Theorem Proving Support for View Consistency Checking

Richard F. Paige* — **Jonathan S. Ostroff**** — **Phillip J. Brooke*****

* *Department of Computer Science, University of York,*
Heslington, York YO10 5DD, United Kingdom. paige@cs.york.ac.uk

** *Department of Computer Science, York University,*
4700 Keele Street, Toronto, Ontario M3J 1P3, Canada. jonathan@cs.yorku.ca

*** *School of Computing, University of Plymouth,*
Drake Circus, Plymouth, Devon, PL4 8AA, United Kingdom. philb@soc.plym.ac.uk

ABSTRACT. *A formal, mechanically checked specification of the consistency constraints between two views of object-oriented systems are presented. The views, described in the BON modelling language, capture the static architecture of systems via contract-annotated class diagrams, and the dynamic view provided by collaboration diagrams. The constraints are specified as an extension of the BON metamodel, and are implemented in PVS. They ensure that the sequence of messages appearing in the dynamic view is legal, given the pre- and postconditions of methods appearing in the static view. An example of how the PVS theorem prover might be used to verify view consistency is described.*

RÉSUMÉ. *Cet article présente une approche formelle et automatisée pour la vérification de la cohérence des contraintes entre deux vues d'un système orientée objets. Les vues, décrites dans le langage de modélisation BON, capture l'architecture statique du système grâce à des diagrammes de classes annotés par des contrats, la vue dynamique est réalisée par des diagrammes de collaboration. Les contraintes sont spécifiées comme une extension du méta-modèle de BON et sont implémentées en PVS. Elles assurent que les séquences de messages qui apparaissent dans la vue dynamique sont légales compte-tenu des pré-post conditions de la vue statique. Un exemple d'utilisation du prouveur PVS pour démontrer la cohérence des vues est décrit.*

KEYWORDS: *view consistency, metamodelling, theorem proving, BON, PVS*

MOTS-CLÉS : *cohérence des vues, méta-modélisation, démonstration, contrat, BON, PVS*

1. Introduction

Consistency checking of the documents that are produced during software development is a difficult task. Consistency checking has been discussed in the context of work on on multi-viewpoint specification [FIN 94] and combining specifications [ZAV 93] written in different formal and semiformal languages. It is especially challenging and relevant when using the modelling language UML [BOO 99], where five different and potentially conflicting views of a software system of interest can be independently constructed. The intent of using multiple, disparate views is to describe different aspects of a system in the most appropriate way. The different descriptions must at some point be combined to form a consistent description of the system that can thereafter be used to produce executable program code. The process of combining the descriptions should identify inconsistencies that need to be resolved by the developers before executable code can be produced.

In general, consistency checking of software development deliverables involves the use of constraints, algorithms, and tools to check that information described in one deliverable (e.g., source code, a UML model) is not contradicted by information described in another deliverable. In a setting where formal specifications are available, this can be reduced to the problem of checking that a conjunction of predicates – each a formal specification of a deliverable – is satisfiable. In general, complete formal specifications of models are usually unavailable, and thus the problem of consistency checking is made more complex and challenging.

The problem of consistency checking is not unique to UML; any modelling or specification language that supports multiple views must be supplemented with techniques for detecting and managing inconsistency. In the case of UML and similar languages, specifying a complete set of consistency constraints is challenging. It is thus useful to be able to define the known consistency constraints in an extensible way. Some constraints, e.g., "modelling elements have unique names", "classes do not generalize themselves", are specified in the UML *metamodel* [OMG 00] and many UML-compliant CASE tools implement some of them, e.g., by restricting the user interface or by requiring views to be constructed in a specific order. However, some of the complex constraints, such as those involving the use of *contracts* [MEY 92] of methods, are not implemented in any tool, and thus developers must rely on their own expertise to identify and resolve inconsistencies. The goal of this paper is to provide guidance, infrastructure, and tool-supported techniques for users of object-oriented modelling languages, and particularly designers of tools that support OO modelling languages, in checking view consistency.

The specific aim of this paper is to formally model, in a machine-checkable language, consistency constraints between two views of an object-oriented system: specifically, the static view presented by class diagrams, and the dynamic view presented by collaboration diagrams. Formulating these constraints is made more challenging by the use of *contracts* – discussed in the sequel – in the static view. The two views will be specified in the BON modelling language [WAL 95]. We make use of BON

since it is designed to use contracts in formally specifying systems, and contracts pose significant challenges with checking view consistency, which we aim to address.

BON (and UML, and similar modelling language) supports two fundamental models: *class diagrams* and *collaboration diagrams*. These diagrams present, respectively, a static structural view of a system, and a dynamic view of a system, the latter captured by describing objects and the messages passed between them. So fundamental are these two types of models in OO computing that they are supported in many object-oriented (OO) modelling languages.

The consistency constraints between views are specified as an extension of the metamodel of BON presented first in [PAI 01b], and implemented in the BON CASE tool [PAI 01a]. Since the metamodel has been implemented in a CASE tool, it cannot be arbitrarily restructured to include additional view consistency constraints without requiring substantial changes to the tool as a whole. Thus, the metamodel will have to be extended carefully, making use of OO extension facilities such as generalization. The constraints will also be specified in the PVS specification language [OWR 01], so that theorem proving technology can be exploited both in checking the consistency of views, and in validating the specifications of the constraints. This is particularly useful given that a formal translation from BON to PVS, and a proof of its correctness, does not yet exist. Additional benefits accrued from using PVS for specifying the constraints will also be discussed. The intent is to use these specifications, and their implementations, in the construction of a CASE tool for BON that supports not only consistency checking but also consistent views by construction, i.e., via reverse engineering.

As stated, the BON language shall be used for describing the two different views. However, the rules that are presented in this paper are not BON dependent; they can be applied equally well to profiles of UML and other modelling languages that support these two views and which support contracts.

2. An Overview of BON

BON [WAL 95] is an OO method possessing a recommended process as well as a graphical language for specifying object-oriented systems. The language provides mechanisms for specifying classes and objects, their relationships, and assertions (written in first-order predicate logic) for specifying the behaviour of routines and invariants of classes.

The fundamental construct in BON is the class. A class has a name, an optional class invariant, and zero or more features. A feature may be an *attribute*, a *query* – which returns a value and does not change the system state – or a *command*, which changes system state but returns nothing. Fig. 1 contains an example of a BON model for the interface of a class *CITIZEN*. A graphical notation is also available for writing class interfaces; it is detailed in [PAI 01b].

class *CITIZEN* **inherit** *PERSON*
feature $\{ANY\}$
    name, *sex* : *STRING*
    age : *INTEGER*
    spouse : *CITIZEN*
    children, *parents* : *SET*[*CITIZEN*]
    single : *BOOLEAN*
        ensure $Result = (spouse = Void)$
    divorce **is**
        modifies *spouse*
        require $\neg$ *single*
        ensure $single \wedge (\textbf{old}\, spouse).single$

invariant
    `single_or_married`: $single \vee spouse.spouse = Current;$
    `number_of_parents`: $parents.count \leq 2;$
    `symmetry`: $\forall c \in children \bullet \exists p \in c.parents \bullet p = Current$
end

Figure 1. *Class CITIZEN textual interface*

BON emphasizes the use of contracts when writing class interfaces. Each routine in a class may have a precondition (**require**) and postcondition (**ensure**), e.g., as in *divorce*, above. The precondition specifies constraints on when the routine can be called; the postcondition specifies constraints on the behaviour of a routine. The **modifies** clause specifies the variables that may be changed by the routine. Classes themselves may be documented via *invariants* which specify conditions that client-accessible routine must maintain. Class invariants describe properties that all instances of a class must obey at well-defined "stable" points of execution.

BON static diagrams consist of one or more classes, drawn as ellipses, organized in *clusters* (drawn as dashed rounded rectangles that may encapsulate classes and other clusters). Classes and clusters interact via two general kinds of relationships.

– **Inheritance:** Inheritance defines a subtyping relationship between a child and one or more parents.

– **Client-supplier:** there are two client-supplier relationships, association and aggregation. Both relationships are directed from a *client* to a *supplier*. Association depicts reference relationships, while aggregation depicts subobject (or part-of) relationships; they are thus directly mapped to the implementation concept of an attribute. Client-supplier relationships can be drawn between classes and clusters; recursive rules are given in [WAL 95] to explain the meaning of cluster relationships.

An example of a simple class diagram, demonstrating much of the static diagram notation, is in Figure 2.

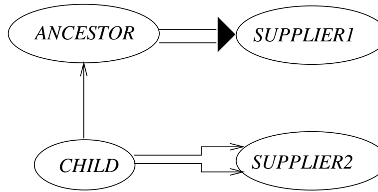

Figure 2. *Basic elements of BON class diagram notation*

BON also provides notation for collaboration diagrams, showing the communication between objects at run-time. Fig. 3 shows an example. Numbers that annotate messages are cross-referenced to a scenario box, detailing the purpose of the message. Messages in dynamic diagrams correspond to feature calls and are thus always potential, depending, for example, on whether the precondition of a feature is enabled or not.

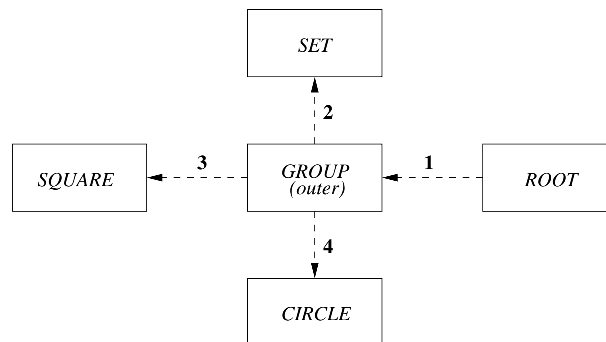

Figure 3. *BON collaboration diagram (without a scenario box)*

UML provides five views of a system [OMG 00]. Each view is considered to be a separate model of the system depicted with a separate diagram. This *multiple model* approach can of course lead to inconsistency: information presented in one diagram may contradict information in another diagram.

By contrast, BON supports the *single model principle*. A full discussion of the principle is beyond the scope of this paper; the paper [PAI 02b] explains it in detail. The basic idea is as follows. As with multiple model approach, languages and methods that support the single model principle also support multiple views of the system. However, the principle aims to ensure consistency of views either by automatic construction or by rigorous analysis. A language that follows the single model principle has several characteristics:

– it is *seamless*, in that its modelling abstractions can be used throughout the development process;

– it is *reversible*: models can be reverse engineered automatically from code;

– it is *wide-spectrum*, applicable to requirements analysis, design, and implementation;

– it provides *conceptual integrity*: it uses a small number of powerful descriptions that work together to help describe the software product; and it provides one good way to describe every construct of interest.

– it provides mechanisms for automatically establishing or checking the consistency of views.

The desirability of having a modelling language that satisfies these characteristics when building high-integrity systems has been argued in detail in [PAI 01c]. Essentially, the argument is that following the principle provides greater assurance and support for building products and having the document deliverables remain consistent. The single model principle also affects the structure of metamodels and how such metamodels can be extended to support view consistency checking.

3. Checking Collaboration Diagrams Against Class Diagrams

In this section, constraints are specified that consistent class diagrams and collaboration diagrams that model a system must obey. It is important to clarify that the constraints are *specifications*; they are not meant to be implemented directly in a CASE tool.

The goal is to check the consistency of one or more BON collaboration diagrams against one or more BON class diagrams, and if the diagrams are inconsistent, to report where the inconsistencies arise. Inconsistencies can arise due to object declaration (e.g., an object is unassociated with any class in a model), or routine invocation (e.g., a routine is being called by a client that does not have the ability to do so, based either on information hiding rules, or on preconditions). It is critical to observe that class diagrams contain only contracts, and not implementations, of routines. Further, the BON assertion language, based on first-order predicate logic, contains constructs that are not executable (e.g., quantifiers over unbounded domains). Thus, in general, consistency checking will not be possible by direct simulation of the collaboration diagram, and will likely require user intervention.

The consistency constraints are *meta-level* constraints; that is, they will be specified at the meta-level of BON, and apply to all models that can be described using BON. Thus, the constraints will be specified as an extension of the BON metamodel, which was first described in [PAI 01b].

There are several main steps to establishing the consistency of class diagrams and collaboration diagrams.

1) **Syntactic Correctness:** ensure that the two diagrams are syntactically correct; there is a BON CASE tool and a context-free grammar that will do this for us [PAI 01a].

2) **Contextual Correctness:** ensure that the diagrams are semantically correct in the sense that they obey typing and scoping rules (e.g., all classes arising in an interface appear in a class diagram). We call these *weak metamodel constraints.* Such constraints are defined as being straightforward to implement in a CASE tool via either user interface mechanisms or by automated analysis.

3) **Semantic Correctness:** check that the sequence of messages being fired in the collaboration diagram is allowable given the pre/postconditions of the routines in the class diagram. These are *strong metamodel constraints* that will likely require use of external tools – particularly, theorem proving technology – and particularly user intervention to validate.

We consider these steps in order, skipping 1) since the BON CASE tool is described elsewhere [PAI 01a]. The remaining constraints will be specified as an extension of the metamodel of BON first presented in [PAI 01b]. It is important to extend the metamodel in such a way so that the additional constraints can easily be introduced in to the existing CASE tool, without requiring substantial changes to the existing system. Some of these constraints, specifically those in 2), are easily implemented in a CASE tool. Other constraints are more complex, and thus will be expressed using the PVS language, so that thereafter the PVS system can be used to check the constraints, either interactively or in batch mode.

In extending the metamodel, we will make use of standard OO practices and design techniques. The BON metamodel has been specified as an object-oriented model, and as will be described, extending it will require extending classes in this model. The *open-closed* principle [MEY 97] is at the foundation of OO model extension, particularly in BON, but also in other modelling languages. In other words, extension of the metamodel will be by inheriting from existing classes and adding new constraints, routines, and attributes. This should be contrasted with the promising template-based metamodel extension mechanisms supported by MML and MMT and used in the proposal for UML 2.0 [CLA 02]. In this approach, extensions to a metamodel can be made by adding new templates, or by changing template instantiations and substitutions. The MML approach appears to be easier to use for metamodel extension, and can be helpful in avoiding problems with fragile base classes. On the other hand, by using standard OO extension mechanisms, we need no special tool support in order to implement the extensions, as standard OO programming constructs will suffice. Further experiments need to be carried out in order to properly contrast the MML approach with standard OO extension approaches.

Before specifying the rules in 2) and 3), we briefly recount the key parts of the BON metamodel from [PAI 01b]. The BON metamodel consists of two clusters and one root class, *MODEL*; every BON model is an instantiation of *MODEL.* The general outline of the metamodel is in Fig. 4.

Well-formedness constraints in the metamodel are specified as clauses in the invariant of *MODEL*, or in classes in the clusters *ABSTRACTIONS* or *RELATIONSHIPS*. New constraints for the rules in 2) and 3) will also be integrated into the metamodel

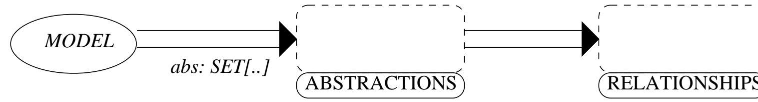

Figure 4. *BON metamodel, abstract view*

as invariant clauses (as we discuss shortly) of extensions of classes in the original metamodel specification.

We now present the consistency constraints, and in doing so apply the textual dialect of BON. These specifications will be used in formulating machine-checkable PVS specifications which can then be applied in automatically proving that a collaboration diagram is consistent with a class diagram.

First, we recap the concept of a routine of a class from [PAI 01b]. A routine has a name, a possibly empty sequence of parameters, a set of accessors, a pre- and post-condition, and a specification, which corresponds to the semantics of the routine. (In [PAI 01b], a routine is specialized into queries, which return values, and commands, which change the state of the system; this is a level of complexity that we can ignore in this paper.) Here is the interface of *ROUTINE*.

class *ROUTINE* **feature**
  name : *STRING*
  parameters : $SEQUENCE[PARAMETER]$
  $pre, post, spec$: *BOOLEAN*
  accessors : $SET[CLASS]$
invariant
  $spec = ((\textbf{old}\ pre) \rightarrow post \wedge t \geq \textbf{old}\ t \wedge t \neq \infty)$
end

spec is the semantics of a routine; *t* is a global clock. According to the invariant of *ROUTINE*, the specification of the routine is satisfied if any implementation starts in a state satisfying the precondition and terminates in finite time in a state satisfying the postcondition. The semantics of specifications is from [PAI 99], where a calculus for refining BON specifications to Eiffel programs is presented.

Part of the PVS formulation of the BON class *ROUTINE* is given below; missing details may be found in [PAI 01b]. A new non-empty type is introduced, and features of the BON class are transformed to PVS functions. The precondition and postcondition are formalized as functions mapping a routine and state (the latter represented as one or two sets of entities, respectively) to a boolean value; the state is needed for composing specifications sequentially.

```
FEATURE: TYPE+
ATTRIBUTE, ROUTINE: TYPE+ FROM FEATURE

routine_name: [ ROUTINE -> string ]
feature_pre:  [ ROUTINE, set[ENTITY] -> bool ]
feature_post: [ ROUTINE, set[ENTITY], set[ENTITY] -> bool]
```

Expressing a routine's specification in PVS is more complicated. The complication does not arise in expressing a specification directly, but in *combining* specifications: PVS requires explicit specification of a function's domain (possibly using an uninterpreted type) in order to support type checking. In the metamodel, a function is a specification with its domain being the state of the specification; thus, a specification's state must be formally specified in the PVS version of the metamodel. In the previous version of the metamodel in [PAI 01b], a routine's specification was not needed, and thus this problem did not arise.

The formulation of specifications is aimed at being able to (sequentially) compose them. The formalization of specifications of a routine requires a new type, `SPECTYPE`, which is a record containing the initial and final state variables of a specification, along with the value of the specification; initial and final state are sets of entities. The functions `oldstate` and `newstate` produce the entities associated with a routine (given the class in which the routine arises), specifically the parameters, local variables, and accessible attributes. It is also necessary to introduce a new type for specifications so that the *frame* of a specification can be expressed.

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

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

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

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

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

The `spec_ax` axiom states that for a routine the prestate and poststate of a specification are that of the routine, and the value of the specification is a function from pre and poststate to a boolean, where the boolean is *true* if and only if the precondition implies the postcondition (we omit the time variable from the PVS translation, but it is trivial to include).

The generic class *SEQUENCE*$[G]$ is defined in [MEY 92]; it represents a packaged, indexable data structure of arbitrary but finite length. Here is an excerpt of its interface. *item* returns the specified item in the sequence, while *head* and *tail* return the first element and all but the first element in the sequence, respectively. *subseq*(t) returns *true* iff t is a subsequence of the current object, while *precedes*$(g1, g2)$ is *true* iff element $g1$ occurs before $g2$ in the sequence. In producing the PVS formalization of *SEQUENCE*, we use the built-in theory of finite sequences.

class *SEQUENCE*$[G]$ **feature**
  size : *INTEGER*
  item$(i$: *INTEGER*$)$: G
  tail : *SEQUENCE*$[G]$
  head : G
  subseq$(t$: *SEQUENCE*$[G])$: *BOOLEAN*
  precedes$(g1, g2 : G)$: *BOOLEAN*
invariant *size* ≥ 0
end

Informally, a message appearing in a collaboration diagram corresponds to a routine call invoked on one or more target objects. More formally, a message in a collaboration diagram consists of a source and a target, a routine (which is the implementation of the message) and a message number. In general, the source and target may be sets of objects, but for simplicity we consider only the case where a message is sent from and to a single object. Recursive rules are given in [WAL 95] for unrolling messages applied to clusters; the extension of the PVS specification to sets of objects is straightforward and is effectively a "lifting" operation.

class *MESSAGE* **feature**
  source, *target* : *OBJECT*
  routine : *ROUTINE*
  number : *INTEGER*
invariant *number* ≥ 1
end

The PVS specification of messages is straightforward and can be found in [PAI 02a]. Specifying collaboration diagrams requires extending the metamodel specification from [PAI 01b]. Particularly, the class *MODEL* must be extended. A model consists of a set of abstractions (which may be clusters, objects, classes, and object clusters) and a set of relationships. To this class, we add, via inheritance, several private features that will be used to produce all abstractions and relationships that make up collaboration diagrams (multiple collaboration diagrams are permitted in our formalisation).

As discussed earlier, BON obeys the single model principle [PAI 02b], in that a unique model of a system exists from which different views can be generated. In this way, consistency of views is guaranteed by construction. Thus, in the metamodel for BON, there is a unique class, *MODEL*, defining the well-formedness constraints on models. Features of this class can be used to generate views. New views can be added

by inheriting from *MODEL* and adding new features. It is not within the spirit of BON to add new views by adding new subclasses of *MODEL*, e.g., *DYNAMIC_MODEL*, etc., as this can easily introduce inconsistency between views.

The existing class *MODEL* includes all features and constraints necessary to model collaboration diagrams. However, it is inconvenient to use for validating the consistency of class diagrams and collaboration diagrams directly. Thus, for convenience, we restructure *MODEL* slightly using inheritance, and introduce several new features for checking the consistency of class diagrams and collaboration diagrams. In particular, we add a feature *occurs*, representing the set of objects appearing in a model; *init*, an initial message that is sent in the collaboration diagram; the features *sequence* and *calls* representing, respectively, the sequence of messages and the sequence of routine calls appearing in a collaboration diagram (the latter is a projection of the former), a scenario box (a free-form block of text describing what is represented by the messages), and queries for producing the collaboration diagram view and the class diagram view from the single model. As well, invariant clauses are added to the extension of *MODEL*; further clauses will be added shortly for checking the consistency of the views. Here is the interface of *EXTENDED_MODEL*. Note that all additional features, with the exception of those for generating new views, are private.

```
class EXTENDED_MODEL inherit MODEL
feature { NONE }
    occurs : SET[OBJECT]
    sequence : SEQUENCE[MESSAGE]
    scenario_box : TEXT
    init : BOOLEAN
    calls : SEQUENCE[ROUTINE]
feature { ANY }
    class_diagram, collab_diagram : EXTENDED_MODEL
invariant
    msgs_in_rels;
    calls_linked_to_msgs;
    objects_in_occurs;
    occurs ⊆ abs;
    calls.length = sequence.length
end
```

The invariant clauses that are named, but not defined above, are as follows. The clause *msgs_in_rels* says that each message in the *sequence* is also a relationship in the model.

$$\forall m \in sequence \bullet m \in rels$$

The clause *calls_linked_to_msgs* states that call *i* in sequence *calls* is the routine associated with message *i* in *sequence*.

$$\forall\, i : 0, .., sequence.length \bullet sequence.item(i).routine = call.item(i)$$

objects_in_occurs states that each object in the source or target of a message occurs in the collaboration diagram.

$$\forall\, m \in sequence \bullet m.source \in occurs \wedge m.target \in occurs$$

(Note that these constraints do not specify anything about consistency of views.)

The queries *class_diagram* and *collab_diagram* produce the two views of the model. These are defined as follows, and are simple projection operations.

class_diagram : *EXTENDED_MODEL*
ensure $Result.abs = \{a \in abs \mid a : STATIC_ABS\}$
$Result.rels = \{r \in rels \mid r : STATIC_REL\}$

collab_diagram : *EXTENDED_MODEL*
ensure $Result.abs = \{a \in abs \mid a : DYNAMIC_ABS\}$
$Result.rels = \{r \in rels \mid r : MESSAGE\}$

To express *EXTENDED_MODEL* in PVS, a new subtype could then be introduced (representing the type of extended models), and then each new routine and constraint in the BON class could be mapped to a PVS function. However, a new PVS subtype is not strictly needed, since PVS does not provide the object-oriented structuring facilities of BON; in other words, metamodel extension is not restricted to use of inheritance. Thus, adding new functions or attributes to the PVS specification is in fact easier than adding new features to classes in BON. The translation process is as follows. We introduce new functions (that operate on variables of type `MODEL`) representing the additional features that we require.

```
 objects_in_model:  [ MODEL -> set[OBJECT] ]
 sequence_model: [ MODEL -> finseq[MESSAGE] ]
 calls_model:    [ MODEL -> finseq[ROUTINE] ]
```

The invariant clauses in *EXTENDED_MODEL* will each be mapped to PVS axioms. Here is an example, stating that *calls* is a projection of *sequence* (the other axioms are straightforward translations of the BON constraints).

```
calls_linked_ax: AXIOM
 (FORALL (mod1:MODEL):
  (FORALL (i:{j:nat|j<length(sequence_model(mod1))}):
   routine_message(sequence_model(mod1)(i))=calls_model(mod1)(i)))
```

To formalize the routines *class_diagram* and *collab_diagram*, we introduce two new functions, `collab_diagram` and `class_diagram`. The specifications of each are similar, so we present only the PVS specification of *collab_diagram* here.

```
collab_diagram(mod1:MODEL): MODEL =
  (# abst := { da:DYN_ABS | member(da,abst(mod1)) },
     rels := { m:MESSAGE | member(m,rels(mod1)) } #)
```

Consistency between views will be specified as four invariant clauses belonging to the class *EXTENDED_MODEL*. For each clause, a PVS formulation is provided when it cannot be found in [PAI 01b]. (In the following, we use *dd* to stand for *collab_diagram* and *cd* for *class_diagram*, respectively.)

1) Each object appearing in the collaboration diagram has a corresponding class in the class diagram.

$$\forall\, o \in dd.occurs \bullet \exists\, c \in cd.abs \mid c.type : CLASS \bullet o.class = c$$

(Note that *cd.abs*, defined in [PAI 01b], is the set of abstractions appearing in the class diagram.)

2) Each message in the collaboration diagram has a corresponding routine call, and that call is permitted based on the list of accessors provided with each routine.

$$\forall\, msg \in dd.sequence \bullet \forall\, o \in msg.source \bullet o.class \in msg.routine.accessors$$

3) Each routine appearing in a message must actually belong to the target class of the message (i.e., routines that are called must exist). This will be checked by the compiler/CASE tool and as such we do not specify it here. However, it is captured in the full specification of the BON metamodel referenced in [PAI 01b]. The constraint in [PAI 01b] is more general in that it checks all features (including attributes) to ensure that they exist. This ensures that if a message is sent from one object to another, there is a link between the two objects.

4) The constraint in 2) establishes that each message in a collaboration diagram corresponds to a routine call. The routines that are called must be enabled (i.e., their preconditions must be true) for the collaboration diagram to be consistent with a class diagram. A precondition can only be true if the sequence of previous calls to routines established a system state that satisfies the precondition. To check this, an initial state, *init*, must be provided (by the developer). The following condition must be true.

$$init \rightarrow dd.calls.item(1).pre$$

i.e., the developer-supplied initial state, specified as a predicate), must imply the precondition of the first element in the sequence of calls in the collaboration diagram. *init* corresponds to the specification of a constructor of the root class in the system, where the specification includes relevant clauses from the invariant of the root class and the

postcondition of the constructor (constructors cannot have preconditions).

For a call $i \geq 2$ in a collaboration diagram to be enabled, the preceding sequence of calls $1, .., i-1$ must produce a state satisfying the precondition of call *i*. We can obtain this state by first sequentially composing the specifications of *init* and calls $1, .., i-1$. This results in a double-state predicate (i.e., in the user-supplied initial state and in the post-state of call $i-1$). We then project out the post-state and check that the result satisfies the pre-state of call *i*. Formally:

$$\forall i : 2, .., dd.calls.length \bullet$$
$$\exists\, dd.occurs \bullet$$
$$(init \,\text{⨾}\, dd.calls.item(1).spec \,\text{⨾} \ldots \text{⨾}\, dd.calls.item(i-1).spec) \rightarrow$$
$$dd.calls.item(i).pre$$

(The definition of sequential composition for specifications P and Q on state s is:

$$P \,\text{⨾}\, Q \quad \widehat{=} \quad \exists\, s' \cdot P[s := s'] \wedge Q[\mathbf{old}\ s := s']$$

where s' is an intermediate state, i.e., for every sequential composition, there is an implicit existential quantification that needs to be instantiated and simplified.)

Expressing constraint 4) in PVS is challenging. The first part, enabling the first message in the collaboration diagram, can be done as follows. *init* is translated to a function mapping a model and a class (the root class) to a boolean. The enabling of the first message is formalised as an axiom.

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

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

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

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

```
  seqspecs(s1,s2:SPECTYPE): SPECTYPE =
     (# old_state := old_state(s1),
```

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

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

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

To complete the PVS formalization of constraint 4), it is helpful to define a function to convert a sequence of messages into a finite sequence of `SPECTYPE`s. This function, `convert`, extracts the routines from the messages and produces specifications from them, by repeated application of function `spec`. Its details can be found in [PAI 02a].

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

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

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

In order to use the theories in carrying out view consistency checking, it is necessary to parse and typecheck the theories using PVS. When we carried out this semi-automatic process, it revealed to us several errors and omissions in the PVS theories. One such example was in the formulation of `seqspecsn`, wherein an erroneous

`MEASURE` was initially provided; an unprovable type check condition was generated. Further omissions were discovered when attempting to prove view consistency conjectures; this is briefly discussed in the next section.

The correctness of the translation of BON constraints to PVS axioms and types has not been proven. In general, proving the correctness of translations for large, full-featured languages is extremely difficult. We view our PVS specification of the metamodel as a mechanism by which the translation can be *tested.* We can write conjectures about properties that we would like the metamodel to have, and can use the PVS system to prove or disprove the conjectures. This will give us greater confidence in the correctness of the metamodel and the translation.

4. Using the PVS Theories

To use the PVS theories for proving view consistency, a BON model can be specified as a PVS conjecture, following the approach presented in [PAI 01b]. These conjectures effectively posit that the model can exist. They must therefore satisfy the consistency constraints as specified in the metamodel. PVS can then be used to import the view consistency axiom as above, and one can then attempt to prove or disprove that the axiom is satisfied by the models.

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

This approach is demonstrated by a small example. Consider the two BON views presented in Fig. 5. The class diagram depicts two classes connected by a client-supplier relationship, whereas the collaboration diagram shows three messages sent between objects. It is assumed that object B is initialised to a state where $i = 4$ and set s is empty. To check the consistency of these views, the two models must be expressed as PVS conjectures, following the approach in [PAI 01b].

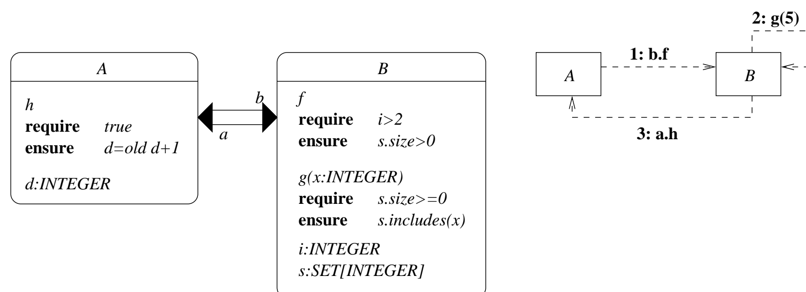

Figure 5. *Two views of a BON system*

A snapshot of the PVS specification of Fig. 5 is shown below. The transformation from BON to the PVS conjecture `vc_example` is mechanical and could be automated. To validate the conjecture (and to show that the BON model can exist and satisfies the view consistency constraint), the conjecture must first be simplified using `(skolem!)` and `(flatten)`. Then, `(lemma)` is used to import the view consistency axioms. A succession of `(replace)`, `(assert)`, and `(expand)` instructions are made to simplify the conjecture and to replace functions with their definitions. Then `(grind)` can be used to discharge the remaining PVS subgoals that arise.

```
vc_example: THEORY
 BEGIN

  IMPORTING metamodel

  ...

  test_vc: CONJECTURE
  (NOT (EXISTS (a,b:CLASS):
   EXISTS (ao,bo:OBJECT):
   EXISTS (atob,btoa:ASSOC):
   EXISTS (m1,m2,m3:MESSAGE):
   EXISTS (bf,ag,ah:ROUTINE):
   EXISTS (sbf,sbg,sah:SPECTYPE):
   EXISTS (ei,es,ed,old_ei,old_es,old_ed,param_x:ENTITY):
   EXISTS (xm:MODEL):
   length(calls_model(xm))=3 AND
   length(sequence_model(xm))=3 AND
   member(ao,objects_model(xm)) AND member(bo,objects_model(xm)) AND
   member(a,abst(xm)) AND member(b,abst(xm)) AND
   object_class(ao)=a AND object_class(bo)=b AND
   member(atob,rels(xm)) AND member(btoa,rels(xm)) AND
   member(m1,rels(xm)) AND member(m2,rels(xm)) AND member(m3,rels(xm)) AND
   routine_message(m1)=bf AND routine_message(m2)=bg AND routine_message(m3)=ah AND
   member(old_ei,oldstate(bf,b)) AND member(old_es,oldstate(bf,b)) AND
   member(old_ei,oldstate(bg,b)) AND member(old_es,oldstate(bg,b))
     AND member(param_x,oldstate(bg,b)) AND
   member(old_ed,oldstate(ah,a)) AND
   member(ei,newstate(bf,b)) AND member(es,newstate(bf,b)) AND
   member(ei,newstate(bg,b)) AND member(es,newstate(bg,b)) AND
   member(param_x,newstate(bg,b)) AND
   member(ed,newstate(ah,a)) AND
   feature_pre(bf,oldstate(bf,b))=(eval_i(old_ei)>2) AND
   feature_post(bf,oldstate(bf,b),newstate(bf,b))=(nonempty?(eval_s(es))) AND
   feature_pre(bg,oldstate(bg,b))=(nonempty?(eval_s(old_es)) OR empty?(eval_s(old_es))) AND
   feature_post(bg,oldstate(bg,b),newstate(bg,b))=(member(eval_i(param_x),eval_s(es))) AND
   feature_pre(ah,oldstate(ah,a))=(true) AND
   feature_post(ah,oldstate(ah,a),newstate(ah,a))=(eval_i(ed)=eval_i(old_ed)+1) AND
   calls_model(xm)(0)=bf AND calls_model(xm)(1)=bg AND calls_model(xm)(2)=ah AND
   sequence_model(xm)(0)=m1 AND sequence_model(xm)(1)=m2 AND sequence_model(xm)(2)=m3 AND
   sbf=spec(bf,oldstate(bf,b),newstate(bf,b)) AND
   sbg=spec(bg,oldstate(bg,b),newstate(bg,b)) AND
   sah=spec(ah,oldstate(ah,a),newstate(ah,b)) AND
   init(xm,oldstate(bg,b))=(eval_i(old_ei)=4 AND empty?(eval_s(old_es)))))
 END vc_example
```

We have carried out case studies in using the theories to prove view consistency using the above approach. Initial attempts at proving view consistency revealed errors and omissions in the PVS theories. One example was with the formalisation of the second part of constraint 4) (which aims to show that a sequence of calls in a

collaboration diagram enables successive calls). The initial formulation omitted the initialisation from the formalisation, and the PVS prover provided a counter-example that led us to conclude that consistency could not be proved.

5. Related Work and Conclusions

The introduction of the UML has spurred recent research on consistency checking, but the topic has been of previous interest in a number of problem domains. Zave and Jackson [ZAV 93] presented a framework for composing specifications via conjunction, with the aim of supporting multi-paradigm specification. Specifications are transformed in to a common semantic domain (in [ZAV 93], they use one-sorted first order logic, but different semantic domains can be chosen) and thereafter combined. They pay particular attention to constructing translations to the common semantic domain so that specifications can be easily and usefully composed, e.g., so as to make consistency as straightforward as possible to establish. The authors' goal is not specifically consistency checking of views, but suggestions and recommendations as to how to use the approach to make consistency checking easier to carry out are provided. They do not specifically focus on the OO realm, and do not explicitly consider tool support. They recognize the problem of semantic fragmentation, i.e., providing a non-standard semantics to commonly used languages, and the problems with formalizing semi-formal notations.

Finkelstein et al. [FIN 94] focus specifically on the problem of detecting inconsistency when combining descriptions of systems from multiple viewpoints. Their work emphasizes that inconsistency is not always undesirable, and that in fact it may provide important information to developers, e.g., related to misunderstandings or confusion with respect to requirements. Thus, their logical framework aims to support developers in identifying inconsistencies and specifying actions to carry out on their identification. Consistency checking is carried out by producing a logical database of formulae describing separate views, as well as further formulae specifying environmental information, e.g., relationships between views. Consistency or inconsistency checking can be carried out using automated theorem provers. This work is related to that carried out at the workshop [EAS 01], where tools and theories for working with inconsistent descriptions of system requirements are presented.

The ADORA project [GLI 01] presents an alternative to UML for OO modelling, wherein all information related to a system is integrated into one coherent model. In this latter regard, it is similar to the *single model principle* described in [PAI 02b]. The integrated model allows consistency constraints to be defined between views. A language and tool for supporting these constraints is discussed in [SCH 98]. Some of the constraints that are checked by this tool are also captured in the UML metamodel, and as such are checked by UML-compliant CASE tools.

Tsiolakis [TSI 01] focuses specifically on consistency checking with the UML, primarily, consistency checks relating class diagrams, sequence diagrams, and state

charts. Diagrams are annotated with additional information relating the separate views, and attributed graph grammars are used as a theoretical underpinning to carry out the consistency checking.

Krishnan [KRI 00] presents work most similar to our own. He presents a PVS framework for consistency checking of UML diagrams, focusing on class diagrams, interaction diagrams, and statecharts (though also including use-case diagrams and others). The PVS theories developed aim at allowing consistency to be checked even when diagrams are partial or incomplete. Using the PVS theories to prove consistency involves the user providing a valid *trace* for execution of the system. Interestingly, the proof strategies and techniques used in Krishnan's approach closely mimic the ones that we discussed in the previous section. However, Krishnan's theories do not include OCL constraints and contracts, though the work could be extended to include them. Krishnan's approach also appears to require more user intervention to generate PVS conjectures about the existence or non-existence of BON models.

Our current focus is on implementing the consistency checking described in this paper. Many of the rules are currently built in to the metamodel implementation provided with the tool. The architecture of the tool makes it straightforward to add new rules to the metamodel, or to replace the metamodel entirely with a new set of rules. Some of the consistency checking cannot be carried out automatically or implemented in the metamodel, e.g., checking that the sequence of messages appearing in a collaboration diagram is allowable, based on contracts. The checks will be sent to the PVS theorem prover and discharged automatically where possible. The paper [PAI 01b] describes how we have successfully used PVS for semi-automatically proving that models satisfy the BON metamodel; the same approach can be used for consistency checking between views. As well, we are currently exploring the use of automated verification technology, particularly FDR, for carrying out the sequencing consistency checks. This will also be very useful for consistency checking of test drivers against collaboration diagrams, since we can effectively represent this as a constraint to be checked on traces.

The metamodel extension presents a specification of a consistency relation for collaboration diagrams and class diagrams. We might prefer to have an algorithmic description of the consistency checking process; however, we view an algorithmic description as an implementation of the consistency relation above. The paper [PAI 02a] outlines algorithms for implementing these specifications in a tool.

6. References

[BEC 99] BECK K., *Extreme Programming Explained*, AWL, 1999.

[BOO 99] BOOCH G., RUMBAUGH J., JACOBSON, I., *The UML Reference Guide,* Addison-Wesley, 1999.

[BRI 01] BRIAND L., LABICHE, Y., "A UML-Based Approach to System Testing", *Proc. UML 2001*, LNCS 2185, Springer-Verlag, 2001.

[CLA 02] CLARK T., EVANS A., KENT, S., "Engineering Modelling Languages: a Precise Meta-modelling Approach", *Proc. FASE 2002*, LNCS 2306, Springer-Verlag, 2002.

[EAS 01] EASTERBROOK S., CHECHIK M., *Proc. Second International Workshop on Living with Inconsistency*, co-located with *ICSE'01*, Toronto, Canada, May 2001.

[GLI 01] GLINZ M., BERNER S., JOOS S., RYSER J., SCHETT N., XIA Y., "The ADORA Approach to Object-Oriented Modeling of Software", *Proc. CAiSE'01*, LNCS 2068, Springer, June 2001.

[FIN 94] FINKELSTEIN A., GABBAY D., HUNTER A., KRAMER J., NUSEIBEH B., "Inconsistency Handling in Multi-Perspective Specification" *IEEE Trans. Software Engineering* 20(8), August 1994.

[KRI 00] KRISHNAN P., "Consistency Checks for UML", *Proc. APSEC 2000*, IEEE Press, December 2000.

[MEY 92] MEYER B., *Eiffel - The Language*, Prentice-Hall, 1992.

[MEY 97] MEYER B., *Object-Oriented Software Construction (Second Edition)*, Prentice-Hall, 1997.

[OMG 00] OMG CONSORTIUM, *UML 1.4 Documentation*, 2000. Available at www.omg.org.

[OWR 01] OWRE S., SHANKAR N., RUSHBY J., STRINGER-CALVERT, D., *PVS System Guide 2.4*, CSL, SRI International, November 2001.

[PAI 01a] PAIGE R., KAMINSKAYA L., "A Tool-Supported Integration of BON and JML", Technical Report CS-TR-2001-04, York University, July 2001.

[PAI 99] PAIGE R., OSTROFF J., "Developing BON as an industrial-strength formal method", *Proc. World Congress on Formal Methods*, LNCS 1709, Springer-Verlag, September 1999.

[PAI 01b] PAIGE R., OSTROFF J., "Metamodelling and conformance checking with PVS", *Proc. Fundamental Aspects of Software Engineering 2001*, LNCS 2029, Springer-Verlag, April 2001.

[PAI 01c] PAIGE R., OSTROFF J., "The Single Model Principle (Extended Abstract)", *Proc. Requirements Engineering 2001*, IEEE Press, August 2001.

[PAI 01d] PAIGE R., OSTROFF J., "A Proposal for a UML-Based Method for Developing Reliable Systems", *Proc. Workshop on Precise UML-Based Methods*, GI Series 7, Lecture Notes in Informatics, German Society, October 2001.

[PAI 02a] PAIGE R., OSTROFF J., BROOKE P., "Checking the Consistency of Class and Collaboration Diagrams using PVS", *Proc. Rigorous Object-Oriented Methods 4 (ROOM4)*, British Computer Society, March 2002.

[PAI 02b] PAIGE R., OSTROFF J., "The Single Model Principle", *Journal of Object Technology* 1(5):63-81, November/December 2002.

[SCH 98] SCHETT N., *A Notation for Integrity Constraints in* ADORA *Models - Concept and Implementation* (in German). Diplomathesis, University of Zurich, 1998.

[TSI 01] TSIOLAKIS A., *Semantic Analysis and Consistency Checking of UML Sequence Diagrams.* Diplomarbeit, TU-Berlin, TR 2001-06, April 2001.

[WAL 95] WALDEN K., NERSON J.-M., *Seamless Object-Oriented Software Architecture*, Prentice-Hall, 1995.

[ZAV 93] ZAVE P., JACKSON M., "Conjunction as Composition", *ACM Transactions on Software Engineering and Methodology* 2(4), October 1993.