Checking the Consistency of Collaboration and Class Diagrams using PVS

Richard F. Paige[1], Jonathan S. Ostroff[2], and Phillip J. Brooke[3]

[1] Department of Computer Science, University of York,
Heslington, York YO10 5DD, United Kingdom. paige@cs.york.ac.uk
[2] Department of Computer Science, York University,
4700 Keele Street, Toronto, Ontario M3J 1P3, Canada. jonathan@cs.yorku.ca
[3] School of Computing, University of Plymouth,
Drake Circus, Plymouth, Devon, PL4 8AA, United Kingdom. philb@soc.plym.ac.uk

Abstract. We present a formal, mechanically checked specification of the consistency constraints between two views of object-oriented systems described in BON: the static view provided by class diagrams annotated with contracts, 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. A sketch of how the constraints might be implemented in the BON CASE tool is also provided.

1 Introduction and Motivation

Consistency checking of documents has long been an important task in software development. It has particularly been emphasized with recent work on viewpoint specification [6] and combining specifications [18]. With the recent interest in UML [2], the consistency checking of independently constructed models of a software system has become of increasingly significant interest.

Consistency checking of software models involves the use of constraints, algorithms, and tools to check that information described in one model is not contradicted by information described in another model. In a setting where formal specifications of models are available, this is essentially the problem of checking that a conjunction of predicates – each a formal specification of a model – 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.

UML is a particularly interesting language for describing software systems and maintaining consistency of description. It supports the use of up to four different views of a single software system. These views may be constructed independently, may overlap, and thus may contradict each other. The intent of using multiple views is to describe different aspects or elements of a system in the most appropriate way. The different descriptions are to be combined to form a consistent, complete, checkable description of the software system that can be used thereafter to generate executable code. This problem is not unique to UML; any modelling language that supports disparate views must be supplemented with techniques for checking consistency. In the case of UML, specifying a complete set of consistency constraints is challenging. Many are specified in the UML metamodel [7] and certain UML-compliant CASE tools implement these. However, some of the complex constraints, e.g., involving the use of *contracts* [5] of methods, are not implemented in any tool, and thus developers must rely on their own expertise to identify and resolve inconsistencies. Some CASE tools in turn implement the consistency constraints by requiring diagrams to be constructed in a specific order, thus preventing inconsistencies from arising by construction. This may be too inflexible in general.

UML supports two fundamental models: *class diagrams* and *collaboration diagrams*. These diagrams present, respectively, static structural information about a system, as well as dynamic information about behaviour, 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 a number of object-oriented (OO) modelling languages, including BON [17].

The aim of this paper is to formally specify (in a machine-checkable language) and describe implementations of consistency constraints between the two BON views – specifically, class diagrams and collaboration diagrams – that can be produced during software development. Formulating the constraints is made more challenging by the fact that BON class diagrams also include contracts, expressing conditions on when methods of a class can be called, and the effect of these methods on the state of an object.

The need for these constraints arose during the proposal of a methodology for building reliable systems [13]. This methodology is novel in that it integrates object-oriented modelling and the practices of Extreme Programming (XP) [1], thus allowing developers to work with code, test drivers, and models, as needed. This motivates the need to allow development products – e.g., code and models – to be consistent by construction, and also to provide tools to check the consistency of products.

The consistency constraints between views are specified as an extension of the metamodel of BON presented first in [11], and implemented in the BON CASE tool [9]. 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, we will have to extend the metamodel carefully, making use of OO extension facilities. The constraints will also be specified in PVS [8], 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. The intent is thus 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.

We shall use the BON language for describing the two different views applied by the methodology, in part because of its simplicity and our familiarity with it. However, the rules that we present 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.

1.1 Overview

We commence with a brief overview of BON, and present an example of each type of BON diagram. We also briefly outline the methodology of [13], as motivation for constructing the consistency constraints. Two approaches to consistency checking of views are identified: *direct* and *indirect*. We explain direct consistency checking of class diagrams against collaboration diagrams, specify the constraints in PVS, and outline an implementation. We sketch how indirect consistency checking via test drivers can be done as well; this is the key link between modelling and XP test drivers, which are interpreted as implementations or refinements of collaboration diagrams. We then discuss related and ongoing work.

2 Background

We outline BON, focusing on the elements of their syntax used throughout the remainder of the paper. In particular, we will make use of the BON text-based notation for writing classes and interfaces, and its graphical notation for writing collaborations. We then briefly summarize the methodology that integrates OO modelling and XP from [13], since the methodology provides the motivation for constructing the consistency constraints.

2.1 BON

BON 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 summarized in [11].

BON models consist of one or more classes 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. Client-supplier relationships can be drawn between classes and clusters; recursive rules are given in [17] to explain the meaning of cluster relationships.

```
class CITIZEN  inherit PERSON
feature {ANY}
    name, sex : STRING
    age : INTEGER
    spouse : CITIZEN
    children, parents : SET[CITIZEN]
    single : BOOLEAN  is
        ensure Result = (spouse = Void)
    divorce is
        modifies spouse
        require ¬ single
        ensure single ∧ (old spouse).single

invariant
    single_or_married: single ∨ spouse.spouse = Current;
    number_of_parents: parents.count ≤ 2;
    symmetry: ∀ c ∈ children • ∃ p ∈ c.parents • p = Current
end
```

Fig. 1. Class *CITIZEN*

BON also provides notation for collaboration diagrams, showing the communication between objects. Fig. 2 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.

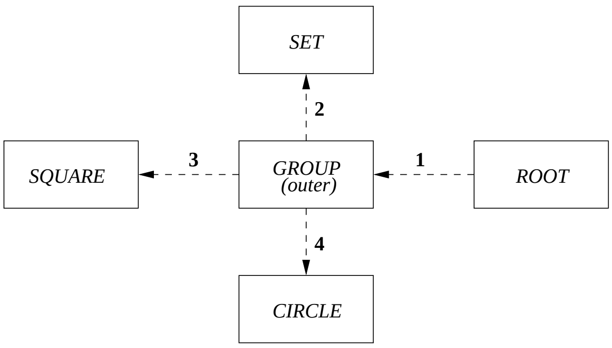

Fig. 2. BON collaboration diagram (without a scenario box)

2.2 A methodology integrating XP and object-oriented modelling

The motivation for being able to check the consistency of contract-annotated class diagrams and collaboration diagrams comes from the proposal of a methodology integrating XP and OO modelling. A draft of this methodology was presented in [13]. The methodology, as proposed, makes use of a selection of OO modelling diagrams, source code (e.g., in Eiffel or Java or some other suitable OO programming language), and test drivers. Test drivers provide a fundamental link between OO modelling and XP practices.

A key difficulty in integrating XP and OO modelling is allowing code to be written before modelling, and to allow modelling before writing code – i.e., to allow developers to select the work product to use at the start of development. It is essential to allow this level of flexibility, so that testing can be carried out when desired, and so that the abstraction capabilities of modelling languages can be fully exploited.

Elements of the methodology are summarized in Fig. 3. The diagram uses UML's package and dependency notations to illustrate the work products delivered by the methodology, and their relationships. The $\langle\langle derive \rangle\rangle$ and $\langle\langle refine \rangle\rangle$ stereotypes are standard UML. The $\langle\langle consistent \rangle\rangle$ stereotype is a refinement of $\langle\langle refine \rangle\rangle$ and indicates that the two products must be consistent either by construction or developer intervention.

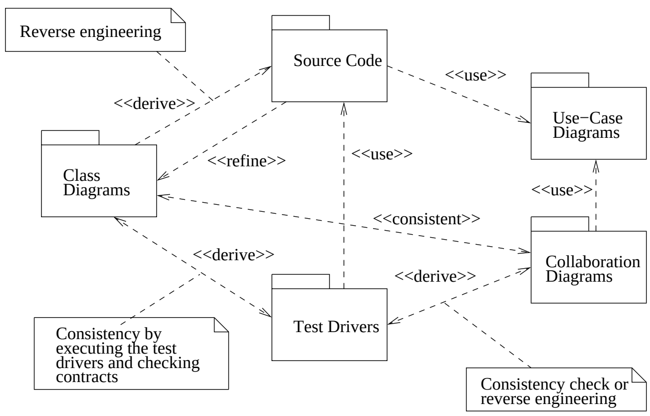

Fig. 3. Work products and their relationships

A key element of Fig. 3 is the relationship between test drivers and collaboration diagrams: a collaboration diagram is viewed as an abstraction of a test driver. Tool support is to be provided so that, given a collaboration diagram, the outline of an executable test driver can be produced; and, given a test driver, a collaboration diagram can automatically be produced. The second key aspect of this diagram is the relationship between class and collaboration diagrams: given one of each such diagrams, we desire to be able to show the consistency of the information contained in both descriptions.

Descriptions of how the relationships between work products can be established are outlined in [13]. In this paper, we focus on the relationships involving collaboration diagrams and class diagrams. Specifically, we will describe how consistency between these products can be established directly via use of a metamodel and tools such as PVS. We also sketch how test drivers can be used as an intermediate product for checking the consistency of the two different diagrams.

3 Foundations and Approaches

The eventual goal of our work is to be able to check any *work product* against any other work product for consistency. A work product, in the context of the methodology of [13], is defined as any one of:

- Machine-checkable Eiffel source code, where classes and routines are annotated with pre/postconditions and invariant clauses.
- A class diagram, showing classes, assertions, and class relationships.
- A collaboration diagram, showing objects and message passing.
- A test driver, which is an Eiffel class that executes a sequence of routines on one or more objects and checks that relevant conditions hold.

Informally, we desire to define a family of *consistency relations* on the work products of interest. We are specifically concerned, in this paper, with checking the consistency of class diagrams with contracts and collaboration diagrams. There are two general approaches to this. One approach is to define a set of rules that can be applied to a given class diagram and a given collaboration diagram. Due to the inclusion of contracts in the class diagram – which may make use of unexecutable constructs – this process may need tool support and user intervention to check consistency.

An alternative approach is to do *indirect* consistency checking of the two work products, via use of test drivers as an intermediate product. This works as follows. We wish to show that a class diagram CD is consistent with a collaboration diagram DD for a system. First, show that CD is consistent with the system's source code (this is easy, via forward or reverse engineering, as supported in any reasonable CASE tool). Then, show that the source code is consistent with any test drivers for the system (again, this is easy - we simply execute the test drivers and if the tests are accepted, the products are consistent). Finally, check that the collaboration diagram DD is consistent with the test drivers. This process is more complex and is discussed in the sequel. But if we can establish separate consistency of these three products, then the class diagram and the collaboration diagram will be consistent.

Thus, one approach to consistency checking of collaboration diagrams and class diagrams is by defining work product consistency in a transitive fashion, and by making use of test drivers as an intermediate product. But it is also desirable in the methodology to be able to check collaboration diagrams against class diagrams directly. This is important because developers may not want to have to create test drivers in order to do consistency checking (and the methodology of Fig. 3 supports this). In the next section, we specify consistency constraints between these diagrams, and then discuss how these constraints can be implemented in a CASE tool.

4 Checking Collaboration Diagrams Against Class Diagrams

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 not consistent, to report where the inconsistencies arise. Inconsistencies can arise due to object declaration (e.g., an object is unassociated with any class), 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.

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

1. Ensure that the two diagrams are syntactically correct; there is a BON CASE tool that will do this for us.
2. 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.*
3. 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 to validate.

We consider these steps in order, skipping (1) since the BON CASE tool is described elsewhere [9]. The remaining constraints will be specified as an extension of the metamodel of BON first presented in [11]. Since the BON CASE tool implements the BON metamodel, we thus have a way of ensuring that they are satisfied. At the same time, 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, e.g., those in (2), are easily implemented in a CASE tool. Other constraints are more complex, and thus we will express them using the PVS language, so that we can thereafter use the PVS system to check the constraints, either interactively or in batch mode.

Before specifying the rules in (2) and (3), we briefly recount the key parts of the BON metamodel from [11]. 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 individual classes appearing in the clusters $ABSTRACTIONS$ or $RELATIONSHIPS$. New constraints for the rules in (2) and (3) will also be integrated into the metamodel as invariant clauses (as we discuss shortly).

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

Fig. 4. BON metamodel, abstract view

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 [11]. A routine has a name, a possibly empty sequence of parameters, a set of accessors, a pre- and postcondition, and a specification, which corresponds to the semantics of the routine. (In [11], 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 $ROUTINE$.

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

$spec$ is the semantics of the 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 [10], where a calculus for refining BON specifications to Eiffel programs is provided.

Part of the PVS formulation of the BON class $ROUTINE$ is given below; missing details may be found in [11]. 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) 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 the concept of a routine's specification in PVS is more complicated. The complication does not arise in expressing a specification directly, but in *combining* specifications. Thus, our 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 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 time variables from the PVS translation for simplicity, but it is straightforward to add them).

The class $SEQUENCE$ is defined in [5]; 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 notion of a finite sequence.

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

$\textbf{invariant}\ size \geq 0$
$\textbf{end}\ -\ SEQUENCE[G]$

Now we can specify the concept of a message that appears in a collaboration diagram. Messages were specified in [11], but we modify the definition slightly here. Informally, a message 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 [17] for unrolling messages applied to clusters; the extension of the PVS description is straightforward.

$$
\begin{array}{l}
\textbf{class}\ MESSAGE\ \textbf{feature} \\
\quad source, target : OBJECT \\
\quad routine : ROUTINE \\
\quad number : INTEGER \\
\textbf{invariant}\ \ number \geq 1 \\
\textbf{end}\ -\ MESSAGE
\end{array}
$$

A partial specification of the PVS formulation of messages follows. A new type is introduced, and the features of the $MESSAGE$ class are represented as functions on the new type.

```
MESSAGE: TYPE+ FROM ABS
routine_message: [ MESSAGE -> ROUTINE ]
number_message: [ MESSAGE -> nat ]
```

Now we can specify the concept of a collaboration diagram. This requires us to extend the specification of the metamodel from [11]. Specifically, we must extend the class $MODEL$. 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[1].

> **Aside.** BON obeys the *single model principle* [12], in that a unique model of a system exists, from which different views can be generated. In this way, consistency of views is guaranteed. 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 $MODEL$,

[1] The specification supports an arbitrary number of collaboration diagrams and class diagrams in a 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 representing the set of objects appearing in a model, the sequence of messages and routine calls appearing in a collaboration diagram, 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 new features, with the exception of those for generating new views, are private.

class $EXTENDED\_MODEL$ **inherit** $MODEL$
feature $\{NONE\}$
  $occurs : SET[OBJECT]$
  $sequence : SEQUENCE[MESSAGE]$
  $scenario\_box : TEXT$
  $calls : SEQUENCE[ROUTINE]$
feature $\{ANY\}$
  $class\_diagram, collab\_diagram : EXTENDED\_MODEL$
invariant
  $msgs\_in\_rels; \quad calls\_linked\_to\_msgs;$
  $objects\_in\_occurs; \quad objects\_in\_abs;$
  $same\_lengths$
end $-$ $EXTENDED\_MODEL$

The invariant clauses 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 each call appearing in sequence $calls$ at element i is the routine associated with the message appearing in $sequence$ at element i.

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

$objects\_in\_occurs$ states that each object in the source or target of a message occurs in the collaboration diagram.

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

$objects\_in\_abs$ is similar, and says that each object in the set $occurs$ is in the set of abstractions belonging to the model.

$$occurs \subseteq abs$$

Finally, $same\_lengths$ states that $calls$ and $sequence$ are the same length.

$$calls.length = sequence.length$$

(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.

$$\begin{aligned}&class\_diagram : EXTENDED\_MODEL\\ &\textbf{ensure}\ Result.abs = \{a \in abs \mid a : STATIC\_ABS\};\\ &\qquad Result.rels = \{r \in rels \mid r : STATIC\_REL\}\end{aligned}$$

$$\begin{aligned}&collab\_diagram : EXTENDED\_MODEL\\ &\textbf{ensure}\ Result.abs = \{a \in abs \mid a : DYNAMIC\_ABS\};\\ &\qquad Result.rels = \{r \in rels \mid r : MESSAGE\}\end{aligned}$$

To express $EXTENDED\_MODEL$ in PVS, we could introduce a new subtype (representing the type of extended models), and then could map each new routine and constraint in the BON class 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. 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, i.e., representing class diagrams, collaboration diagrams, etc.

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

class_diagram collab_diagram:  [ MODEL -> MODEL ]
```

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. The specifications of each are similar, so we present only the PVS specification of $collab\_diagram$ here.

```
collab_diagram_ax: AXIOM
(FORALL (mod1:MODEL):
  abst(collab_diagram(mod1)) = { da:DYN_ABS | member(da,abst(mod1))} AND
  rels(collab_diagram(mod1)) = { m:MESSAGE | member(m,rels(mod1)) })
```

Let xm be an extended model. Consistency between views will be specified as four invariant clauses in $EXTENDED\_MODEL$. For each clause, a PVS formulation is provided when it cannot be found in [11]. (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 [11], 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 [11]. The constraint in [11] 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). A precondition can only be true if the sequence of previous calls to routines left the state of the system satisfying 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.
   For a call $i \geq 2$ 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 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:

$$\forall\, i : 2, .., dd.calls.length \bullet$$
$$\exists\, dd.occurs \bullet$$
$$(dd.calls.item(1).spec \;;\; \ldots \;;\; dd.calls.item(i-1).spec) \rightarrow$$
$$dd.calls.item(i).pre$$

(Recall that the definition of sequential composition is:

$$P;\; Q = \exists\, s' \cdot P[s := s'] \wedge Q[\mathbf{old}\; s := s']$$

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 set of entities (the state) to a boolean.

```
init: [ MODEL, set[ENTITY] -> bool ]

init_ax: AXIOM
(FORALL (mod1:MODEL):
  (FORALL (old_s:set[ENTITY]):
    init(mod1,old_s) IMPLIES feature_pre(calls_model(mod1)(0),old_s)))
```

The second part is much 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 $P;\; Q$ 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: [ SPECTYPE, SPECTYPE -> SPECTYPE ]

  seqspecs_ax: AXIOM
  (FORALL (s1,s2: SPECTYPE):
     seqspecs(s1,s2) =
     (# 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 function `seqspecsn`.

```
  seqspecsn: [ finseq[SPECTYPE] -> SPECTYPE ]

  seqspecsn_ax1: AXIOM
  (FORALL (seq1: finseq[SPECTYPE]):
    length(seq1)=2 IMPLIES seqspecsn(seq1) = seqspecs(seq1(0),seq1(1)))

  seqspecsn_ax2: AXIOM
  (FORALL (seq1:finseq[SPECTYPE]):
     length(seq1)>2 IMPLIES
       seqspecsn(seq1) = seqspecs(seq1(0),seqspecsn(rest(seq1))))
```

To complete the PVS formalization of constraint (4), we define a function to convert a sequence of messages into a finite sequence of `SPECTYPE`s. Effectively, this extracts the routines from the messages and produces specifications from them.

```
convert: [ finseq[MESSAGE] -> finseq[SPECTYPE] ]

convert_ax: AXIOM
(FORALL (msgs1:finseq[MESSAGE]):
  length(msgs1)=length(convert(msgs1)) AND
  (FORALL (i:{j:nat|j<length(msgs1)}):
    (EXISTS (c:CLASS):
      member(routine_message(msgs1(i)),class_features(c)) IMPLIES
       convert(msgs1)(i) =
        spec(routine_message(msgs1(i)),
             oldstate(routine_message(msgs1(i)),c),
             newstate(routine_message(msgs1(i)),c)))))
```

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

```
views_consistent: AXIOM
(FORALL (mod1:MODEL):
  (FORALL (i:{j:nat|0<j AND j<length(calls_model(mod1))}):
    LET loc_spec:SPECTYPE = (seqspecsn(convert(sequence_model(mod1)^(0,i-1)))) IN
     (EXISTS (new:set[ENTITY]):
      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))))))))
```

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.

To use the axiom, we can specify a set of BON models as PVS conjectures, following the approach presented in [11]. These conjectures effectively posit that the set of models can exist. They must therefore satisfy the consistency constraints. We can then use PVS to import the view consistency axiom as above, and attempt to prove or disprove that the axiom is satisfied by the models.

Aside. We have not proved the correctness of the translation from BON constraints to PVS axioms and types. 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. **End of Aside.**

This is 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 next subsection briefly suggests how the BON CASE tool might support this consistency checking.

4.1 Implementation and algorithms

A sketch of an implementation for consistency checking of collaboration diagrams and class diagrams is as follows. First, assume that rules (1), (2), and (3) above, have been checked – this is straightforward and can easily be implemented in the CASE tool framework of [9] (in fact, most of these rules have already been implemented). Rule (4) is to be checked, informally, as follows: convert the collaboration diagram into an annotated finite state machine (FSM), following this algorithm. Given a user-supplied initial state (specified as an assignment of values to entities), simulate the finite state machine. Each state in the machine represents the execution of a routine; a transition represents the termination of one message and the commencement of the next. On entry to the state, the precondition of the routine is checked; if it is satisfied, simulation continues, otherwise it halts and feedback is provided. On exiting the state, the specification of the routine (i.e., $routine.spec$, as above) is added to a *constraint store*. This constraint store might be a set of conjectures in PVS. One might envision PVS running in the background, discharging obligations as they are generated by the simulation. An alternative to using PVS would be to consider a constraint store akin to that used in constraint programming. A constraint solver could then be applied as each new condition is added.

To complete the implementation, we must indicate how the "next state" in the state machine is selected. This is done according to sequence number. So, after executing in state $a.f$ (representing the call $a.f$) which has sequence number n, the next state will be the one reachable with sequence number $n + 1$. This, of course, assumes that the sequence numbers are contiguous.

Producing a FSM from a collaboration diagram is straightforward and follows the approach of [3]. We produce a FSM because it is a simple computational model and it is easy to implement; it is also sufficient for simulating BON collaboration diagrams. Much of the complexity of the translation in [3] arises from UML's sequence diagrams (including concepts such as return calls, exception handling, and nesting). These problems do not in general arise in BON collaboration diagrams. The above algorithm is currently being implemented in the BON CASE tool of [9].

4.2 Checking Collaboration Diagrams against Test Drivers

We have shown how to directly check collaboration diagrams with class diagrams for consistency. When developing software using the methodology of [13], it is possible that developers will have constructed test drivers before models. These can be used not only for the usual testing purposes but also in the consistency checking process, as sketched in Section 3: consistency of class diagrams and collaboration diagrams can be checked transitively via checking collaboration diagrams against test drivers. We briefly sketch how this might be done.

Assume that we have written a test driver, e.g., in Eiffel. A test driver is a class with a method that creates objects and generates a sequence of messages. An example might be as follows.

```
class TEST_DRIVER creation make
feature r : REPOSITORY; s : SET; b : BUYER; a : ADDRESS;
  make is do
    create r; create s; create b; create a;
    r.get_leads; s.get_next_buyer; b.get_address; a.print
  end
end -- TEST_DRIVER
```

We want to determine if the sequence of calls appearing in the test driver is an implementation of the calls appearing in the collaboration diagram. Unfortunately, we cannot just take the FSM constructed in the algorithm of Section 4.1 and simulate it directly. This is insufficient because the test driver may make calls to routines that do not appear in the collaboration diagram (i.e., implementation details). Instead, we take a metamodelling approach. We first specify the concept of a test driver, and then describe the consistency constraints that test drivers and collaboration diagrams must obey. We only sketch the approach here; details can be found in [14].

A test driver td consists of program text, a set of object creation statements, and a sequence of routine calls. The driver must obey the following constraints.

1. For each entity used in a test driver, there is a suitable instantiation and creation statement in the program text.
2. If a routine call i precedes a routine call j in the sequence of test driver calls, then the call corresponding to i in the program text precedes the call for j in the program text.

We can now define consistency for test drivers and collaboration diagrams as follows. Assume that we have a metamodel class $TEST\_DRIVER$. (Note that test drivers are not part of the extended model). An additional routine is needed to check that the test driver is consistent with collaboration diagrams. This routine, $cw\_testdriver$, is as follows.

$$
\begin{array}{l}
cw\_testdriver(td : TEST\_DRIVER) : BOOLEAN \\
\textbf{require } td \neq Void \\
\textbf{ensure } Result = td.occurs.includes(collab\_diagram.occurs) \wedge \\
\qquad\qquad collab\_diagram.calls.subseq(td.calls)
\end{array}
$$

Thus a test driver td and an extended model xm are consistent iff

$$xm.cw\_testdriver(td)$$

The first conjunct of the postcondition of the routine above states that the entities occurring in the collaboration diagram must be included in the entities arising in the test driver. The second conjunct states that the sequence of calls appearing in the collaboration diagram must be a subsequence of the calls appearing in the test driver. We note the similarity between this specification and CSP's "hide" or "interface" construct.

The function $subseq$ is defined as follows. Informally, it returns $true$ if its argument is a not necessarily contiguous subsequence of the target.

$$
\begin{array}{l}
subseq(t : SEQUENCE[G]) : BOOLEAN \\
\textbf{require } true \\
\textbf{ensure} \\
\quad size = 0 \rightarrow Result = true; \\
\quad size > 0 \wedge t.size = 0 \rightarrow Result = false; \\
\quad size = 1 \wedge t.size > 1 \rightarrow Result = t.includes(item(1)); \\
\quad size > 1 \wedge t.size = 1 \rightarrow Result = false; \\
\quad size > 1 \wedge t.size > 1 \rightarrow \\
\qquad Result = (t.includes(s.item(1)) \wedge tail.subseq(t))
\end{array}
$$

An implementation of $cw\_testdriver$, in terms of abstract syntax trees and direct simulation of the collaboration diagram, is outlined in [14].

5 Related Work and Conclusions

The introduction of the UML has spurred much recent research on consistency checking, but the topic has been of past interest and study. Zave and Jackson [18] presented a framework for composing specifications via conjunction, with the aim of supporting multi-paradigm specification. In their approach, specifications are transformed in to a common semantic domain (in [18], 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 checking as straightforward as possible to carry out. The authors' goal is not specifically consistency checking, 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.

Finkelstein et al. [6] 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.

The ADORA project [4] 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 [12]. The integrated model allows consistency constraints to be defined between views. A language and tool for supporting these constraints is discussed in [15]. 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 [16] focuses specifically on consistency checking with the UML, primarily, consistency checks relating class diagrams, sequence diagrams, and state charts. In their approach, diagrams are annotated with extra information relating the separate views, and attributed graph grammars are used as a theoretical underpinning to carry out the consistency checking.

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. The basic architecture is shown in Fig. 5.

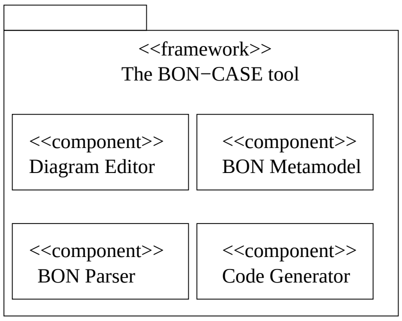

Fig. 5. Architecture of the BON CASE tool

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 [11] 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.

References

1. K. Beck. *Extreme Programming Explained*, AWL, 1999.
2. G. Booch, J. Rumbaugh, and I. Jacobson. *The UML Reference Guide*, Addison-Wesley, 1999.
3. L. Briand and Y. Labiche. A UML-Based Approach to System Testing, in *Proc. UML 2001*, LNCS 2185, Springer-Verlag, 2001.
4. M. Glinz, S. Berner, S. Joos, J. Ryser, N. Schett, and Y. Xia. The ADORA Approach to Object-Oriented Modeling of Software. In *Proc. CAiSE'01*, LNCS 2068, Springer, June 2001.
5. B. Meyer. *Eiffel - The Language*, Prentice-Hall, 1992.
6. A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specification. *IEEE Trans. Software Engineering* 20(8), August 1994.
7. OMG Consortium, UML 1.4 Documentation, 2000. Available at www.omg.org.
8. S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS System Guide 2.4*, CSL, SRI International, November 2001.
9. R.F. Paige and L. Kaminskaya. A Tool-Supported Integration of BON and JML, Technical Report CS-TR-2001-04, York University, July 2001.
10. R.F. Paige and J.S. Ostroff. Developing BON as an industrial-strength formal method. In *Proc. World Congress on Formal Methods*, LNCS 1709, Springer-Verlag, September 1999.
11. R.F. Paige and J.S. Ostroff. Metamodelling and conformance checking with PVS. In *Proc. Fundamental Aspects of Software Engineering 2001*, LNCS 2029, Springer-Verlag, April 2001.
12. R.F. Paige and J.S. Ostroff. The Single Model Principle (Extended Abstract). In *Proc. Requirements Engineering 2001*, IEEE Press, August 2001.
13. R.F Paige and J.S. Ostroff. A Proposal for a UML-Based Method for Developing Reliable Systems, in *Proc. Workshop on Precise UML-Based Methods*, GI Series 7, German Society, October 2001.
14. R.F. Paige, J.S. Ostroff, and P.J. Brooke. Checking the Consistency of Views in PVS. Technical Report TR-CS-2002-01, Department of Computer Science, York University, February 2002.
15. N. Schett. *A Notation for Integrity Constraints in* ADORA *Models - Concept and Implementation* (in German). Diplomathesis, University of Zurich, 1998.
16. A. Tsiolakis. *Semantic Analysis and Consistency Checking of UML Sequence Diagrams.* Diplomarbeit, TU-Berlin, TR 2001-06, April 2001.
17. K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Architecture*, Prentice-Hall, 1995.
18. P. Zave and M. Jackson. Conjunction as Composition, *ACM Transactions on Software Engineering and Methodology* 2(4), October 1993.