicted in Fig. 4.

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

- *Type redefinition:* BON allows types of features to be redefined when they are inherited. A type in the signature of a feature can be replaced with a subtype (this is the covariant rule [3]). So, in class $RELATIONSHIP$, the attributes $source$ and $target$ are given types $ABSTRACTION$. But in the child classes $INHERITANCE$ and

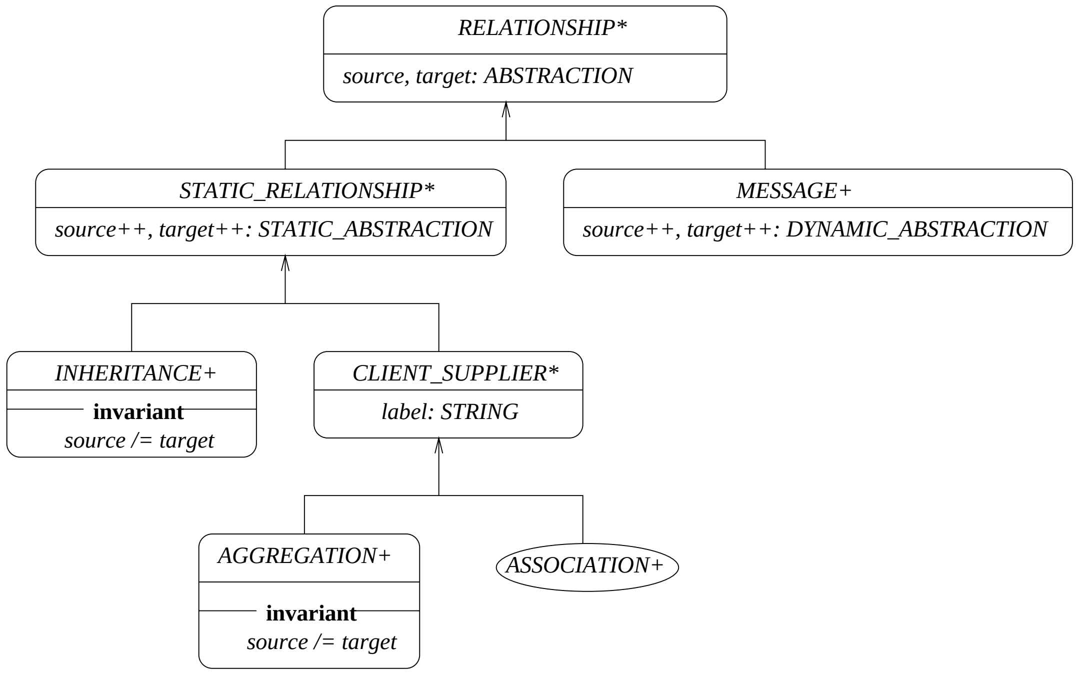

Figure 4: BON metamodel, relationship cluster

$CLIENT\_SUPPLIER$, the types are redefined to $STATIC\_ABSTRACTION$, because these relationships can only be applied to static abstractions. On the other hand, the types of $source$ and $target$ in class $MESSAGE$ are redefined to $DYNAMIC\_ABSTRACTION$, because messages only apply to such abstractions.

- *Messages:* messages are drawn between dynamic abstractions, i.e., objects and object clusters. No other information is associated with a message. Arbitrary text comments (e.g., numbers, informal descriptions of the purpose of a message) can be associated with a message. These are entirely informal and do not add to the semantics of the message at all. Such comments could easily be added to the metamodel by adding a $STRING$ query to the $MESSAGE$ class. This treatment of messages is in correspondence to the view of object communication diagrams being *rough sketches*.

- *Aggregation:* an aggregation relationship cannot target its own source. This is precisely captured by the class invariant on $AGGREGATION$. However, the same is not true of associations (references): the source of an association may be the same as the target, and there may be bidirectional associations (i.e., two classes may associate with each other).

- *Inheritance:* a class cannot inherit from itself. This is captured by the invariant of class $INHERITANCE$. It must also be guaranteed that there are no cycles in the inheritance graph, but this must be captured at the model level (because all inheritance relationships in the model, i.e., the transitive closure of the inheritance graph must be obtainable in order to verify this).

Note that we do not introduce a class for dynamic relationships that corresponds to the notion of a static relationship; this is because BON possesses only one dynamic relationship, the $MESSAGE$. For the sake of extensibility, it may be preferable to introduce a corresponding notion of *DYNAMIC_RELATIONSHIP*.

3.2 The abstractions cluster

The $ABSTRACTIONS$ cluster contains details about the basic abstractions that may appear in a BON model. It also expresses constraints on how these abstractions may be used. The cluster is depicted in Fig. 5.

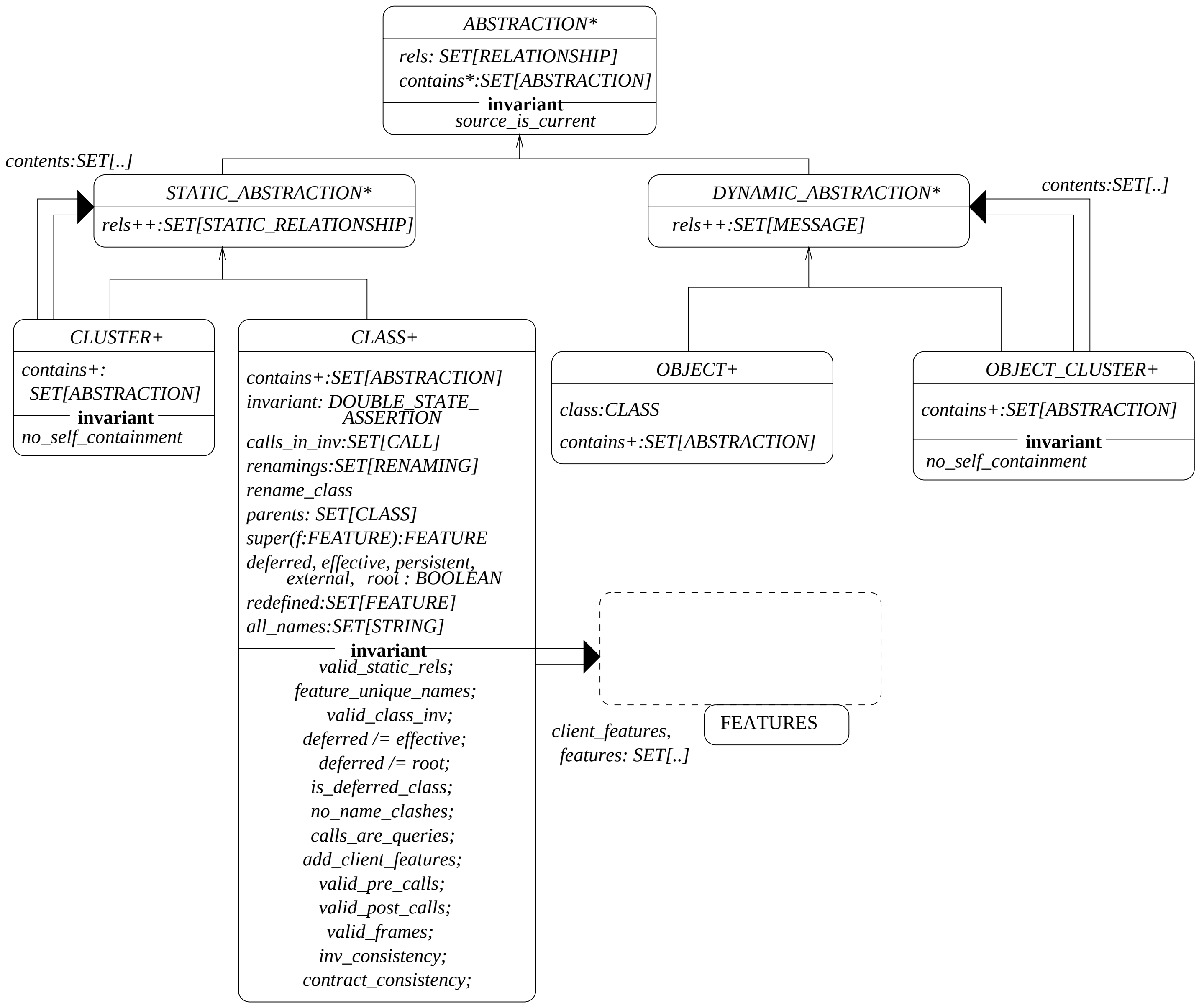

Figure 5: BON metamodel, abstractions cluster

The $FEATURES$ cluster will be described in detail in the next section. It contains the constraints and basic abstractions relevant to features. In particular, the cluster introduces a basic abstraction, $FEATURE$, representing a feature of a class.

The fundamental notion of an $ABSTRACTION$ is a deferred class: instances of $ABSTRACTION$ cannot be created. Classification is used to separate all abstractions into two subtypes: static and dynamic abstractions. Special static abstractions are $CLASS$es and $CLUSTER$s. Dynamic abstractions are $OBJECT$s and $OBJECT_CLUSTER$s. Clusters, both static and object, may contain other abstractions, but the well-formedness constraint on such a containment relation is that static clusters contain only static abstractions, and dynamic clusters contain only dynamic abstractions. This is enforced by the association relationships in Fig. 5.

Now we can express the constraints on abstractions. First off, each abstraction has to be the source of each relationship it is involved in.

$$source\_is\_current \;\; \widehat{=} \;\; \forall\, r \in rels \bullet r.source = Current$$

In each class, the names of features are unique. This is a constraint in the invariant of $CLASS$.

$$feature\_unique\_names \;\; \widehat{=} \;\; \forall\, f_1, f_2 \in features \bullet (f_1.name = f_2.name \rightarrow f_1 = f_2)$$

Each class has a (possibly empty) set of $renamings$ that are applied to inherited features. The class $RENAMING$ is a simple data structure that provides two queries, old_name and new_name, as well as mutator commands. One feature is provided with $CLASS$ that allows a model to rename inherited features. $rename\_class$ changes the name of all features that are included in the set of $renamings$.

$$\begin{array}{l} rename\_class \\ \mathbf{ensure}\ \forall\, f \in features \bullet (\exists\, r \in renamings \mid r.old\_name = \mathbf{old}\ f.name \bullet f.name = r.new\_name) \end{array}$$

Another feature provided is all_names, which returns the set of all names of features.

$$\begin{array}{l} all\_names : SET[STRING] \\ \mathbf{ensure}\ Result = \{s : STRING \mid (\exists\, f \in features \bullet s = f.name)\} \end{array}$$

It must be ensured that there are no name clashes due to inherited features sharing names with new features. The only features that may appear in both a parent and a child class are redefined features. This is guaranteed by the invariant clause $no\_name\_clashes$. This assertion is true iff the set of names of features in a class does not intersect with the set of names of features inherited from parents (and under all specified renamings).

$$\begin{array}{l} no\_name\_clashes \,\widehat{=} \\ \quad \{n : STRING \mid (\exists\, f \in features \bullet f.name = n \wedge \neg\, f.redefined)\} \cap \\ \quad \{n : STRING \mid (\exists\, p \in parents \bullet \exists\, f \in p.features \bullet \\ \qquad\qquad\qquad \exists\, r \in renamings \mid r.old\_name = f.name \wedge r.new\_name = n)\} = \varnothing \end{array}$$

Next, it must be guaranteed that all calls made to features in the invariant of a class are made according to the information hiding model: if the invariant of a class C calls a feature f of object o, then C must be given access to f, either explicitly on the access list for the feature f, or by specifying that f can be accessed by ANY client. Similar constraints will be included in the $FEATURES$ cluster for preconditions and postconditions. This constraint makes use of feature $isvalid$ of the hierarchy of feature $CALL$s (defined in Section 3.3).

$$valid\_class\_inv \;\; \widehat{=} \;\; \forall\, call \in calls\_in\_inv \bullet call.f : QUERY \wedge call.isvalid(Current)$$

A class is either deferred or effective (but not both). Deferral may only occur if the class has at least one deferred feature.

$$is\_deferred\_class \;\; \widehat{=} \;\; (deferred \Leftrightarrow \exists\, f \in features \bullet f.deferred) \wedge deferred \neq effective$$

A cluster (object or static) of abstractions cannot contain itself. The $no\_self\_containment$ constraint thus appears in both the class $CLUSTER$ and $OBJECT\_CLUSTER$.

$$no\_self\_containment \;\widehat{=}\; Current \notin contents$$

The constraint $valid\_static\_rels$ in class $CLASS$ ensures that all relationships in which a class is involved are valid. There are three components to validity (one component for each concrete type of relationship). For inheritance relationships, it must be ensured that the relationship targets a parent of the source.

$$\forall\, p \in parents \bullet \exists\, i : INHERITANCE \mid i.target = p \bullet i \in rels \tag{1}$$

For associations, there must be a feature corresponding to the association in the class.

$$\begin{array}{l}\forall\, f \in features \mid f : QUERY \bullet \\ \quad \exists\, a : ASSOCIATION \mid a.target = f.Result \bullet a \in rels\end{array} \tag{2}$$

Finally, for aggregations, there must be a subobject corresponding to the relationship. Subobjects are generated by parameterless queries with no contract.

$$\begin{array}{l}\forall\, a \in rels \mid a : AGGREGATION \wedge a.source = Current \bullet \\ \quad \exists\, f : QUERY \mid a.target = f.Result \wedge f.parameters.count = 0 \wedge f.no\_contract \bullet \\ \quad\quad f \in features\end{array} \tag{3}$$

The conjunction of (1), (2), and (3) forms the constraint $valid\_static\_rels$.

The feature $client\_features$ possessed by the abstraction $CLASS$ defines all features that are generated by client-supplier relationships. These features must also be added to the set $features$. This is captured by the following two expressions, which, conjoined, form the invariant clause $add\_client\_features$. The first expression establishes that each client-supplier relationship either generates a query in the source class of the relationship (when the source is a class), or it generates a query in *some* source class of the relationship (when the source is a cluster).

$$\begin{array}{l}\forall\, cs \in rels \mid cs : CLIENT\_SUPPLIER \bullet \\ \quad \exists\, q \in client\_features \mid q : QUERY \bullet q.name = cs.label \wedge \\ \quad\; (cs.target : CLASS \rightarrow q.Result = cs.target \wedge \\ \quad\; cs.target : CLUSTER \rightarrow \\ \quad\quad \exists\, ca \in cs.target.contains \mid ca : CLASS \bullet q.Result = ca)\end{array}$$

The second constraint establishes that all features generated by client-supplier relationships are also features of the class.

$$client\_features \subseteq features$$

The conjunction of the above two constraints forms the clause $add\_client\_features$ of class $CLASS$.

A class may make use of a number of entities in preconditions, postconditions, and class invariants. A consistency rule exists for such entities: an entity must either be $Current$ (i.e., a reference to the current object), $Result$, a feature of the class, or a parameter of the feature that makes the call in a contract. This consistency constraint is formalized in two invariant clauses of $CLASS$. The first, $inv\_consistency$, establishes that all invariant calls are either to the entity $Current$ or a feature.

$$\begin{array}{lcl}inv\_consistency & \widehat{=} & \forall\, call \in calls\_in\_inv \bullet \\ & & \quad (call.e.name = Current.name) \vee \\ & & \quad (\exists\, f \in features \bullet f.name = call.e.name)\end{array}$$

$Current$ and $Result$ are, of course, instances of the class $ENTITY$ which are in turn features of class $MODEL$.

$$Current, Result : ENTITY$$

A second constraint is needed, in $CLASS$, to establish the validity of calls made in preconditions and postconditions.

$$\begin{aligned} contract\_consistency \;\widehat{=}\; & \forall\, call \in \{call : CALL \mid \\ & \quad (\exists\, f \in features \bullet call \in f.calls\_in\_pre \cup f.calls\_in\_post)\} \bullet \\ & \quad call.e.name = Current.name \lor \\ & \quad call.f : QUERY \rightarrow call.e.name = Result.name \lor \\ & \quad \exists\, f1 \in features \bullet (f1.name = call.e.name) \lor \\ & \quad\; \exists\, i : INTEGER \mid f1.parameters.inrange(i) \bullet f1.parameters(i).name = call.e.name) \end{aligned}$$

Finally, the constraint $valid\_frames$ establishes that each feature in the class (either declared in the class, or generated by a client-supplier relation) has a valid frame, possessing only parameterless queries belonging to the class. Recall that in BON only commands of a class can change values of queries.

$$\begin{aligned} valid\_frames \;\widehat{=}\; & \forall\, f \in features \bullet \\ & \quad \forall\, q \in f.frame \bullet q \in features \land q.parameters.length = 0 \end{aligned}$$

3.3 The features cluster

The $FEATURES$ cluster describes the notion of a feature that is possessed by a class. Features may be queries (which provide a value) or commands (which change the state of an object responding to a message), but not both – BON does not allow ‘hybrid’ features, e.g., as in C++ or Java, which can both return a value and change the state of an object.

Features have optional parameters, an optional precondition, an optional postcondition, and an optional *frame* (optionality is expressed by allowing the set of queries that make up the frame to be empty). The pre- and postcondition are assertions, constructed using the BON assertion language (see [14] for details and a grammar) that includes boolean expressions, quantifiers, and set expressions. Query calls may appear in a boolean expression; the set of query calls that appear in the precondition and postcondition of a feature must be represented, in order to ensure that the calls are valid. To ensure the validity of an assertion, each feature will have a list of *accessors*, which are classes that may use the feature as a client. The frame is a set of queries, belonging to the class that has the feature, that the feature may modify. Queries themselves cannot modify any attributes. Finally, a feature has properties: it may be deferred (i.e., without implementation, wherein a subtype will provide an appropriate implementation), effective (implemented) or redefined. These properties correspond to stereotypes in UML.

In order to properly express the information hiding model of BON, it is necessary to model the concept of a query call (a call may occur in an assertion). There are two kinds of calls in BON: *direct calls*, which are of the form $e.f(a)$, where e is an entity, f a query, and a arguments; and, *chained calls*, which sequence together an ordered collection of feature calls, e.g., $e.f.g(a)$. The metamodel must express and constrain both types of calls, ensuring that it is legitimate for an invoking feature to make such calls.

Fig. 6 depicts the cluster. The assertion cluster is described in Fig. 7.

The assertion cluster simply describes the language of assertions in BON. There are two kinds of assertions: single-state assertions (e.g., as used in preconditions or class invariants, and which mention variables in a single state) and double-state assertions, e.g., as used in postconditions of commands. The assertion cluster depends on a class $EXPRESSION$ and a class $BOOLEAN\_EXPRESSION$, which define the expression language of BON. These classes contain features giving the type of the result of evaluating the expression, as well as the sequence of strings (tokens) that make up the expression. It is easy to determine what is an expression in BON; the context-free grammar for expressions is provided in [14].

The constraint no_old of class $SINGLE\_STATE\_ASSERTION$ guarantees that the keyword **old** is not used in the assertion. Similarly, no_result ensures that $Result$ is not used in a double-state assertion.

$$\begin{aligned} no\_old \;&\widehat{=}\; \mathbf{not}\; contents.contains(\text{“}old\text{”}) \\ no\_result \;&\widehat{=}\; \mathbf{not}\; contents.contains(\text{“}Result\text{”}) \end{aligned}$$

It is worth observing that the class $FEATURE$ specifies that in a BON model, a feature may possess a precondition and a postcondition, but makes no mention of implementation. It is not possible with BON (as defined in [14]) to

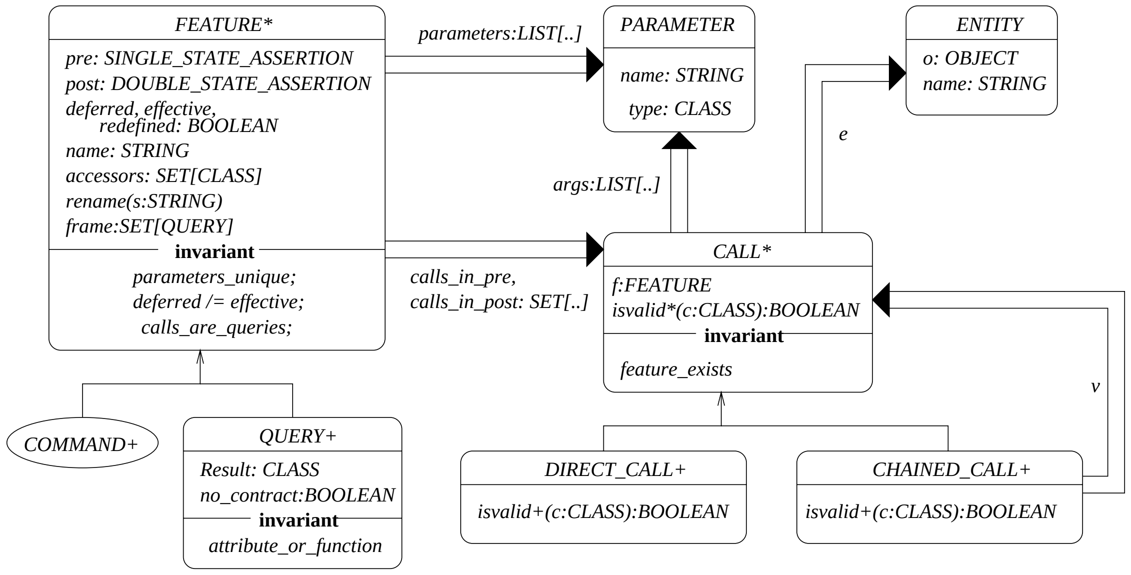

Figure 6: BON metamodel, features cluster

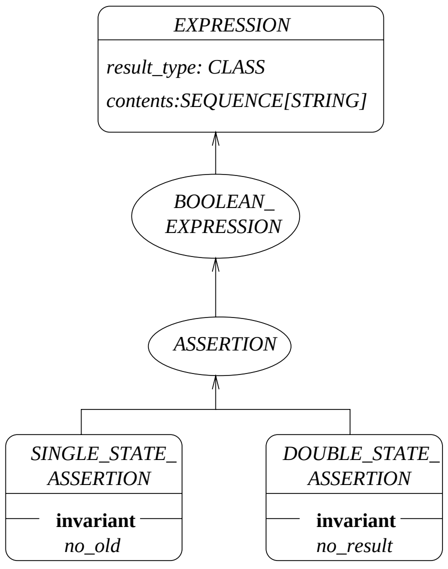

Figure 7: Assertion cluster

specify program code explicitly, thus the omission. However, in BON a contract is just a predicate, and programs can be expressed as predicates[1]. Alternatively, it might be useful to extend BON, as well as the metamodel, to allow features to have implementations. This could be done by producing child classes of $COMMAND$ and $QUERY$, where these child classes add a new feature $code : STRING$, which is an implementation of the feature's contract, in some programming language. The reason why two subclasses – one inheriting from $COMMAND$, the other from $QUERY$ – are necessary is that each subtype of $FEATURE$ will place different constraints on an implementation. For example, the program for a $COMMAND$ cannot make use of the variable $Result$; the program for a $QUERY$ should not assign values to any variables other than locals (i.e., side-effects should be prohibited).

We now move on to the constraints that belong in the features cluster. As explained previously, a feature has a possibly empty list of parameters (attributes can be viewed as queries with an empty list of parameters, providing that the query has no contract). For each feature, the names of parameters must be unique.

$$\begin{aligned} parameters\_unique \quad \widehat{=} \quad & \forall\, i : INTEGER \mid parameters.valid\_index(i) \bullet \\ & \quad \neg\ \exists\, j : INTEGER \mid parameters.valid\_index(j) \wedge j \neq i \bullet \\ & \qquad parameters.item(i) = parameters.item(j) \end{aligned}$$

Class $FEATURE$ also possesses a command that can be used to change the name of the feature.

$$\begin{array}{l} rename(s : STRING) \\ \mathbf{ensure}\ name = s \end{array}$$

The $CALL$ hierarchy possesses a shared query, $isvalid$, that validates a call against the information hiding model. $isvalid$ is declared as a deferred query in $CALL$, and is implemented in $CALL$'s descendents. Effectively, $isvalid$ in $DIRECT\_CALL$ checks to see if, for a call of the form $e.f(a)$, that the class containing the routine that is invoking f has been given permission, by the owner of f (which is e's class) to access the feature. Permission may be given explicitly for a class, or may be provided by specifying that ANY client can access the feature. For $CHAINED_CALL$s, the validation process must be recursive, i.e., each call in the chain must be validated against the information hiding model.

Here is the contract for $isvalid$ of class $DIRECT\_CALL$.

$$\begin{array}{l} isvalid(c : CLASS) : BOOLEAN \\ \mathbf{ensure}\ Result = (c \in f.accessors \vee ANY \in f.accessors) \end{array}$$

The contract for $isvalid$ of class $CHAINED\_CALL$ is similar, except for a call to the $isvalid$ feature of the successor in the chain of calls.

$$\begin{array}{l} isvalid(c : CLASS) : BOOLEAN \\ \mathbf{ensure}\ Result = ((c \in f.accessors \vee ANY \in f.accessors) \wedge v.isvalid(c)) \end{array}$$

We can now use the $isvalid$ features to establish that assertions appearing in a precondition or postcondition are valid with respect to the information hiding model. The constraints $valid\_pre\_calls$ and $valid\_post\_calls$ are used to accomplish this.

$$valid\_pre\_calls \quad \widehat{=} \quad \forall\, f \in features \bullet \forall\, call \in f.calls\_in\_pre \bullet call.isvalid(Current)$$

$$valid\_post\_calls \quad \widehat{=} \quad \forall\, f \in features \bullet \forall\, call \in f.calls\_in\_post \bullet call.isvalid(Current)$$

All calls made in the precondition or postcondition of the feature must be queries.

$$calls\_are\_queries \quad \widehat{=} \quad \forall\, c \in (calls\_in\_pre \cup calls\_in\_post) \bullet c.f.is\_query$$

We could, in fact, establish this property by a type constraint in the $CALL$ class itself (instead of declaring a feature $f : FEATURE$, declare $q : QUERY$). However, at some point we may want to extend the metamodel to include

[1]The paper [8] shows how to express a selection of Eiffel programming constructs as predicates, in the style of [1].

implementations, and therein calls to commands will also be possible. The invariant clause $attribute\_or\_function$ expresses that a query is either an attribute (in which case it has an empty list of parameters and no contract) or a function (in which case it has zero or more parameters, or a contract, or both, and its frame is empty).

$$\begin{aligned} attribute\_or\_function \quad = \quad & (no\_contract \wedge parameters.empty) \vee \\ & (parameters.empty \vee \neg\, no\_contract \rightarrow \\ & \ frame = \varnothing) \end{aligned}$$

The constraint $object\_takes\_message$ of class $CALL$ guarantees that, for a call $o.f$, the class of o possesses a feature f.

$$object\_takes\_message \quad \widehat{=} \quad f \in (obj.class.features \cup obj.class.client\_features)$$

3.4 Constraints on the model

The diagram in Figure 3, combined with Figs. 4 and 5, captures many of the details of the BON metamodel, but not all. Constraints on the use of relationships and the nesting of clusters must be added, in particular, to the invariant of $MODEL$. In this section, we precisely specify the constraints that are a part of the invariant of $MODEL$ in Fig. 3.

The first constraint that we capture more precisely is related to clusters: clusters cannot contain themselves, and separate clusters are either nested or disjoint. To accomplish this, we first specify the feature $contains$, which is declared in class $ABSTRACTION$ and which is inherited by each subtype. Informally, for an abstraction a, $a.contains$ is the set of all abstractions contained with a. If a is a class or an object, $a.contains$ is empty. If a is a cluster, $a.contains$ is the set of all classes, clusters, and objects contained within a or within a subcluster. Effectively, $contains$ unfolds any nesting that is present in a cluster.

For a cluster (object or class) the specification for $contains$ is as follows. Everything that is in the $contents$ feature, or that is contained in a nested cluster, is returned by $contains$. Here is the specification of $contains$ for static $CLUSTER$s (the specification for $OBJECT_CLUSTER$s is identical, except for the type returned by the function).

$$\begin{aligned} & contains : SET[STATIC\_ABSTRACTION] \\ & \textbf{ensure } Result = contents \cup \\ & \qquad \{c : STATIC\_ABSTRACTION \mid (\exists\, c1 \in contents \mid c1 : CLUSTER \bullet c \in c1.contains)\} \end{aligned}$$

Now, the constraint $disjoint\_clusters$ from class $MODEL$ can be specified as follows. This constraint establishes that cluster abstractions are either nested (either cluster a is in b or b is in a), or do not intersect.

$$\begin{aligned} disjoint\_clusters \quad \widehat{=} \quad & \forall\, a, b \in abs \mid a \neq b \wedge (a, b : CLUSTER \vee a, b : OBJECT\_CLUSTER) \bullet \\ & a \in b.contains \vee b \in a.contains \vee (a.contains \cap b.contains = \varnothing) \end{aligned}$$

The second constraint we capture guarantees that a static abstraction never inherits from another static abstraction that descends from the original abstraction, i.e., there are no cycles in the inheritance graph. To specify this, we first specify the function $closure$ of the class $MODEL$ that results in the set of all inheritance relationships in the model, plus all of the implicit relationships that exist due to the transitivity of inheritance.

$$\begin{aligned} & closure : SET[INHERITANCE] \\ & \textbf{ensure } Result = \{r \in rel \mid r : INHERITANCE\} \cup \\ & \qquad \{r : INHERITANCE \mid (\exists\, r_1, r_2 \in rel \mid r_1, r_2 : INHERITANCE \wedge r_1.source = r_2.target) \bullet \\ & \qquad\ r.source = r_2.source \wedge r.target = r_1.target)\} \end{aligned}$$

Now we can precisely capture the requirement that the inheritance graph has no cycles in it. This is carried out by calculating the inheritance closure and by ensuring that for each inheritance relation from abstraction A to abstraction B, there is no corresponding inheritance relation from B to A in the closure.

$$\begin{aligned} inh\_no\_cycles \quad \widehat{=} \quad & \forall\, r \in closure \bullet \neg\, \exists\, r_1 \in rel \mid r_1 : INHERITANCE \bullet \\ & (r.source = r_1.target \wedge r.target = r_1.source) \end{aligned}$$

Several straightforward constraints related to the model can now be captured. First, all abstractions that occur in relationships must be in the model.

$$abs\_in\_model \;\widehat{=}\; \forall\, r \in rel \bullet r.target \in abs \land r.source \in abs$$

Next, each abstraction in the model must have a unique name.

$$unique\_abs\_names \;\widehat{=}\; \forall\, a_1, a_2 \in abs \bullet (a_1.name = a_2.name \to a_1 = a_2)$$

It must also be ensured that the label of a client-supplier relationship does not clash with the name of any feature in the client abstraction, when the client is a class. Note that this constraint should *not* hold for cluster clients.

$$\begin{aligned} labels\_unique \;\widehat{=}\; & \forall\, r \in rel \mid r : CLIENT\_SUPPLIER \land r.source : CLASS \bullet \\ & \quad r.label \notin \{n : STRING \mid \exists\, f \in r.source.features \mid n = f.name\} \end{aligned}$$

In the case where the client is a cluster, the requirement is that at least one class contained within the cluster does not use the name on the label. To specify this, we first specify the function $all\_classes : SET[CLASS]$, which is a feature of class $CLUSTER$: it returns all the classes contained in a cluster, or in any nested sub-cluster.

$$\begin{aligned} & all\_classes : SET[CLASS] \\ & \textbf{ensure}\ Result = \{c : CLASS \mid c \in contains\} \cup \\ & \qquad\qquad \{c : CLASS \mid \exists\, c1 \in contains \land c \in c1.contains\} \end{aligned}$$

Then the unique label constraint for cluster clients can be stated as follows.

$$\begin{aligned} labels\_unique\_cl \;\widehat{=}\; & \forall\, r \in rel \mid r : CLIENT\_SUPPLIER \land r.source : CLUSTER \bullet \\ & \quad \exists\, c \in r.source.all\_classes \bullet r.label \notin c.all\_names \end{aligned}$$

We next capture the constraints on aggregation relationships: that they cannot originate from and target the same abstraction, and there cannot be bidirectional aggregation relationships. The first constraint is a clause of the invariant of $AGGREGATION$ (see Fig. 2). The second constraint must belong to the invariant of $MODEL$, since it can only be established by considering all aggregation relationships in the model.

$$\begin{aligned} no\_bidir\_agg \;\widehat{=}\; & \neg\, \exists\, r_1, r_2 \in rel \mid r_1, r_2 : AGGREGATION \land r_1 \neq r_2 \bullet \\ & \quad r_1.client = r_2.supplier \land r_1.supplier = r_2.client \end{aligned}$$

The next property of $MODEL$ that we capture will ensure that each object in the model is an instance of either a class in the model, or a primitive type (primitive types are declared in Fig. 8.

$$\begin{aligned} objects\_typed \;\widehat{=}\; & \forall\, o \in abs \mid o : OBJECT \bullet \\ & \quad \exists\, c \in abs \mid c : CLASS \bullet o.class = c \lor \\ & \quad (o.class : BOOLEAN \lor o.class : ANY \lor o.class : CHARACTER \lor \\ & \quad\; o.class : INTEGER \lor o.class : STRING \lor o.class : REAL) \end{aligned}$$

Parameters of features must obey constraints, but only with respect to the model itself, in that the type of each parameter, and the type of the result of a query, must be classes in the model.

$$\begin{aligned} parameters\_named \;\widehat{=}\; & \forall\, f \in \{g : FEATURE \mid \exists\, c \in abs \mid c : CLASS \bullet g \in c.features\} \bullet \\ & \quad (\forall\, p \in f.parameters \bullet p.type \in abs \land f : QUERY \to f.Result \in abs) \end{aligned}$$

There must also be no clashes between inherited features and features that are defined within a class.

$$no\_feature\_clashes \;\widehat{=}\; \forall\, c \in abs \mid c : CLASS \bullet c.features \cap c.inherited\_features.rename = \varnothing$$

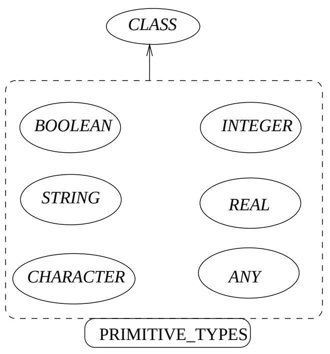

Figure 8: Primitive classes

A model may have a *root* class, a class that possesses a routine from which computation commences (in Java, the root class has a method called `main`). In BON, there can be no more than one root class and there must be at least one instance of the root in the model (though multiple instances of the root class are permitted).

$$unique\_root\_class \quad \widehat{=} \quad \forall\, c1, c2 \in abs \mid c1 : CLASS \wedge c2 : CLASS \bullet (c1.root \wedge c2.root) \rightarrow (c1 = c2)$$

To guarantee that there is at least one instance of the root, a constraint is written that states that of all the objects in the set of abstractions, at least one can have the $root$ property.

$$atleastone\_inst\_root \mathrel{\widehat{=}} \exists\, o \in abs \mid o : OBJECT \bullet o.class.root$$

All features that are redefined in a model must obey the *covariant* rule. Informally, this means:

- the result type of a query in a class may be replaced by a more specific type, when the query is inherited by a child class.
- the type of a parameter of a query or a command may be replaced by a more specific type in a child class.
- preconditions of features may be weakened.
- postconditions of features may be strengthened.

We do not capture the last two (semantic) properties in the metamodel; recall that a metamodel expresses the syntactic well-formedness constraints on models. To capture the first two aspects of covariance, we first must define a function $subtype(c1, c2 : CLASS) : BOOLEAN$, which returns $true$ if and only if $c2$ is a subtype of $c1$. BON allows type interrogation in contracts, specifically, by the colon operator ':'. This operator requires its right operand to be a class constant (e.g., $x : C$); it cannot be applied to arbitrary expressions in its right operand. Thus, we require a function that can be applied to expressions. The function works by returning true if the two arguments are the same class, or if there is an (explicit or implicit) inheritance relationship between the two class arguments in the inheritance graph.

$$\begin{array}{l} subtype(c1, c2 : CLASS) : BOOLEAN \\ \textbf{ensure } Result = (c1 = c2) \vee \\ \qquad\qquad (\exists\, i \in rels \mid (i : INHERITANCE \wedge i.source = c1 \wedge i.target = c2) \bullet \\ \qquad\qquad\quad i \in closure \end{array}$$

Next, we define the function $covariant(f1, f2 : FEATURE) : BOOLEAN$ which takes two features as arguments and returns $true$ iff $f1$ is a covariant redefinition of $f2$.

$$\begin{array}{l} covariant(f1, f2 : FEATURE) : BOOLEAN \\ \textbf{require } (f1 : QUERY \wedge f2 : QUERY) \vee (f1 : COMMAND \wedge f2 : COMMAND) \\ \textbf{ensure } (\forall\, i : 0..f1.parameters.length \Leftrightarrow 1 \bullet \\ \qquad\qquad subtype(f1.parameters.item(i).type, f2.parameters.item(i).type); \\ \qquad (f1 : QUERY \wedge f2 : QUERY \rightarrow subtype(f1.Result, f2.Result)) \end{array}$$

With $covariant$ defined, we can express the constraint that all feature redefinitions obey the covariant rule. Note that this is a system-level validity check.

$$model\_covariance \quad \widehat{=} \quad \forall\, c \in abs \mid c : CLASS \bullet \forall\, f \in c.redefined\_features \bullet covariant(f, c.super(f))$$

Finally, we must ensure that all relationships that are possessed by abstractions are also possessed by the model.

$$\forall\, a \in abs \bullet a.rels \subseteq rels$$

This completes the BON version of the metamodel. All features of BON have been captured, with two exceptions.

1. **Repeated inheritance.** An inheritance arrow can be adorned with a circle containing a number. The number indicates the number of inheritance relationships from a child class to a parent. This is syntactic sugar that can be removed and replaced with the appropriate number of inheritance arrows.

2. **Client-supplier multiplicities.** A client-supplier arrow can be adorned with a diamond containing a number indicating the number of relationships from client to supplier. As was for repeated inheritance, the diamond is syntactic sugar that can be removed and replaced with the appropriate number of client-supplier relationships.

3.5 Syntactic versus semantic metamodel constraints

In the context of object-oriented modeling languages, a metamodel is usually defined to be a specification of the well-formedness constraints that all models, written using the notation of the language, must obey. The metamodel of the Unified Modeling Language, for example, specifies the syntactic constraints on drawing models. It contains little or no information about *semantic* constraints. Semantic constraints on a model can take two forms:

- *semantic well-formedness constraints*, capturing, for example, scoping rules (e.g., that all methods used in a UML constraint in a model are declared in an appropriate modeling element, like a class; or that only public methods are used in a constraint outside of a class), type rules (e.g., that expressions are type-correct), or uniqueness rules (e.g., that parameters in a method have distinct names).

- constraints defining the *meaning* of modeling elements, e.g., the meaning of a class, type, interface, etc.

A semantic well-formedness constraint in a metamodel differs from a syntactic well-formedness constraint in the same way that semantic checking differs from syntactic checking in a compiler. The metamodel for BON that we have presented includes both syntactic and semantic well-formedness constraints, though no constraints defining the meaning of modeling elements. Some examples of semantic well-formedness constraints include:

- the constraints ensuring that all assertions in a model obey the information hiding model.

- the constraint ensuring that all abstractions in a model have unique names

- the constraint ensuring that when a feature f is called on an object o, the class of o has a feature f.

Our interpretation of a metamodel for an OO modeling language is that it is a specification that is to be used by tool builders. Tool builders must know all the relevant constraints that exist on constructing models. A good tool will enforce both the syntactic constraints – e.g., ensuring that only the proper modeling elements are allowed, and that legal connections between elements are used – as well as semantic well-formedness constraints. In principle, the semantic constraints could be left out of the metamodel and expressed in a separate document. They could be implemented in a tool that constructed a symbol table and performed type and scope checking on BON models. Instead, we capture all the well-formedness constraints in the metamodel, and implement them within the PVS system. In this way, we have a single tool for carrying out conformance checking, type checking, and scope control.

We now turn to a formal specification of the metamodel, captured in the PVS specification language.

4 Specifying the Metamodel in PVS

In this section, we present a formal specification of the BON metamodel in the PVS specification language. The formal specification will not be a direct translation of the precise metamodel described in the previous section: BON is an object-oriented language, while PVS is a theory-based language that uses functions and higher-order logic. In order to aid readability, we aim to preserve naming conventions as much as possible, and to parallel the structure of the BON metamodel in the PVS language, by using PVS theories and structuring mechanisms.

The reasons for using PVS's specification language to formally specify the metamodel are:

- The PVS specification language has a formal semantics that can be used for reasoning.
- PVS provides industrial-strength tool support that we can use to prove properties regarding the metamodel, e.g., consistency.
- We can use PVS to typecheck the metamodel as an added assurance as to its validity. Feedback from typechecking can be used to help debug the metamodel.

The approach we took to metamodeling in PVS was to, as closely as possible, mimic the structure of the BON specification in Section 4. To this end, the BON metamodel will consist of a set of abstractions and a set of relationships that are constrained by axioms. In the BON version of the metamodel, information hiding and encapsulation was used to structure the constraints on the model; for example, the constraint that cluster should be disjoint is a constraint on the model as a whole, and as such is expressed as part of the class invariant for $MODEL$. In PVS, there is a limited notion of information hiding. However, we can use the PVS `THEORY` construct to simulate information hiding to a certain degree.

We present the PVS version of the metamodel selectively, focusing on the critical aspects of the specification, and on particular strengths and weaknesses with PVS for this task. In producing the PVS version, we created several theories:

- `abs_names.pvs`: declares the names of all abstractions used in the metamodel.
- `rel_names.pvs`: declares the names of all relationships used in the metamodel.
- `assertions.pvs`: declares and defines the assertion language of BON.
- `abstractions.pvs`: defines the constraints on using the abstractions that exist in BON.
- `relationships.pvs`: defines the constraints on using the relationships that exist in BON.
- `metamodel.pvs`: the metamodel, which introduces global constraints on all the abstractions and relationships that are part of a model.

The two theories ending in `_names` are constructed to centralize all type declarations that need to be used by the other theories. We present the theories in order, focusing on the core elements that will allow us to capture the well-formedness constraints on BON models.

4.1 Theory of assertions

The theory of assertions simply declares a number of types, and a type hierarchy, that will be used by the other theories when assertions and boolean expressions need to be described. Effectively, this theory formalizes the BON model presented in Fig. 7. The theory declares several fundamental types: a type of expressions (which we do not define further – the BON grammar given in [14] formally specifies what is a valid expression), a type of boolean expressions, and two subtypes of assertions: single-state assertions (which are used for preconditions and invariants) and double-state assertions, used in the postconditions of commands and queries (note that even though a query cannot make reference to **old** variables in postconditions, a query can still take time to execute, and thus implicitly the postcondition can make reference to **old** values of a clock variable; see [7] for more on this issue).

```
assertions:THEORY
BEGIN
   % Basic type of expressions
   EXPR: TYPE+

   % Boolean expressions, a subtype of BON expressions.
   BOOLEAN_EXPR: TYPE+ FROM EXPR

   % Assertions are a special type of boolean expression. We
   % distinguish between boolean expressions and assertions for
   % extensibility reasons: if we later choose to talk about
   % implementation of features, we will need some way of
   % describing an executable language of boolean expressions.

   ASSERTION: TYPE+ FROM BOOLEAN_EXPR
   SINGLE_STATE_ASSERTION, DOUBLE_STATE_ASSERTION: TYPE+ FROM ASSERTION

   % The type of the result of evaluating the expression.

   result_type_expr: [ EXPR -> CLASS ]

   % Evaluator for boolean expressions.

   eval_expr: [ BOOLEAN_EXPR -> bool ]
   CONVERSION eval_expr
```

4.2 Theory of abstractions

The theory of abstractions presents the basic notions of abstractions that can appear in a BON model, and provides well-formedness constraints on how these abstractions can be used.

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. At the same time, we introduce primitive types, as subtypes of the class abstraction. This information is declared in the PVS theory `abs_names.pvs`.

```
abs_names: THEORY
BEGIN

  ABS: TYPE+

  % Subtypes: the two kinds of abstractions are static
  % abstractions and dynamic abstractions.

  STATIC_ABS, DYN_ABS: TYPE+ FROM ABS

  % Static abstractions are classes and clusters.

  CLASS, CLUSTER: TYPE+ FROM STATIC_ABS

  % Dynamic abstractions are objects and object clusters.

  OBJECT, OBJECT_CLUSTER: TYPE+ FROM DYN_ABS

  % Primitive classes are subtypes of CLASS.

  BOOLEAN_, ANY_, STRING_, REAL_, INTEGER_, CHARACTER_: TYPE+ FROM CLASS
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. The remaining abstraction concepts that we need to model include features (which may be queries or commands, and may optionally have parameters), feature calls (which may appear in an assertion associated with a feature), entities, parameters, and assertions.

```
abstractions: THEORY
  IMPORTING abs_names, rel_names, assertions

  % Features are queries or commands with optional parameters.

  FEATURE: TYPE+
  QUERY, COMMAND: TYPE+ FROM FEATURE

  % Entities appear in contracts; they can be attached to an object,
  % or they can be void. Note that an entity is not the same thing as
  % an object; an entity is used in a specification to refer to an
  % object. We provide functions to acquire the name of an entity and
  % the object attached to an entity.

  ENTITY: TYPE+
  nonvoid_entity: [ ENTITY -> bool ]
  entity_object: [ ENTITY -> OBJECT ]
  entity_name: [ ENTITY -> string ]
```

Parameters are modeled as functions from features to finite sequences of records. We also provide functions to acquire all relationships associated with a static or dynamic abstraction.

```
  % The type PARAM is used to represent a single parameter, consisting of a
  % name and a type.

  PARAM: TYPE = [# name:string, param_type: CLASS #]

  % A feature has a (possibly empty) finite sequence of parameters.

  IMPORTING sequences[PARAM]
  parameters: [ FEATURE -> finite_sequence[PARAM] ]

  % Relationships associated with abstractions. STATIC_REL and MESSAGE are
  % declared in the theory rel_names

  static_rels: [ STATIC_ABS -> set[STATIC_REL] ]
  dynamic_rels: [ DYN_ABS -> set[MESSAGE] ]
```

The BON version of the metamodel specified a hierarchy of different kinds of calls. We model an identical hierarchy, using subtyping, in PVS, declaring an abstract notion of a call, as well as two concrete subtypes. We also provide features to acquire the entity associated with a call, the feature being invoked, arguments associated with the invocation, and the remainder of a chained call. For example, if a call is of the form $e.f$ then e is the entity and f the feature being invoked. If a call is of the form $e.f.g$ then e is the entity, and $f.g$ is the remainder of the chained call.

```
  % A call has an entity, a feature, arguments, as well as functions to
  % be used to ensure that the call is valid.

  CALL: TYPE+
  DIRECT_CALL, CHAINED_CALL: TYPE+ FROM CALL

  % Features of calls.

  call_entity:  [ CALL -> ENTITY ]
  call_feature: [ CALL -> FEATURE ]
  call_args:    [ CALL -> finite_sequence[PARAM] ]

  % The remainder of a chained call.

  rest: [ CHAINED_CALL -> CALL ]

  % Features used to validate a call. These features will be used
  % in part to ensure that a call obeys the information hiding
  % model (a type check condition, to be sure, but one that is
  % important to capture in a metamodel that is to be used by
  % tool builders).

  call_isvalid: [ CALL, CLASS -> bool]
  direct_call_isvalid: [ DIRECT_CALL, CLASS -> bool ]
  chained_call_isvalid: [ CHAINED_CALL, CLASS -> bool ]
```

Primitive BON classes were modeled, in the `abs_names` theory, as subtypes of type `CLASS`. We may want to use instances of these classes in contracts. Thus, we need to declare representative instances of each primitive type and provide conversion routines to map instances into PVS types. Note that we only need and want one instance of each of `REAL_`, `INTEGER_`, etc., because these are effectively metaclasses and when instantiated will express classes.

We also declare two special constant entities, *Current* and *Result*, which can appear in contracts.

```
bool_object, real_object, string_object, integer_object,
  char_object, any_object: CLASS

% Conversions that the PVS typechecker can automatically apply.
% No conversion is provided for ANY objects (there is no way to
% express this in PVS.

interp_bool:   [ bool_object -> bool ]
interp_int:    [ integer_object -> int ]
interp_string: [ string_object -> string ]
interp_real:   [ real_object -> real ]
interp_char:   [ char_object -> character ]

CONVERSION interp_bool, interp_int, interp_string, interp_real, interp_char

Current, Result: ENTITY
```

We must now represent constraints on abstractions. In the BON version of the metamodel, this took the form of features and class invariants. In PVS, the well-formedness constraints will appear as functions 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.

object_class: [ OBJECT -> CLASS ]

% The name of an abstraction.

abs_name: [ ABS -> string ]

% The contents of an (object or class) cluster. Note that clusters may
% be nested.

cluster_contents: [ CLUSTER -> set[STATIC_ABS] ]
object_cluster_contents: [ OBJECT_CLUSTER -> set[DYN_ABS] ]

% The invariant of a class; the features of a class.

class_invariant: [ CLASS -> SINGLE_STATE_ASSERTION ]
class_features: [ CLASS -> set[FEATURE] ]

% Properties of a class. A class may be deferred, effective, a root,
% persistent, or external.

deferred_class, effective_class, persistent, external, root: [CLASS->bool]
```

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 useful information about features.

```
% The name of a feature.

feature_name: [ FEATURE -> string ]

% Precondition and postcondition of a feature.

feature_pre: [ FEATURE -> SINGLE_STATE_ASSERTION ]
feature_post:[ FEATURE -> DOUBLE_STATE_ASSERTION ]

% Properties of a feature. A feature may be deferred, effective, or redefined.

deferred_feature, effective_feature, redefined_feature: [ FEATURE -> bool ]

% The set of classes that can legally access a feature. Note that some
% features are private, in which case accessors will return the empty set.

accessors: [ FEATURE -> set[CLASS] ]

% The frame of a feature, consisting of parameterless, contractless queries.

feature_frame: [ FEATURE -> set[QUERY ] ]

% A function returning true iff a query has a contract. Used to establish
% whether a query can appear in a frame or not.

query_no_contract: [ QUERY -> bool ]

% The result of a query.

query_result_class : [ QUERY -> CLASS ]
```

It is frequently useful to be able to talk about the contents of a cluster, i.e., all features or all classes that are contained within a cluster. We introduce functions to let us accomplish this. We also need to be able to obtain all redefined features that belong to a specific class. Functions are also provided to acquire all abstractions contained within a cluster (including abstractions that are nested, i.e., if a cluster contains another cluster, everything in the innermost cluster will also be returned by these functions).

```
% Contents of clusters: all classes or features that
% belong to a static cluster.

all_classes:  [ CLUSTER -> set[CLASS] ]
all_features: [ CLUSTER -> set[FEATURE] ]

% All redefined features that belong to a class.

redefined_features: [ CLASS -> set[FEATURE] ]

% All abstractions contained within a cluster.

static_contains: [ CLUSTER -> set[STATIC_ABS] ]
dynamic_contains:[ OBJECT_CLUSTER -> set[DYN_ABS] ]
```

We want to be able to capture the concept of a legal set of calls. Consider an assertion in BON, e.g., a precondition or a class invariant. Such an assertion may call queries. It is important to be able to verify that all query calls are legal according to the information hiding model (i.e., that legal clients are calling the query). To accomplish this, we introduce functions that give us all the calls associated with a precondition, postcondition, and invariant. We also introduce a function that will provide us with all calls associated with a class (this will be useful in verifying a messaging model against a class model).

```
% All the calls that appear in an assertion.

calls_in_pre, calls_in_post: [ FEATURE -> set[CALL] ]

calls_in_inv: [ CLASS -> set[CALL] ]

calls_in_class: [ CLASS -> set[CALL] ]
```

Finally, to talk about renaming of features, we introduce a new type that embodies the concept of a renaming: a mapping from an old name to a new name. Each class thereafter possesses a set of renamings, which is applied to the features that it inherits from any parents. We also introduce two functions that can be used to either rename a feature or rename all features in a class (given a set of renamings).

```
   % The abstraction of a renaming.

   RENAMING: TYPE = [# old_name:string, new_name:string #]

   % The set of renamings associated with a class.

   renamings: [ CLASS -> set[RENAMING] ]

   % Functions that can be used to rename a single feature or all features
   % possessed by a class.

   rename_feature: [ FEATURE, string -> FEATURE ]
   rename_class:   [ set[FEATURE], set[RENAMING] -> set[FEATURE] ]

   % A useful function is all_names, which can be used to generate all
   % feature names, given a set of features. This is useful in ensuring
   % that there are no name clashes.

   all_names: [ set[FEATURE] -> set[string] ]
```

We now provide a number of examples of axioms, which define the constraints on BON models. The first example ensures that all features of a class have unique names (since BON does not permit overloading of features based upon signature information). The axiom states that if two features of a class have the same name then they must be the same feature.

```
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))
```

Two axioms define the function `all_names` (used to calculate all feature names associated with a class) and the function `rename_feature`, used to change the name of a feature.

```
all_names_ax: AXIOM
  (FORALL (f:set[FEATURE]):
    all_names(f) = { s:string | EXISTS (f1:FEATURE):
                          feature_name(f1)=s })

rename_feature_ax: AXIOM
  (FORALL (f:FEATURE): (FORALL (s:string):
     feature_name(rename_feature(f,s)) = s))
```

As an example of using the `rename_feature` function, we define the function `rename_class`, which applies a set of renamings to a class, and uses `rename_feature` in the process.

```
rename_class_ax: AXIOM
  (FORALL (c:CLASS):
    rename_class(class_features(c),renamings(c)) =
      { f:FEATURE | EXISTS (f1:FEATURE):
       (member(f1,class_features(c))
       AND
       EXISTS (r:RENAMING): member(r,renamings(c)) AND
          old_name(r) = feature_name(f1)
       IMPLIES
          rename_feature(f1,old_name(r))=f AND
          feature_name(f)=new_name(r)) }
  )
```

Two further axioms ensure that clusters (both static and dynamic) do not contain themselves.

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

  no_self_containment_ocl: AXIOM
    (FORALL (ocl:OBJECT_CLUSTER):
       not member(ocl, object_cluster_contents(ocl)))
```

All parameters of a feature in a class must have unique names.

```
parameters_unique: AXIOM
 (FORALL (c:CLASS) :
   (FORALL (f:FEATURE) : member(f,class_features(c)) IMPLIES
     (FORALL (i,j:{k:nat|k<length(parameters(f))}) :
       (i /= j) IMPLIES
       (name(parameters(f)(i)) /= name(parameters(f)(j))))))
```

The following axioms define the set of all classes contained within a cluster. Two axioms are needed – one to deal with top-level classes, the other to deal with nested clusters (similar axioms will be needed for object clusters as well).

```
all_classes_ax1: AXIOM
  (FORALL (cl:CLUSTER): (FORALL (c:CLASS):
  member(c,cluster_contents(cl)) IMPLIES member(c,all_classes(cl))))

all_classes_ax2: AXIOM
  (FORALL (cl,cl2:CLUSTER):
     member(cl,cluster_contents(cl2)) IMPLIES
     (FORALL (c:CLASS): member(c,all_classes(cl)) IMPLIES
     member(c,all_classes(cl2))))
```

Now we demonstrate how to specify that an assertion is valid according to the information hiding model. This, technically, is a type-check (semantic) condition that could be omitted from a metamodel. We show how to integrate such lightweight well-formedness constraints into a metamodel here.

Here is an example of specifying that an assertion is valid according to the information hiding model. The axiom `valid_precondition_calls` ensures that: (a) all calls in a precondition are legal (according to the accessor list for the 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)))))
```

The axioms for ensuring that postcondition calls and calls in a class invariant are valid are similar.

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

valid_class_invariant: AXIOM
  (FORALL (c:CLASS):
    (FORALL (call:CALL): member(call,calls_in_inv(c)) IMPLIES
        QUERY_pred(f(call)) AND
        call_isvalid(f(cal))))
```

These axioms all make use of the `isvalid` function that can be applied to a direct or a chained call. This function verifies the following.

- For a direct call of the form $o.f(a)$, the function ensures that the class containing the call (in an assertion) is given permission by the class of o to invoke feature f.

- For a chained call of the form $o.p.f(a)$, the function first checks that the class containing the call $o.p$ has permission (from the class of o) to access feature p, and then continues the process by ensuring that the call $p.f(a)$ is valid.

Here are the PVS specifications of `isvalid`, first for direct calls, then for chained calls.

```
dircall_isvalid_ax: AXIOM
  (FORALL (dir1:DIRECT_CALL): (FORALL (c:CLASS):
     direct_call_isvalid(dir1,c) =
       (member(c,accessors(call_feature(dir1))) OR
        member(any_object, accessors(call_feature(dir1))))))

chained_call_isvalid_ax: AXIOM
  (FORALL (ch1:CHAINED_CALL): (FORALL (c:CLASS):
     chained_call_isvalid(ch1,c) =
       (member(c,accessors(call_feature(ch1))) OR
        member(any_object,accessors(call_feature(ch1))) AND
       call_isvalid(rest(ch1),c)))))
```

In order for the chain of calls to `isvalid` within `chained_call_isvalid` to work, we must provide an axiom defining the behaviour of `call_isvalid` on different types of calls (this was provided automatically in BON due to dynamic binding and polymorphism. We effectively mimic this in PVS by using subtypes and axioms.)

```
call_isvalid_ax: AXIOM
  (FORALL (call:CALL): (FORALL (c:CLASS):
     (DIRECT_CALL_pred(call) IMPLIES
        call_isvalid(call,c)=direct_call_isvalid(call,c))
      AND
     (CHAINED_CALL_pred(call) IMPLIES
        call_isvalid(call,c)=chained_call_isvalid(call,c))))
```

We provide an axiom related to frames. It ensures that each feature of a class has a valid frame, made up only of parameterless queries without contracts, where the queries are declared within the class itself.

```
valid_frame_ax: AXIOM
  (FORALL (c:CLASS): (FORALL (f:FEATURE):
    member(f,class_features(c)) IMPLIES
      (FORALL (q:QUERY): member(q,feature_frame(f)) IMPLIES
         member(q,class_features(c)) AND length(parameters(q))=0 AND
         query_no_contract(q))))
```

The function `calls_in_class` extracts all calls associated with a class, whether they appear in an invariant clause or in a pre- or postcondition associated with a class feature. The function can then be used to verify that all calls in a class are legal; that is, if there is a call $obj.f$, the class of obj possesses a feature f. Note that this axiom ignores information hiding (that is captured by the previous axioms).

```
calls_in_class_ax: AXIOM
  (FORALL (c:CLASS): calls_in_class(c) =
     union( calls_in_inv(c),
            { call:CALL | (EXISTS (f:FEATURE):
             (member(f,class_features(c)) OR
              member(f,client_features(c)))  IMPLIES
              member(call,calls_in_pre(f)) OR
              member(call,calls_in_post(f)))
             }))
```

And here is the axiom to ensure that all calls are legal (in the sense that a call $o.f$ is legal only when the class of o provides a feature f).

```
valid_call_ax: AXIOM
  (FORALL (c:CLASS): member(c,abst) IMPLIES
    (FORALL (call:CALL): member(call,calls_in_class(c)) IMPLIES
      member(f(call), class_features(object_class(obj(call)))) OR
      member(f(call), client_features(object_class(obj(call))))))
```

Axioms are also needed to ensure that all uses of entities are legal. This is captured in the following two axioms. An entity is considered to be legal if it is the name *Current* or *Result*, the name of a feature of a class, or the name of a parameter of the feature that owns the contract. The first axiom establishes the validity of class invariants.

```
inv_consistency_ax: AXIOM
  (FORALL (c:CLASS): (FORALL (call:CALL):
     member(call,calls_in_inv(c)) IMPLIES
     (entity_name(call_entity(call)) = entity_name(Current)) OR
     (EXISTS (f:FEATURE):
        member(f,class_features(c) IMPLIES
        feature_name(f) = entity_name(call_entity(call)))))
```

The second axiom ensures the validity of contracts, i.e., pre- and postconditions.

```
contract_consistency_ax: AXIOM
  (FORALL (c:CLASS): (FORALL (call:CALL):
     (EXISTS (f:FEATURE): (member(f,class_features(c)) AND
       member(call,union(calls_in_pre(f),calls_in_post(f)))) IMPLIES
     ( (entity_name(call_entity(call)) = entity_name(Current)) OR
       (entity_name(call_entity(call)) = entity_name(Result)) OR
       (EXISTS (f1:FEATURE): member(f1,class_features(c)) IMPLIES
          (entity_name(call_entity(call)) = feature_name(f1)) OR
          (EXISTS (i:{k:nat|k<length(parameters(f1))}):
             entity_name(call_entity(call)) = name(parameters(f1)(i))))))))
```

Classes may have properties, e.g., they may be deferred or effective. Here are two examples, showing that a class cannot be both deferred and effective, and that if a class is deferred, it cannot be the root.

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

  deferred_root_ax: AXIOM
    (FORALL (c:CLASS):(NOT (deferred_class(c) IFF root(c))))
```

A deferred class has at least one deferred feature. Correspondingly, a class is effective if all its features are effective. Finally, a feature cannot be both deferred and effective.

```
deferred_class_ax: AXIOM
  (FORALL (c:CLASS):
    (deferred_class(c) IFF
    (EXISTS (f:FEATURE):
      member(f,class_features(c)) IMPLIES deferred_feature(c,f))))

effective_class_ax: AXIOM
  (FORALL (c:CLASS):
    (effective_class(c) IFF
    (FORALL (f:FEATURE):
       member(f,class_features(c)) IMPLIES effective_feature(c,f))))

deferred_feature_ax: AXIOM
  (FORALL (c:CLASS): (FORALL (f:FEATURE):
    member(f,class_features(c)) IMPLIES
     (NOT (deferred_feature(c,f) IFF effective_feature(c,f)))))
```

We define a function that will let us calculate all the redefined features that are associated with a class.

```
redef_feature_ax: AXIOM
  (FORALL (c:CLASS):
     redefined_features(c) =
     { f:FEATURE | member(f,class_features(c)) AND
                   redefined_feature(c,f) })
```

Finally, we define the functions to calculate all abstractions contained within a static or object cluster. These functions effectively unfold all nesting that may exist within a cluster.

```
static_contains_ax1: AXIOM
  (FORALL (cl:CLUSTER):
    static_contains(cl) =
      union(cluster_contents(cl),
            { g1:STATIC_ABS | (EXISTS (cl2:CLUSTER):
                                member(cl2,cluster_contents(cl)) IMPLIES
                                member(g1,static_contains(cl2))) }))

dynamic_contains_ax1: AXIOM
  (FORALL (ocl:OBJECT_CLUSTER):
    dynamic_contains(ocl) =
      union(object_cluster_contents(ocl),
            { d1:DYN_ABS | (EXISTS (ocl2:OBJECT_CLUSTER):
                             member(ocl2,object_cluster_contents(ocl)) IMPLIES
                             member(d1,dynamic_contains(ocl2))) }))
```

4.3 Theory of relationships

The theory of relationships (PVS files `rel_names.pvs` and `relationships.pvs`) declares and defines the three basic relationships that exist in BON: associations (reference relationships), aggregations (part-of or sub-object relationships) and inheritance (subtyping). Several constraints must be made on relationships, including that inheritance and aggregation relationships cannot be self-targeted. Further constraints on relationships must be specified in the metamodel itself, when we can access the set of all relationships.

To express 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. We first declare all of the names and types that we need, in `rel_names.pvs`, and then constrain these types further in `rel_names.pvs`.

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

    REL: TYPE+

    % Specific relationships: client-supplier, messages, and inheritance.

    INH, C_S, MESSAGE: TYPE+ FROM REL

    % Specific subtypes of client-supplier: association and aggregation.

    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 targeted). Thus, each relationship has a source and a target, and these concepts are modeled using PVS functions. The other concept we need to model is the label on a client-supplier relationship

```
  relationships: THEORY
  BEGIN
    IMPORTING rel_names, abs_names

    % The source and target of a relationship.

    source_rel, target_rel:     [ REL -> ABS ]
    cs_source, cs_target:       [ C_S -> STATIC_ABS ]
    inh_source, inh_target:     [ INH -> STATIC_ABS ]
    agg_source, agg_target:     [ AGG -> STATIC_ABS ]
    assoc_source, assoc_target: [ ASSOC -> STATIC_ABS ]
    msg_source, msg_target:     [ MESSAGE -> DYN_ABS ]

    % The label on each client-supplier relationship.

    cs_label: [ C_S -> string ]
```

It is important to note the range type of the more specific functions, e.g., `inh_source`. An inheritance relationship, client-supplier relationship, and message can actually be between static and dynamic abstractions, and not just classes or objects. Thus, for example, an inheritance relationship can be directed from a child class to a *cluster*; the intended meaning of such a relationship is that the child class inherits features from all classes contained or nested within the parent cluster.

```
 source_rel_ax: AXIOM
   ((FORALL (i:INH): source_rel(i)=inh_source(i)) AND
    (FORALL (a:AGG): source_rel(a)=agg_source(a)) AND
    (FORALL (m:MESSAGE): source_rel(m)=msg_source(m)) AND
    (FORALL (c:C_S): source_rel(c)=cs_source(c)) AND
    (FORALL (as:ASSOC): source_rel(as)=assoc_source(as)) AND
    (FORALL (a:AGG): cs_source(a)=agg_source(a)) AND
    (FORALL (as:ASSOC): cs_source(as)=assoc_source(as)))

target_rel_ax: AXIOM
  ((FORALL (i:INH) : target_rel(i) = inh_target(i)) AND
   (FORALL (a:AGG) : target_rel(a) = agg_target(a)) AND
   (FORALL (m:MESSAGE) : target_rel(m) = msg_target(m)) AND
   (FORALL (c:C_S) : target_rel(c) = cs_target(c)) AND
   (FORALL (as:ASSOC) : target_rel(as) = assoc_target(as)) AND
   (FORALL (a:AGG): cs_target(a)=agg_target(a)) AND
   (FORALL (as:ASSOC): cs_target(as)=assoc_target(as)))

 % Inheritance relationships cannot be from an abstraction to itself.

 inh_ax: AXIOM
   (FORALL (i:INH): NOT (inh_source(i)=inh_target(i)))

 % Aggregation relationships cannot be from an abstraction to itself.

 agg_ax: AXIOM
   (FORALL (a:AGG): NOT (agg_source(a)=agg_target(a)))
```

The theory of relationships is quite simple, because many of the constraints on using relationships have to be specified at the metamodel level. These *global* constraints 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 the metamodel itself.

4.4 The metamodel theory

The PVS file `metamodel.pvs` 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. 3: a model consists of a set of abstractions. The set of relationships we express is, as in the BON version, a convenience that makes specification of some of the operations and constraints simpler. Further well-formedness constraints must be added to restrict how abstractions and relationships are used in the context of a model.

```
metamodel: THEORY
BEGIN
IMPORTING abstractions, relationships

% A BON model consists of a set of abstractions.

abst:  SET[ABS]

% We also calculate all relationships in a model, since it
% makes specification of operations easier.

rels: SET[REL]
```

Two axioms are needed to calculate the set of all relationships. The first is used to determine all the static relationships, the second dynamic relationships.

```
rels_ax1: AXIOM
  (FORALL (sa:STATIC_ABS): member(sa,abst) IMPLIES
   (FORALL (sr:STATIC_REL): member(sr,static_rels(sa)) IMPLIES
      member(sr,rels)))

rels_ax2: AXIOM
  (FORALL (da:DYN_ABS): member(da,abst) IMPLIES
    (FORALL (m:MESSAGE): member(m,dynamic_rels(da)) IMPLIES
      member(m,rels)))
```

Now we must write constraints on how models can be formed from a set of relationships and abstractions. The first constraint we write ensures that inheritance hierarchies do not have cycles. We express this by computing 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 the inheritance relationship.

```
% Inheritance closure: the set of all explicit or implicit (due to
% transitivity) inheritance relationships in a model.

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):
        inh_source(r)=inh_source(r2) AND
        inh_target(r)=inh_target(r1) AND
        member(r,inh_closure)))
```

To specify that inheritance relationships do not generate cycles, we assert that if there is an inheritance relationship from abstraction A to abstraction B, there cannot be a different inheritance relationship from B to A in the closure.

```
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)))
```

It must be established that each relationship that is owned by an abstraction must have that abstraction as the source of the relationship. Also, it must be guaranteed that all abstractions that are the source or target of any relationship must also be in the model.

```
source_rel_ax1: AXIOM
  (FORALL (sa:STATIC_ABS): (FORALL (sr:STATIC_REL):
    member(sr,static_rels(sa)) IMPLIES source_rel(sr)=sa))

source_rel_ax2: AXIOM
  (FORALL (da:DYN_ABS): (FORALL (m:MESSAGE):
    member(m,dynamic_rels(da)) IMPLIES source_rel(m)=da))

abst_in_model: AXIOM
  (FORALL (r3:REL): member(r3,rels)
    IMPLIES
  (member(source_rel(r3),abst) AND member(target_rel(r3),abst)))
```

Several useful functions need to be defined in the metamodel, particularly those related to calculating the effect of inheritance. Specifically, given a class, it is useful to be able to determine all its inherited features as well as the parents of the class (if any). Such functions would then be used, for example, in ensuring that there will be no name clashes due to inheritance, and that the child class modifies the behaviour of the parent classes according to the rules of BON. We also calculate the set of all classes in the model.

```
% All classes in a model.

model_classes: set[CLASS] = { c:CLASS | member(c,abst) }

% Given a class, inherited_features returns the set of features
% that the class acquires from all its parents. The set may be
% empty, in the case where the class has no parents.

inherited_features: [ CLASS -> set[FEATURE] ]

% Define the behaviour of inherited_features.

inh_feature_ax1: AXIOM
  (FORALL (c:CLASS): member(c,abst) IMPLIES
     (FORALL (i:INH):
        (member(i,inh_closure) AND inh_source(i)=c AND
         CLASS_pred(inh_target(i)))
     IMPLIES
     (FORALL (f:FEATURE):
        member(f,class_features(inh_target(i))) IMPLIES
        member(f,inherited_features(c)))))

 % A second axiom is needed for the case where the target of an
 % inheritance relationship is a cluster.

 inh_feature_ax2: AXIOM
   (FORALL (c:CLASS): member(c,abst) IMPLIES
      (FORALL (i:INH):
        (member(i,inh_closure) AND inh_source(i)=c AND
         CLUSTER_pred(inh_target(i)))
      IMPLIES
      (FORALL (f:FEATURE):
         member(f,all_features(inh_target(i))) IMPLIES
         member(f,inherited_features(c)))))
```

It is also useful to be able to obtain, given a class and a specific feature belonging to the class, the *parent's* version of the feature. This corresponds to the notion of the `super` method available in the Java programming language. We first provide a function to calculate the set of parents of a class, and then specify the $super$ function.

```
parents: [ CLASS -> set[CLASS] ]

parents_ax: AXIOM
  (FORALL (c:CLASS): member(c,abst) IMPLIES
    (EXISTS (si:set[INH]): subset?(si,rels) AND
      (FORALL (i:INH): (member(i,si) AND inh_source(i)=c) IMPLIES
      parents(c) = { d:CLASS | member(d,abst) AND
                     ((CLASS_pred(inh_target(i)) IMPLIES inh_target(i)=d)
                     AND
                     (CLUSTER_pred(inh_target(i)) IMPLIES
                        member(d,all_classes(inh_target(i))))) } )))

 % The super function returns a parent's version of a feature, given
 % a specific class and feature.

 super: [ CLASS,FEATURE -> FEATURE ]

 super_ax: AXIOM
   (FORALL (c:CLASS): (FORALL (f1,f2:FEATURE):
     (member(c,abst) AND member(f1,inherited_features(c)) AND
      member(f2,class_features(c)) AND redefined_feature(c,f2) AND
      feature_name(f1)=feature_name(f2))
     IMPLIES
     super(c,f2) = f1))
```

Features of a class may also be generated via client-supplier relationships. The function `client_features` generates all such features. Two axioms are needed, one for the case where the target of a client-supplier relationship is a class, the second where the target is a cluster. In the second axiom, it is asserted that there is a class contained within the target cluster that is used as the result type of the generated feature.

```
% client_features is the set of all features generated by client-
% supplier relationships.

client_features: [ CLASS -> set[FEATURE] ]

client_ax1: AXIOM
  (FORALL (c:CLASS): (FORALL (cs:C_S): member(cs,static_rels(c)) AND
     CLASS_pred(cs_target(cs))  IMPLIES
  (EXISTS (q:QUERY): member(q,client_features(c)) IMPLIES
     feature_name(q) = cs_label(cs) AND
     query_result_class(q) = cs_target(cs))))

client_ax2: AXIOM
  (FORALL (c:CLASS): (FORALL (cs:C_S): member(cs,static_rels(c)) AND
     CLUSTER_pred(cs_target(cs)) IMPLIES
  (EXISTS (q:QUERY): member(q,client_features(c)) IMPLIES
    feature_name(q) = cs_label(cs) AND
    (EXISTS (ca:CLASS): member(ca,all_classes(cs_target(cs)))
    IMPLIES
    query_result_class(q) = ca))))
```

All features that are in the set `client_features` must also be in the set of class features. Note that a separate axiom guarantees that the client features introduce no name clashes.

```
add_client_features: AXIOM
  (FORALL (c:CLASS): member(c,abst) IMPLIES
     (FORALL (f:FEATURE): member(f,client_features(c)) IMPLIES
                          member(f,class_features(c))))
```

Two further functions will be used, later, in ensuring covariant redefinition of features. In BON, a feature's signature can be redefined to a subtype. The function `is_subtype` is used to check this.

```
% is_subtype returns true iff the second argument is a (BON) subtype
% of the first. Subtyping in BON means that a subtype (subclass) is
% a (not necessarily proper) descendent of a supertype (superclass).

is_subtype: [ CLASS,CLASS -> bool ]

is_subtype_ax: AXIOM
  (FORALL (c1,c2:CLASS):
    ((c1=c2) OR
    (EXISTS (i:INH): inh_source(i)=c1 AND inh_target(i)=c2 AND
                     member(i,inh_closure)))
    IFF is_subtype(c1,c2))
```

Two properties must be checked to ensure proper covariant redefinition. The first syntactic property is that classes that appear in the signature of a feature can be changed to subclasses. The second property is that contracts can be changed to promise more than the original contract. The latter is a semantic property that will not be checked in the metamodel (this would be checked by a tool that allows modelers to prove properties about their model – see Section 5 for discussion on this).

```
% The function covariant takes two features and results in true
% iff the second feature covariantly redefines the first.
% This function checks _only_ the syntactic aspects of covariance,
% i.e., signature redefinitions.

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)))

covariant_ax2: AXIOM
  (FORALL (com1,com2:COMMAND):
     covariant(com1,com2) IFF
     length(parameters(com1))=length(parameters(com2)) AND
     (FORALL (i:{j:nat|j<length(parameters(com1))}):
        is_subtype(param_type(parameters(com1)(i)),
                   param_type(parameters(com2)(i))))
```

The primary purpose of introducing the `covariant` function is to ensure that all classes that carry out a redefinition of a feature obey the covariance rule. This is accomplished by the axiom `model_covariance`. Note that model covariance is, in general, part of the behavioural subtyping rule which BON (and Eiffel) obeys. Behavioural subtyping includes both syntactic and semantic constraints (we do not model the semantic constraints in the metamodel).

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

All objects that appear in the model have a class for a type. This class must either be in the model (in which case it cannot be deferred), or it must be a primitive type (e.g., $INTEGER$).

```
objects_typed: AXIOM
  (FORALL (o:OBJECT): member(o,abst) IMPLIES
    ((member(object_class(o),abst) AND
      NOT deferred_class(object_class(o)))
    OR
      BOOLEAN__pred(object_class(o)) OR
      REAL__pred(object_class(o)) OR
      INTEGER__pred(object_class(o)) OR
      STRING__pred(object_class(o)) OR
      CHAR__pred(object_class(o))))
```

We next write axioms to guarantee that all clusters in a model are either nested or disjoint, and that each abstraction in a model has a distinct name.

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

disjoint_dynamic_clusters: AXIOM
  (FORALL (d1,d2:OBJECT_CLUSTER):
   (member(d1,abst) AND member(d2,abst) AND d1/=d2) IMPLIES
     (member(d1,dynamic_contains(d2)) OR member(d2,dynamic_contains(d1))
     OR
    empty?(intersection(dynamic_contains(d1),dynamic_contains(d2)))))

% Each abstraction in a model must have a unique name.

unique_abst_names: AXIOM
  (FORALL (a1,a2:ABS): (member(a1,abst) AND member(a2,abst))
    IMPLIES
  (abs_name(a1)=abs_name(a2) IMPLIES a1=a2))
```

It must also be ensured that there are no bidirectional aggregation relationships (i.e., one abstraction cannot be a sub-structure of a second abstraction, which is also a sub-structure of the first abstraction) and that no name clashes are allowed in a class.

```
% No bidirectional aggregation relationships are allowed.

no_bidir_agg: AXIOM
  (NOT (EXISTS (r1,r2:AGG): (member(r1,rels) AND member(r2,rels) AND r1/=r2)
    IMPLIES
  (agg_source(r1)=agg_target(r2) AND agg_target(r1)=agg_source(r2)))

% No name clashes are permitted in a class.

no_name_clashes: AXIOM
  (FORALL (c:CLASS):
    empty?(intersection(difference(all_names(class_features(c)),
                                  all_names(redefined_features(c))),
                        all_names((rename_class(inherited_features(c),
                                                renamings(c))))))))
```

An unintuitive axiom to formalize in PVS is that to ensure that labels appearing on a client-supplier relationship are distinct, i.e., the labels do not clash with names appearing in the feature list of a class. 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.

```
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 a feature of 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(class_features(c)))))
```

It was only when typechecking an early version of the PVS metamodel that we discovered the need for axiom `labels_unique_ax2`. Our original formulation considered only the case where the source of the client-supplier relationship was a class. The typechecker provided us with a proof obligation with the assumption

```
CLASS_pred(cs_source(cs))
```

This says that the source of any client-supplied relationship must be an instance of a *CLASS*, which of course cannot be assumed to be 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 properties that needed to be formalized.

Finally, we capture root properties for a model: a model can have at most one root class, and that class must be instantiated at least once.

```
unique_root_class: AXIOM
  (empty?({c:CLASS | member(c,abst) AND root(c)})  OR
  singleton?({c:CLASS | member(c,abst) AND root(c)}))

atleastone_inst_of_root: AXIOM
  (NOT empty?({o:OBJECT | member(o,abst) AND root(object_class(o)) }))
```

The complete metamodel written in PVS typechecks automatically without any user intervention.

4.5 Using the metamodel in conformance checking

The metamodel presented in the previous sections can be used to verify that BON models obey the well-formedness constraints, i.e., that a BON model is an instance of the metamodel. Verification follows the process of proving PVS `CONJECTURE`s, using the axioms of the metamodel.

There are several ways to use the metamodel theory to check that models satisfy or fail to satisfy the metamodel. The examples demonstrate several different approaches. The basic approach for conformance checking is as follows.

1. Formulate a BON model, following the method suggested in [14]. Currently, all features of the BON language are supported by the metamodel, though multiplicity of relationships must be expanded by the user before applying the metamodel (since multiplicity annotations are syntactic sugar, this is an acceptable procedure and can in fact be automated).

2. Express the model in PVS, as a conjecture. There are two ways to do this, phrasing the conjecture as a question about existence, or about non-existence, of a model. In formulating the PVS version of the model, the following guidelines can be followed.

   - BON classes are constants of type `CLASS`; objects are constants of `OBJECT` type. Clusters are constants of type `CLUSTER`.
   - Relationships are constants of an appropriate type, e.g., `AGG`, `INH`. Each relationship should be a member of its source abstraction.
   - The constraints on each abstraction can be defined in several ways: as boolean functions (whereupon the conjecture to be checked is the conjunction of the boolean functions); or, as a predicate consisting of the definitions of the following functions and constraints.
     – `static_rels` and `dynamic_rels`: the relationships that apply to each abstraction.
     – uniqueness constraints for each abstraction, i.e., that each abstraction differs from all others.
     – membership constraints, so that each abstraction is an element of `abst`, and each relationship is an element of `rel`.
     – names and contracts of features, if relevant.

3. Apply each axiom from the metamodel to the conjecture, and attempt to prove (or disprove) the conjecture. Many axioms will prove automatically, though some (e.g., the unique root conjecture, see below) will require user assistance.

Note that this process only checks the syntactic well-formedness of BON models; it guarantees nothing about the semantic well-formedness, e.g., whether behavioural subtyping between child and parent classes is obeyed, whether feature contracts are refined under redefinition, et cetera. A prototype tool, first proposed in [7], is under development that will allow PVS users to check such semantic properties using the PVS theorem prover.

We now illustrate the use of the metamodel in several short examples.

4.5.1 Bidirectional aggregations

According to the definition of BON, it is not possible for a class A to have an aggregation relationship directed to class B when B has an aggregation directed at class A as well. This is because aggregation represents the subobject relationship. We should be able to prove, using the metamodel theory presented in the previous section, that there is no BON model that possesses a bidirectional aggregation. This could be captured in the following theory (an informal explanation follows the theory).

```
  use_metamodel: THEORY
  BEGIN
    IMPORTING metamodel

    c1, c2: VAR CLASS
    a1, a2, y: VAR AGG

    no_bidirectional_aggregations: CONJECTURE
      (FORALL (c1,c2:CLASS):
          member(c1,abst) AND member(c2,abst) AND
          c1/=c2 IMPLIES (FORALL (a1,a2:AGG): a1/=a2 AND
            member(a1,rels) AND member(a2,rels) AND
            member(a1,static_rels(c1)) AND
            member(a2,static_rels(c2)) AND
            agg_source(a1)=c1 AND
            agg_source(a2)=c2)
          IMPLIES
          NOT (agg_target(a1)=c2 AND agg_target(a2)=c1)))
END use_metamodel
```

The conjecture is as follows. Suppose that there are two distinct classes, $c1$ and $c2$, and two distinct aggregation relationships $a1$ and $a2$. Suppose as well that $a1$ is sourced from class $c1$, and $a2$ is sourced from class $c2$. Then it cannot be the case that $a1$ targets $c2$ nor that $a2$ targets $c1$.

The conjecture can be proved using the PVS theorem prover as follows. First, introduce as a lemma the axiom `no_bidir_agg`, then instantiate the lemma with the two aggregation relationships $a1$ and $a2$. After flattening, one application of `(grind)` proves the conjecture automatically.

One weakness of the preceding example is that it seems difficult to automatically generate such a conjecture from a BON model. Ideally, we want to be able to draw models, automatically generate a PVS specification of the model, and then prove that the model satisfies (or fails to satisfy) the axioms that make up the metamodel. The following alternative formulation of the problem given above lends itself more to an automatic generation of a PVS version of a model. Suppose that we have a model with a bidirectional aggregation between classes $c1$ and $c2$. We formulate the model in PVS as a negated conjecture: effectively, we are postulating that no such model exists. We would like to prove this conjecture is true; alternatively, we could remove the negation and attempt to prove that such a model is valid, or try to discover a counterexample.

```
use_metamodel: THEORY
BEGIN
  IMPORTING metamodel

  c1, c2: VAR CLASS
  a1, a2, y: VAR AGG

  valid_model: CONJECTURE
    (NOT (EXISTS (c1,c2:CLASS):
       (member(c1,abst) AND member(c2,abst) AND c1/=c2) IMPLIES
       (EXISTS (a1,a2:AGG): (a1/=a2 AND
       member(a1,rels) AND member(a2,rels)) IMPLIES
       (member(a1,static_rels(c1)) AND member(a2,static_rels(c2)) AND
       cs_source(a1)=c1 AND cs_source(a2)=c2 AND
       cs_target(a1)=c2 AND cs_target(a2)=c1))))
END use_metamodel
```

The conjecture simply expresses the values of the static relationships in the models. To prove or disprove the validity of the conjecture, we must load each axiom of the metamodel and attempt to prove that the conjecture is valid. If we load the axiom that states that no model can possess a bidirectional aggregation (`no_bidir_agg`), then the conjecture proves using two skolemizations, one application of the lemma, an instantiation, and one application of `(grind)`.

4.5.2 Inheritance cycles

A more complex conjecture is that no valid BON model permits cycles in its inheritance graph. Consider Fig. 9, an *illegal* BON model; the model is not well-formed because of the cycle-introducing inheritance relationship from class A to class C (the labels $i1$ through $i3$ on the inheritance arrows are for reference only). If we can describe this model in PVS, then we should be able to prove, using the metamodel, that it is not well-formed.

The conjecture is captured in the following theory.

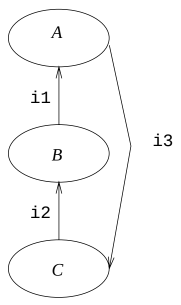

Figure 9: Inheritance graph with cycles

```
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. Suppose that there is a model consisting of the distinct classes a, b, and c with three distinct inheritance relationships $i1$, $i2$, and $i3$. Then the model cannot draw the inheritance relationships as shown in Fig. 9, that is, with $i1$ directed from b to a, with $i2$ directed from c to b, and with $i3$ directed from a to c.

It is somewhat more complicated to prove this conjecture than the one in the previous subsection. To prove the conjecture using PVS requires use of three lemmas, two of which define the inheritance closure of a model, with the third being `inh_wo_cycles`. Here is the proof, in PVS format.

```
(|use_metamodel2|
 (|inh_closure_test| "" (SKOLEM!)
  ((""" (FLATTEN)
    ((""" (SKOLEM!)
      ((""" (FLATTEN)
        ((""" (LEMMA "closure_ax1" ("r" "i1!1"))
          ((""" (LEMMA "closure_ax1" ("r" "i3!1"))
            ((""" (LEMMA "closure_ax1" ("r" "i2!1"))
              ((""" (PROP)
                ((""" (LEMMA "closure_ax2" ("r1" "i1!1" "r2" "i2!1"))
                  ((""" (PROP)
                    (("1" (SKOLEM!)
                      (("1" (PROP)
                        (("1" (LEMMA "inh_wo_cycles" ("r" "r!1"))
                          (("1" (PROP)
                            (("1" (INSTANTIATE -1 ("i3!1"))
                              (("1" (GRIND) NIL NIL)) NIL))
                            NIL))
                          NIL))
                        NIL))
                      NIL)
                     ("2" (GRIND) NIL NIL))
                    NIL))
                  NIL))
                NIL))
              NIL))
            NIL))
          NIL))
        NIL))
      NIL))
    NIL))
  NIL))
```

Effectively, after appropriately instantiating the axioms related to constructing the inheritance graph, the conjecture proves automatically using `(grind)`.

4.5.3 Unique roots

Ideally, we would like to use the metamodel to prove that our models satisfy the metamodel. A second example of how we might do this is the following. Suppose that we have two distinct classes A and B. Our informal reading of the metamodel indicates that not both A and B can be root classes (from which execution of the model will begin). We capture this in a theory, as follows.

```
roots: THEORY
BEGIN
IMPORTING metamodel

    a,b: CLASS

    one_root: CONJECTURE
      (member(a,abst) AND member(b,abst) AND a/=b)
        IMPLIES
      (NOT (root(a) AND root(b)))

END roots
```

We now want to check that our conjecture satisfies the metamodel. We scan the metamodel and notice the axiom `unique_root_class`.

```
    unique_root_class_ax: AXIOM
      (empty?({c:CLASS | member(c,abst) AND root(c)})  OR
       singleton?({c:CLASS | member(c,abst) AND root(c)}))
```

How do we use this axiom in proving the conjecture? We first introduce the axiom (use the `lemma` prover command), then expand the definition of `singleton?`. Suitable instantiations and skolemizations are carried out, and `grind` takes care of the rest.

4.5.4 Non-intersecting clusters

Clusters in a BON model are either nested (i.e., one cluster is contained within a second cluster), or they are disjoint and do not share contents. Consider the following BON model, where there are two clusters, X and Y. Cluster X contains class A. Cluster Y contains class B and also contains class A; this is illegal, and we should be able to catch this mistake with the metamodel, because it prohibits overlapping clusters. The model is depicted in Fig. 10.

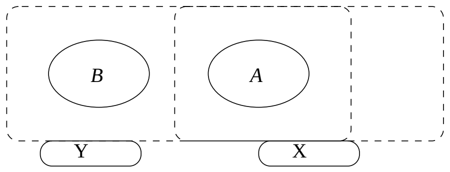

Figure 10: Overlapping cluster example

The following conjecture posits that no such model can exist. It would be proved by instantiating the disjoint cluster axiom, and by unfolding the definition of the function `static_contains`.

```
nonintersecting_clusters: THEORY
BEGIN
IMPORTING metamodel

  x,y:VAR CLUSTER
  a,b:VAR CLASS

  nonintersect: CONJECTURE
  (NOT (EXISTS (x,y:CLUSTER): x/=y AND member(x,abst) AND
   member(y,abst) AND NOT member(x,cluster_contents(y)) AND
   NOT member(y,cluster_contents(x)) AND
   (EXISTS (a,b:CLASS): member(a,abst) AND member(b,abst) AND
    a/=b AND member(a,cluster_contents(x)) AND
    member(b,cluster_contents(y)) AND member(a,cluster_contents(y)))))
END roots
```

4.5.5 Obeying export policies of classes

As another example of using the metamodel, we show how to check that a model correctly obeys the export policies of all classes in the model. Recall that in BON, each class has a collection of features. The features in a class interface are divided into sections, where each section is prefaced with a list of client classes that may use the features. A list may consist of the name $NONE$ (indicating that the features in the following section are private and inaccessible to no clients), the name ANY (indicating that the following features are accessible to all clients), or specific class names.

Consider the following BON model, depicted in Fig. 11. Suppose that h, w, and m are all queries, and that the following calls to these queries are made.

- Query m of class C is called in the invariant of class B.
- Queries h and w are called in the precondition of feature m of class C.

Note that m is a private feature of class C; thus the call to this feature in the invariant of B is illegal. Similarly, the call to w of class B is illegal in the precondition of m, because w is accessible only to the client A (as well as features of B). We would like to show that this model does not obey the constraints in the BON metamodel. We will show that, in particular, 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. First, we state a conjecture: that the model in Fig. 11 cannot exist.

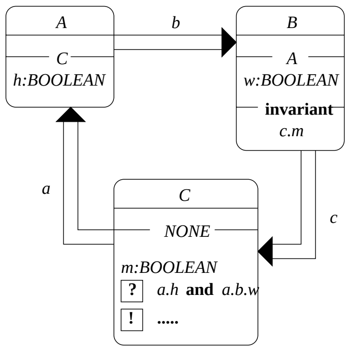

Figure 11: BON model with ill-formed assertions

```
info_hiding: THEORY
BEGIN
  IMPORTING metamodel
  a, b, c: VAR CLASS
  a1, b1, c1: VAR ASSOC
  h, w, m: VAR QUERY
  oa, ob, oc: VAR OBJECT
  ea, eb, ec: VAR ENTITY
  call1, call2, call_anon: VAR DIRECT_CALL
  call3: VAR CHAINED_CALL

  test_info_hiding: CONJECTURE
    (NOT (EXISTS (a, b, c: CLASS):
          EXISTS (h, w, m: QUERY):
          EXISTS (a1, b1, c1: ASSOC):
          EXISTS (oa, ob, oc: OBJECT):
          EXISTS (ea, eb, ec: ENTITY):
          EXISTS (call1, call2, call_anon: DIRECT_CALL):
          EXISTS (call3: CHAINED_CALL):
          member(a1, static_rels(c)) AND
          member(b1, static_rels(a)) AND
          member(c1, static_rels(b)) AND
          member(h, class_features(a)) AND
          member(w, class_features(b)) AND
          member(m, class_features(c)) AND
          member(c, accessors(h)) AND
          member(a, accessors(w)) AND
          empty?(accessors(m)) AND
          cs_source(a1) = c AND
          cs_source(b1) = a AND
          cs_source(c1) = b AND
          cs_target(a1) = a AND
          cs_target(b1) = b AND
          cs_target(c1) = c AND
          call_args(call1) = empty_seq AND
          call_args(call2) = empty_seq AND
          call_args(call3) = empty_seq AND
          call_entity(call1) = ea AND
          call_entity(call2) = ec AND
          call_entity(call_anon) = eb AND
          call_entity(call3) = ea AND
          call_feature(call1) = h AND
          call_feature(call2) = m AND
          call_feature(call_anon) = w AND
          rest(call3) = call_anon AND
          member(call1, calls_in_pre(m)) AND
          member(call3, calls_in_pre(m)) AND
          member(call_anon,calls_in_pre(m)) AND
          member(call2, calls_in_inv(b))))
 END info2
END info_hiding
```

Production of this conjecture from the model can be entirely mechanized. We have presented the conjecture in a form that could be automatically generated from the BON model in Fig. 11. Thus, there are details in the conjecture that are not needed in order to prove that the conjecture is true. In particular, we will need the *accessors* set for feature

m, as well as the definition of the calls used in the invariant of B.

To prove the conjecture, we first skolemize seven times, then flatten the assumptions. We introduce the lemma `valid_class_invariant`, and substitute class B and call $call3$ for the bound variables in this axiom. We then use `typepred` to bring the type assumptions on m into the proof. The definition of the function `class_isvalid` must be used (when applied to a direct call); this is accomplished by invoking the lemma defining the behaviour of the function. Then, one application of `grind` proves the conjecture automatically. Thus, the model is invalid according to the well-formedness constraints of the metamodel.

5 Discussion, Lessons Learned, and Related Work

By producing a metamodel for BON, we have taken a step towards placing the modeling 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 writers and language users to understand.

We constructed the BON metamodels in the following way.

1. We studied the context-free grammar for the textual dialect of BON [14]. This provided syntactic structuring information on abstractions (i.e., what abstractions could appear in a model, and how they could be related), features, and properties of abstractions. From the context-free grammar, we distilled and wrote down, in natural language, most of the informal constraints on using BON.

2. We next turned to the BON reference book [14], particularly the chapters on the BON assertion language, static diagrams, and dynamic diagrams. Here, we searched for inconsistencies, lack of clarity, and constraints that were missing from the context free grammar. These constraints were written down, in natural language, and added to the list constructed in Step 1. This provided us with approximately 25 well-formedness constraints on the use of BON.

3. Next, we took the constraints and expressed them in BON. The most difficult step, as always in object-oriented analysis, design, and implementation, was in identifying the underlying abstractions (finding the classes [3]). Our analysis suggested that a BON model consists of a set of abstractions (classes, objects, clusters); each abstraction would have a set of relationships that it is involved in (as the source of a relationship). The well-formedness constraints would have to operate on the set of abstractions. Abstractions and relationships would be modeled separately, as individual BON clusters.

   After deciding on abstractions to use in describing the metamodel, we filled in the details of each subsystem. The relationships cluster was straightforward, as we used classification and inheritance, as well as covariant redefinition, to capture many of the informal well-formedness constraints on the use of relationships. The abstractions cluster required us introduce both static and dynamic abstractions (in order to allow for the nesting that can occur with clusters) as well as classes for representing features and properties of features.

4. The final step was to express the metamodel in PVS. For the most part, we based the PVS formulation on the BON version of the metamodel, in order to better ensure consistency between versions, and for the sake of understandability: the BON version provides structuring information, and this proved very helpful in (a) choosing the right abstractions to use in PVS; and (b) structuring these abstractions, using PVS's subtype mechanisms and theory import and export.

We learned several things about PVS, BON, and metamodeling 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, more precise descriptions of constraints, and indications as to how PVS theories might be related via import and export relationships. The PVS version of the metamodel is not a direct transliteration of the BON metamodel. However, we did determine several helpful heuristics that can be used, in general, to help model object-oriented concepts in PVS:

- Class hierarchies can be modeled using PVS types and subtypes, e.g., as in the hierarchy of abstractions.

- Features (both queries and commands) can be described as functions that take an object as an argument, possibly with further arguments. Procedures, which modify objects, are functions that take an invoking object as an argument and return a new object. Thus, for example, the procedure call $o.p$, which changes the state of object o of class O, could be expressed in PVS as a function override, e.g.,

```
p: [ O -> O ]

p_ax: AXIOM
  (FORALL (o:O): p(o) = o WITH [ ... ]
```

This follows the approach to modeling OO systems using bunches in [6].

- Axioms can be used to define the behaviour (i.e., the pre- and postconditions) of the features, as well as more detailed type constraints.
- Conversions had to be generated in order to translate BON built-in primitive types, e.g., $INTEGER$, $STRING$, as PVS types. The PVS `CONVERSION` facility is ideal for handling this kind of problem.
- PVS requires types, functions, variables, and constants to be declared before they are used in formulae or axioms. Thus, it was not possible to completely transliterate the structure of the BON version of the metamodel into PVS.
- BON, by default, allows covariant redefinition of feature signatures when they are inherited. This was how we modeled the well-formedness constraints on relationships, e.g., that a client-supplier relationship must be between two static abstractions. This is expressed in PVS by using its subtyping mechanism.

We found PVS particularly helpful in debugging the metamodel. Our initial metamodel contained several type errors – specifically, 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 PVS type checker caught automatically. This information was used in updating and correcting the metamodel as it was being constructed.

5.1 Metamodeling with PVS

Our choice of PVS for writing the formal version of the BON metamodel was justified in Section 1.1. Primarily, we used PVS because it is an industrial-strength formal specification language, and because it is amenable to tool support, provided by the PVS system. For models as critical as the metamodel of a standardized modeling language, it is vital for correctness and consistency to be established; automated tools can help establish these properties.

We noted several significant advantages to using the PVS specification language in formally capturing the BON metamodel.

- PVS is a strongly typed specification language. The type system caught several flaws and omissions in early drafts of the metamodel, e.g., the constraints on client-supplier relationships that can be applied to both class and cluster targets. These type errors became apparent when automatic type checking of the PVS metamodel failed; counterexamples to typechecking were therefore produced. Such constraints would not have been discovered so easily with an untyped specification language, or especially a language that was unsupported by tools.
- The PVS built-in libraries, particularly for specifying and reasoning about sets, sequences, integers, etc., proved to be vital in adequately capturing the BON metamodel. Without a theory of sequences or of integers, parameters or covariant subtyping would not be expressible; without a theory of sets, classes and objects would not have been expressible. Thus, the PVS libraries allowed us to describe a richer theory for the BON metamodel, and let us more easily express the concepts and constraints that made up the metamodel.
- The semi-automatic type checking capabilities of PVS caught several minor mistakes in the theories. Once these errors had been corrected, type checking was performed automatically.
- The PVS specification language was expressive enough to describe the BON metamodel in a way that closely mimicked the structure of the precise description in BON itself. This was possible through use of PVS's subtyping mechanisms.

- PVS's theory structuring mechanisms (e.g., `IMPORTING`) allowed us to deal with the complexity of the metamodel description, and also let us more closely mimic the structure of the BON metamodel presented in BON. Structure is a key mechanism for dealing with complexity and improving the understandability of system descriptions.

5.2 Comparison with the UML metamodel

The UML metamodel was presented in [4]. It is written in a combination of UML itself (which is used to present the abstract syntax of UML), natural language, and the Object Constraint Language, which is a precise specification language that is used to specify well-formedness constraints. In this section, we briefly compare the BON metamodel with the UML metamodel. This is a useful task for several reasons:

- It provides us with a better understanding of the differences between the two modeling languages, particularly in terms of the fundamental description concepts that are used in producing models, and in terms of what issues will arise in producing tools to support the modeling languages.

- It provides a foundation for comparing the complexity and understandability of the two modeling languages, on more objective grounds than has been previously done, e.g., in [7].

The UML metamodel consists of three top-level packages, depicted in Fig. 12.

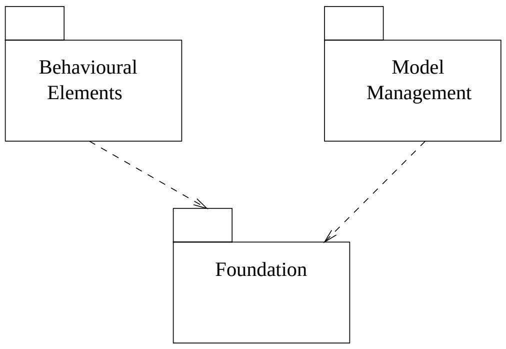

Figure 12: Top-level packages

The Behavioural Elements package describes concepts such as collaborations, use cases, and state machines – concepts that can be used to describe system behaviour. The Model Management package describes the concept of a model, as well as concepts such as presentation style. The presentation style focuses on how modeling concepts, such as class, packages, and the like, are to be presented on the printed page or display. The Foundation package describes the fundamental modeling elements that will appear in all models, such as classes, packages, interfaces, associations, and attributes.

There is no direct mapping between elements in the BON metamodel and the UML metamodel. Much of the Behavioural Elements package has no equivalent in BON, excepting some aspects of the Common Behaviour (e.g., objects) and Collaborations (e.g., messages). Similarly, many aspects of the Foundation package – excepting the UML Core – have no BON equivalents. The Model Management package in UML roughly corresponds to the class $MODEL$ in the BON metamodel, except that in the BON metamodel, no mention is made of presentation style. Thus, where the UML metamodel integrates both syntactic constraints pertaining to presentation as well as well-formedness, the BON metamodel focuses strictly on constraints related to well-formedness, and so the BON metamodel must be supplemented, in some way, by a description of what constitutes a valid presentation style.

The most significant commonality between the BON and UML metamodels is with the the UML Core package and the entire BON metamodel. The Core Package, which is enclosed in the Foundation package, consists of several sub-packages. The *Backbone* describes UML modeling concepts, such as modeling elements, operations, features, methods, attributes, and namespaces. It roughly corresponds to the Abstractions cluster in the BON metamodel (with some differences, which we describe shortly). The Backbone is described in Fig. 13.

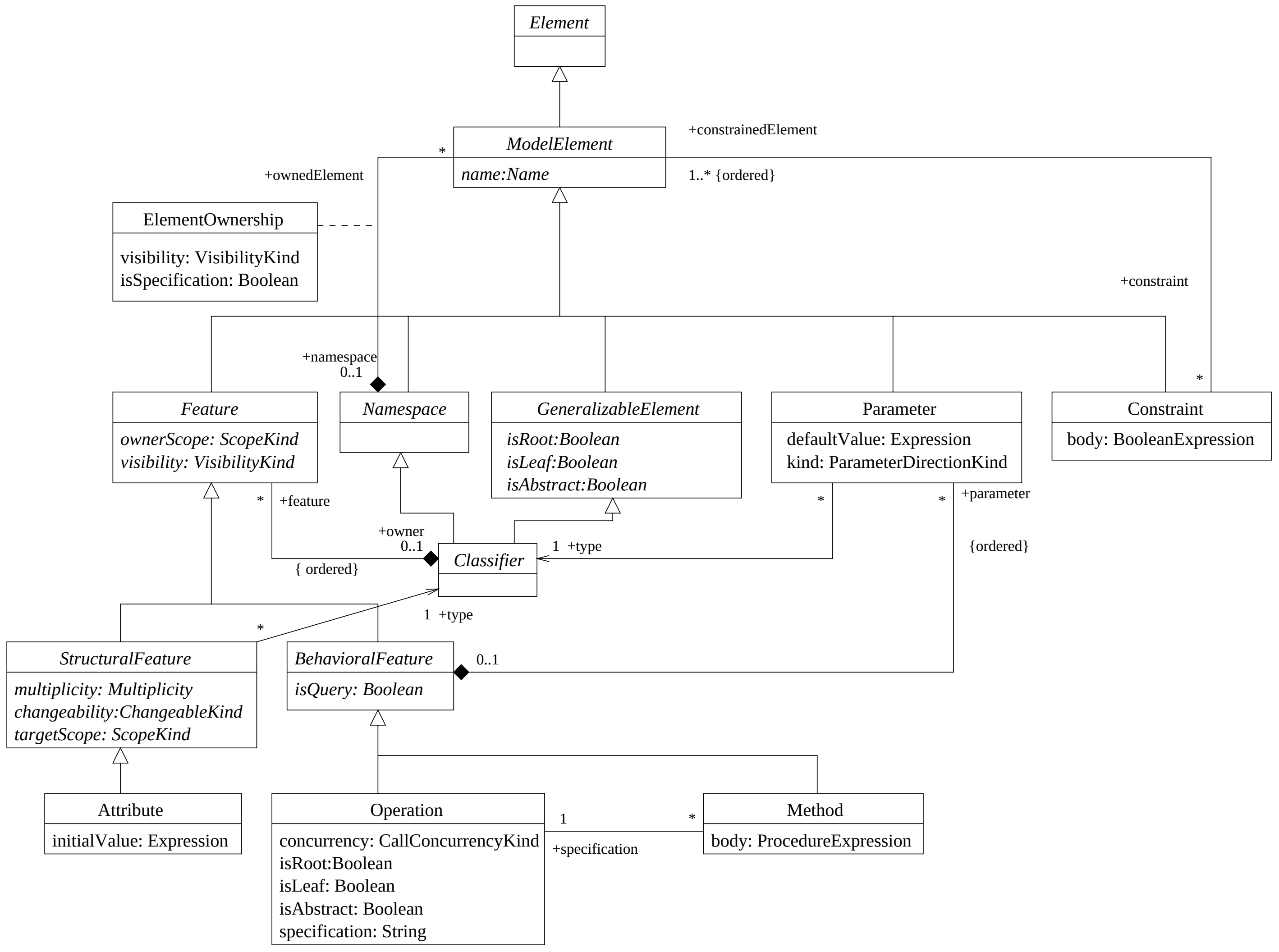

Figure 13: Core Package – Backbone

The key class in the Backbone package is *ModelElement*; it roughly corresponds to the BON class *ABSTRACTION*, but is actually more general. The ModelElement represents anything that can appear in a UML model, including Classifiers (e.g., a Class, an Interface), but also Relationships (e.g., Association, Generalization), and Attributes. Thus, in the UML metamodel, a model itself is a set of ModelElements, whereas in BON a model is a set of abstractions and a set of relationships, wherein the use of relationships is strictly constrained according to well-formedness constraints.

The BON metamodel uses the BON type system to constrain the use of relationships, e.g., by saying that associations (a static relationship) can be applied only to static abstractions like classes and clusters. Correspondingly, the UML metamodel must supply constraints to express that relationships like Association can be applied only to static modeling elements. We prefer the approach taken in the BON metamodel: constraints on the use of relationships are statically and automatically checkable (e.g., by a simple compiler) whereas OCL constraints are not, in general, checkable by static tool without user intervention.

In the UML metamodel, everything is a modeling element. Could this approach be applied to the BON metamodel, and if so, what would the advantages be? Suppose that in the BON metamodel, everything is an *ABSTRACTION*, and classification is used to separate static from dynamic abstractions. The class hierarchy might appear as shown in Fig. 14. Abstractions now include ‘classifiers’ (in the UML sense), ‘instances’, ‘flow’, and ‘relationships’. The hierarchy has been partially flattened, though some classification is still carried out to distinguish – in terms of their types – static abstractions from dynamic ones.

Each abstraction that represents a relationship (i.e., $INHERITANCE$, $AGGREGATION$, $ASSOCIATION$, etc.) has features representing the source and target of the relationship. The source and target features are queries of type $ABSTRACTION$.

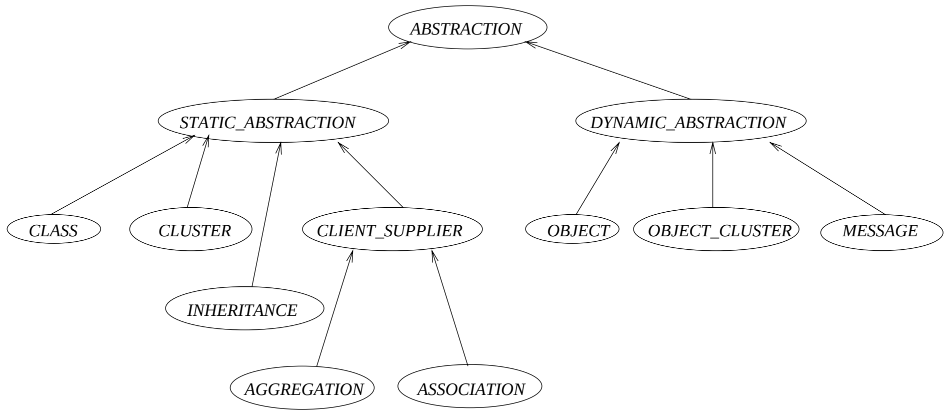

Figure 14: BON metamodel using UML approach

With the hierarchy presented in Fig. 14, a model can be described as just a set of abstractions. However, now, a relationship is an abstraction, and extra well-formedness constraints must be added in order to guarantee that, e.g., an inheritance relationship is applied only to static abstractions — classes and clusters — and not, e.g., to messages of aggregations. Such a constraint might be written, in the class invariant of $INHERITANCE$, as follows:

$$(source : CLASS \lor source : CLUSTER) \land (target : CLASS \lor target : CLUSTER)$$

This is low-quality object-oriented design: if a new static classifier needs to be added to the hierarchy, then all static relationship classes must be modified to take into account the new classifier (i.e., in the invariant of $INHERITANCE$, a new disjunct would have to be added to each conjunct). This violates the single-choice principle [3]: when a system has a list of options, one abstraction should know about the list. Thus it would be preferable to not put the well-formedness constraints in the separate relationship classes, but in the $MODEL$ class instead. In $MODEL$, the constraint would be written

$$\begin{aligned}\forall\, r \in abs \mid r : INHERITANCE \bullet (&r.source : CLASS \lor r.source : CLUSTER) \land \\ (&r.target : CLASS \lor r.target : CLUSTER)\end{aligned}$$

(and similarly for each static and dynamic relationships.) Constraints like these are more expensive to check, because of the quantifiers, than the simple type conditions that are present in the original BON metamodel. Thus, we prefer the original hierarchy and separation of a model into abstractions and relationships, since it is a higher-quality object-oriented design and because it is cheaper to typecheck. We suggest that the UML metamodel presents an awkward and incomplete classification of modeling concepts that does not lead to a quality object-oriented design. We also suggest that, in metamodeling (and modeling in general) it is preferable to use the type rules of the language instead of the constraint language, since the former is more likely to be automatically checkable than the latter.

A second question to ask is whether the BON metamodel would benefit from addition of an ultimate class parent, akin to ModelElement in the UML metamodel. In this sense, a $MODEL\_ELEMENT$ would have two children: a $RELATIONSHIP$ and an $ABSTRACTION$; the hierarchies of subclasses would otherwise remain as shown in Figs. 4 and 5. Then, a $MODEL$ would be very simple: it is a set of $MODEL_ELEMENT$s, and the BON metamodel would closely mimic the UML structure. We do not see any further advantages in doing this. Typically, in object-oriented design, parent classes such as $MODEL\_ELEMENT$ are added so as to (a) centralize behaviour that all child classes will need, or (b) to make use of polymorphism and dynamic binding. Neither (a) nor (b) are relevant in the construction of the BON metamodel. We do not observe any behaviour that is common to both abstractions and relationships in a BON model. And, polymorphism and dynamic binding is used when we want to treat a number of different types in the same way, through the same interface. It is unlikely that we will want to do this in the BON metamodel: we will instead want to establish specific constraints on specific abstractions and relationships; polymorphism will be of minimal use in doing so.

We now return to the UML metamodel, and turn our focus to relationships. The Backbone package says nothing about relationships in UML; these are described in the Relationships package, shown in Fig. 15. This UML package roughly corresponds in intent to the BON Relationships cluster.

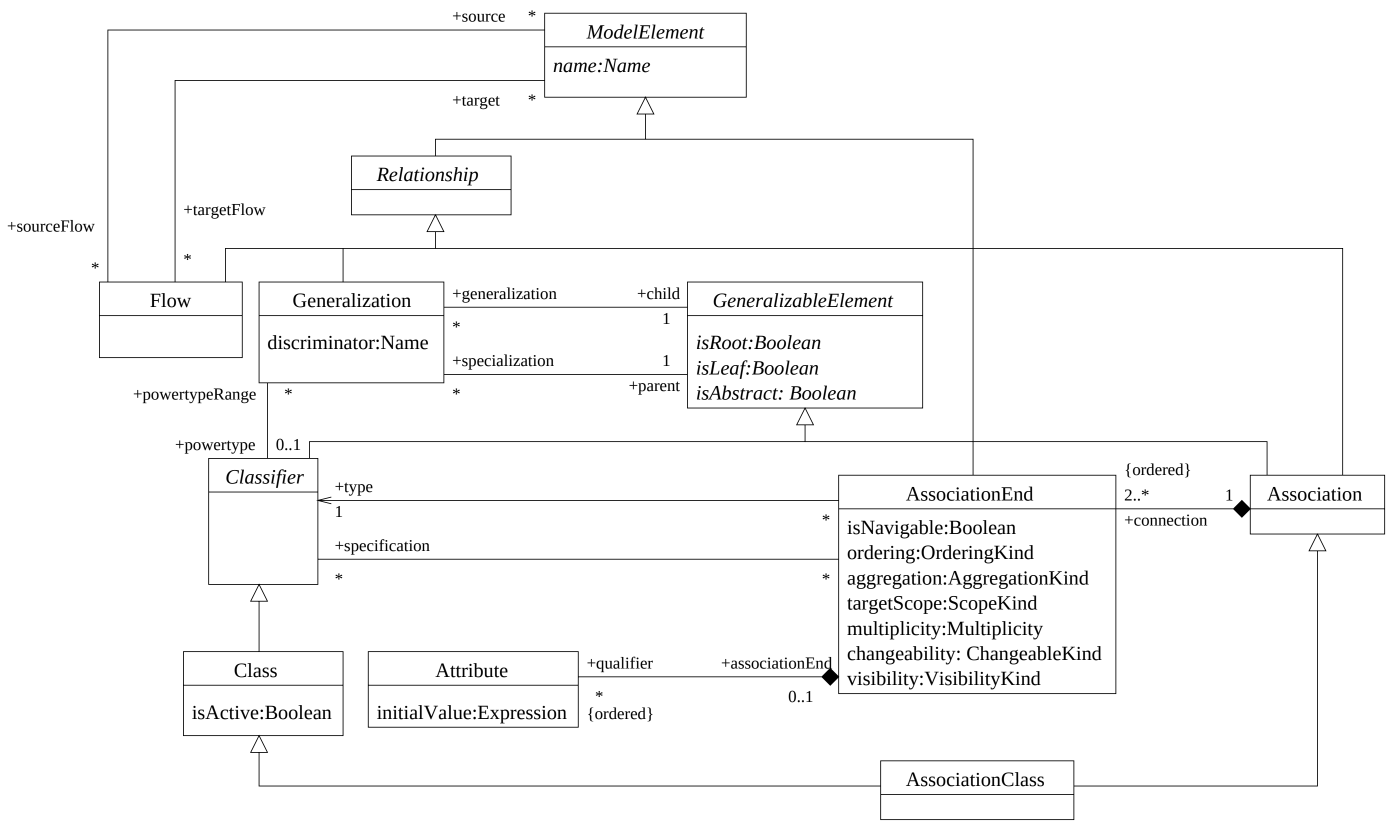

Figure 15: Core Package – Relationships

The three main relationships in UML are Flow, Generalization, and Association. These are approximate equivalents to the BON relationships $MESSAGE$, $INHERITANCE$ and $CLIENT\_SUPPLIER$. There is no BON equivalent to the notion of an AssociationClass in UML. This UML class is used to describe properties on relationships. In BON, relationship properties are expressed as constraints, in class invariants.

The other aspect of the Core Package that is comparable to BON is the Classifiers package, depicted in Fig. 16. This package shows several fundamental UML concepts, including the notions of Class, Interface, and DataType.

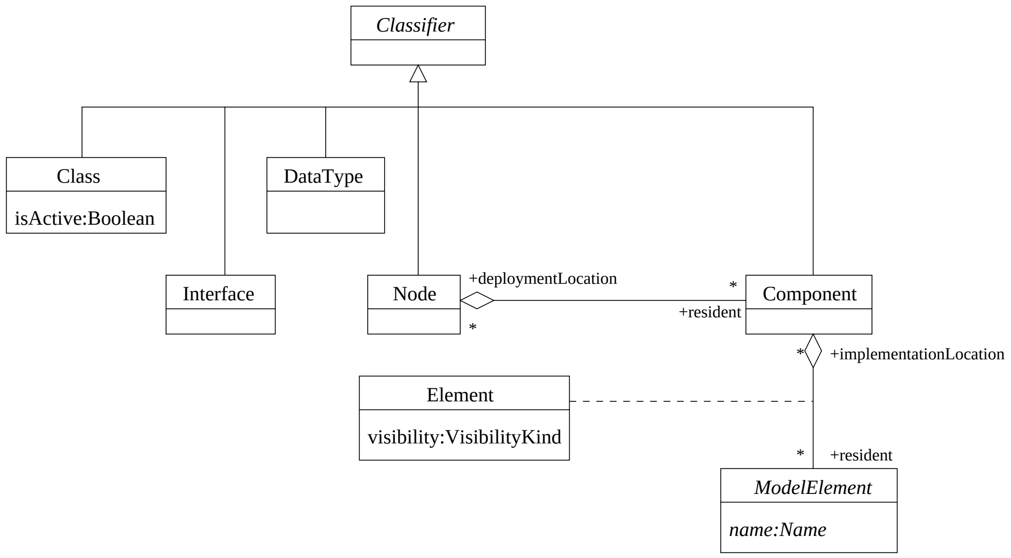

Figure 16: Core Package – Classifiers

A simpler picture is presented in the BON metamodel: the only classifier that is available is the $CLASS$: the notions of DataType and Interface are subsumed by $CLASS$, while abstractions like $OBJECT$ and $CLUSTER$ are not themselves classifiers. Specifically, the UML concept of an Interface is expressed in BON by tagging an instance of a class with the property $deferred$ (see Fig. 5). The BON metamodel does not distinguish primitive datatypes from classes, because this is an implementation-level issue: the metamodel should distinguish valid models from invalid models, and should not make mention of implementation hints or issues. It should be possible to provide hints to implementors (e.g., "the $INTEGER$ class should be implemented as a primitive type for the sake of efficiency"), but this information does not need to be a well-formedness constraint of the metamodel.

Several minor points should be mentioned regarding the BON metamodel and UML metamodel.

- BON does possess a select, fixed set of stereotypes [7], e.g., $external$, $interfaced$, $persistent$, et cetera. These can only be applied to classes [14], and thus are metamodeled as boolean features of class $CLASS$. There is no corresponding class Stereotype, as in the UML metamodel, because user-defined stereotypes cannot be constructed in BON.
- The UML metamodel possesses the notion of a Namespace (see Fig. 13), which is used to help deal with name clashes that can arise when importing packages, or when generalizing multiple packages or classifiers. No such abstraction is needed or present in BON. In BON, only classes can introduce feature names, and classes can also rename features as they are inherited. Thus, classes possess the means to eliminate name clashes, and no extra metamodeling abstraction is required to solve this problem.
- The UML metamodel distinguishes Attributes and Queries, whereas BON has only queries (attributes should be thought of as queries that access some hidden physical representation). By eliminating the distinction between queries and attributes, a metamodel can be simplified; removing the distinction in the UML metamodel could lead to the removal of the StructuralFeature class in Fig. 13.
- The UML metamodel distributes graphical constraints and textual constraints on metamodeling concepts over packages. For example, the concept of a ModelElement is constrained in the Backbone package as well as the Relationships package. To understand the concept of a ModelElement, a reader must understand information that is distributed over several packages.

We return, briefly, to the issue of presentation style. As mentioned earlier, the UML metamodel describes the presentation style of modeling elements, while the BON metamodel does not. How could we, if desired, add such information to the BON metamodel? One approach would be to use the Bridge design pattern. Each element that may appear in a BON model, e.g., a $CLASS$, an $INHERITANCE$ relationship, etc., is associated with a corresponding $PRESENTATION$ class, thus forming a parallel hierarchy to that for abstractions and relationships. This design pattern allows us to decouple the notion of presentation style from notions of modeling elements, thus improving maintainability: changing the presentation style does not change the modeling element, and vice versa – the two concepts can vary independently. This is useful if it is likely that the presentation style of one or several modeling elements may change frequently.

5.3 Contrasting formal specifications of metamodels

A formal specification of a subset of the UML metamodel was presented in [13]. The metamodel was specified in Alloy [2], a formal specification language that was designed to be amenable to automatic analysis, simulation, and consistency checking.

It is worth briefly contrasting the use of PVS for metamodeling with the use of Alloy for the same task. Alloy is a simpler specification language than PVS. It treats scalars as singleton sets. Types are implicit, and are associated with domains – declared sets that are not subsets of any other set. Thus, classes correspond to sets that are subsets of domains. Alloy specifications are structured into paragraphs. The domain paragraph declares sets that are used in the specification. The state paragraph declares additional sets with elements drawn from the domains. Following these paragraphs comes state invariants.

There are a number of differences between PVS and Alloy that are relevant to metamodeling:

- PVS is a richer specification language than Alloy. It possesses interpreted types (e.g., `int`, `seq`) whereas all types in Alloy are implicit. PVS lets us more conveniently specify such things as the number of arguments in a feature call corresponds to the expected number of parameters. Alloy does not yet support sequences (though it should be possible to add them atop indexed relations).

- PVS is not an object modeling language; it is a general specification language. Alloy is intended directly for object modeling (it possesses a graphical sublanguage akin to UML). Interestingly, though, Alloy provides only minimal structuring facilities: Alloy models are flat, in that invariants are global, and are applied to everything declared in the domain and state paragraphs. This seems to be against the tenets of object-oriented modeling: invariants and constraints should be kept with the modeling elements to which they apply, for maintainability and reuse.

- PVS provides generic types, which are useful in specifying and reasoning about features, particularly their parameters and arguments. Alloy has no notion of generic. The BON language also supports generics.

- PVS provides substantial tool support – its theorem prover is industrial strength and proven to be applicable to large-scale applications. The Alloy tool, Alcoa, is under development. The PVS tool suite was essential in debugging and verifying the accuracy of the metamodel.

- PVS uses standard predicate logic and set theory in specifying a metamodel: a BON model is just a set of abstractions and relationships. By comparison, Alloy – while founded on set theory – has a relational flavour to it. The UML metamodel presented in [13] uses relational operators (e.g., transitive closure) to specify constraints. In our opinion, using predicate logic and set theory to build up theories, and reusing and importing these theories, is simpler, easier to read, and more likely to be understandable by novice formal specification language users.

In the UML 1.3 document [4], the UML metamodel is captured in a combination of a core subset of UML, OCL expressions, and natural language. The work of the USE project [10] has in part developed a toolset for validating UML models and OCL constraints, based on simulating UML models and interpreting OCL constraints. The tool has been applied to part of the UML metamodel. Their tool allows conformance checking, using a similar procedure to what has been documented in this report. In conformance checking, a UML model (typically described in XMI) is imported, and the model is traversed and elements are instantiated as objects of the UML metamodel. Then, all constraints are checked on the resulting snapshot. Effectively, this is conformance checking via simulation, whereas with our approach, conformance checking resolves to proving a conjecture in PVS. The USE approach supports a subset of UML/OCL.

6 Conclusions

We have presented two metamodels for the BON modeling language. The first metamodel, written in BON itself, aimed at a precise description of the well-formedness constraints of the language. The constraints were expressed both graphically – in terms of BON graphical syntax – and textually, in terms of assertions that are applied to specific classes. This metamodel was constructed with the aim of understanding: explaining to users and tool developers the constraints on writing and applying BON. The second metamodel, written in the PVS specification language, is formal – in that the specification language has a formal syntax and semantics – and is intended to be used for analysis, in particular by modeling language developers, who are responsible for validating the consistency of their metamodel. The PVS tool can be used to assist this process.

There are, in our opinion, two vital questions that must be answered by any metamodel.

1. *What can legally appear in a model?* According to the BON metamodel, there are only abstractions and relationships. In UML, anything that is a ModelElement (e.g., Classifiers, Relationships) can appear in a model. It should be as straightforward as is possible to determine what can appear in a model, by examining the metamodel.

2. *How can elements in a model be legally connected?*

One question that arose when we were constructing the precise version of the metamodel (written in BON) was: to what extent should the assertion language be used? Certainly, all well-formedness constraints could be expressed in the assertion language. However, BON also possesses a type system (based on classes) and it was possible to push off certain well-formedness constraints (e.g., that static relationships are only between static abstractions) to the type system rather than the assertion language. A similar question arises when studying the UML metamodel: to what extent should OCL be used? Our conclusion is that as many constraints as possible should be expressed in the type system of the modeling language, because it is more likely to be automatable than the assertion language.

In metamodeling with BON, we identified a number of characteristics that we believe all quality metamodels should possess.

- They should be *understandable*. The intent with a metamodel is that it be read by language users and by tool writers. Thus, understandability of the metamodel is utterly critical, and, in our opinion, supersedes all other characteristics.

- They should be *precise*, so that readers unambiguously understand the metamodel, and they should be *formal* so that it can be guaranteed, by the metamodelers, that they are consistent.

- They should be *well-structured* and *documented*, the latter preferably being in natural language.

Understandability is clearly the critical element in a metamodel. A metamodel, more so than other models, is intended to be read by language users, developers, and tool writers. Thus, serious effort must be expended on structuring, presenting, and documenting the metamodel so that it is understandable by as many readers as possible.

References

[1] E.C.R. Hehner. *A Practical Theory of Programming*, Springer-Verlag, 1993.

[2] D. Jackson. Alloy: a Lightweight Object Modelling Notation. Technical Report, MIT Laboratory for Computer Science, 1999.

[3] B. Meyer. *Object-Oriented Software Construction*, Prentice-Hall, 1997.

[4] Object Modelling Group. *OMG Unified Modelling Language Specification*, OMG, June 1999.

[5] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. The PVS Language Reference Version 2.3, SRI International Technical Report, September 1999.

[6] R. Paige and E.C.R. Hehner. Bunches for Object-Oriented, Real-Time, and Concurrent Specification. In *Proc. World Congress on Formal Methods (FM'99): Volume 1*, LNCS 1708, Springer-Verlag, 1999.

[7] R. Paige and J. Ostroff. A Comparison of BON and UML. In *Proc. Second International Conference on the Unified Modelling Language (UML'99)*, LNCS 1723, Springer-Verlag, 1999.

[8] R. Paige and J. Ostroff. An Object-Oriented Refinement Calculus, submitted December 1999 (Revised August 2000).

[9] R. Paige, J. Ostroff, and P. Brooke. Principles for Modeling Language Design. *Information and Software Technology* 42(10):665-675, June 2000.

[10] M. Richters and M. Gogolla. Validating UML Models and OCL Constraints. In *Proc. UML 2000*, LNCS 1939, Springer-Verlag, October 2000.

[11] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[12] J.M. Spivey. *The Z Reference Manual*, Second Edition, Prentice-Hall, 1992.

[13] M. Vaziri and D. Jackson. Some Shortcomings of OCL, the Object Constraint Language of UML. Technical Report, MIT Laboratory for Computer Science, December 1999.

[14] K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Development*, Prentice-Hall, 1995.