
# Checking the Consistency of Views using PVS

Richard F. Paige, Jonathan S. Ostroff, and Phillip J. Brooke

Technical Report CS-2002-01

February 18, 2002

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

# Checking the Consistency of Views 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.** A method, based on BON [17], for building reliable object-oriented software systems was proposed in [14]. It combined use of modelling and Extreme Programming [1] (XP) practices, emphasizing the use of a limited set of views of a software system, with consistency rules and automatic generation tools defined between the views. This paper builds upon this framework and formally specifies the consistency constraints between the two BON views: the static view provided by class diagrams, and the dynamic view provided by collaboration diagrams. The constraints are specified as an extension of the BON metamodel from [12], and are implemented in PVS. They ensure that the sequence of messages appearing in the dynamic view are permitted, given the contracts appearing in the static view. A sketch of how the constraints might be implemented in a BON CASE tool is also provided.
*A revised version of this paper was published in Proc. Fourth Workshop on Rigorous Object-Oriented Methods (ROOM4), London, UK, Springer, March 2002.*

## 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 [7] 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 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. Unfortunately, UML does not provide a full set of rules or tools to check consistency. Many consistency constraints are specified in the UML metamodel [8] and certain UML-compliant CASE tools implement many of these. However, some of the complex constraints, which involve information scattered across very different views of the software system, are not implemented in any tool, and thus developers must rely on their own expertise to identify and resolve inconsistencies.

The 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 OMT, Fusion, and BON [17].

The aim of this paper is to formally specify and describe implementations of consistency constraints between BON views – particularly, but not exclusively, class diagrams and collaboration diagrams – that can be produced during software development. These rules will be used in the context of a proposed methodology for building reliable systems [14]. 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, or 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 will be specified as an extension of the metamodel of BON presented first in [12]. As such, the constraints will be specified in PVS [9], so that theorem proving technology can be exploited both in checking the consistency of views, and in validating the specifications of the constraints. The intent is to use these specifications, and their implementations, in the construction of a CASE tool for BON that supports consistency checking and also consistent views by construction.

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 UML and other modelling languages that support these two views.

The methodology in which the consistency rules are to be specified integrates OO modelling and XP; thus, a programming language is required with which to describe test drivers. We use Eiffel [6] for this purpose. An advantage of Eiffel is that it offers built-in support for specifying contracts (pre- and postconditions), which can assist in the testing process. However, the rules that we develop and present in this paper are not Eiffel-dependent; they can be applied equally well to Java, C++, and other OO programming languages as well.

### 1.1 Overview

We commence with a brief overview of the methodology of [14], focussing on the notations used and the manner in which XP is integrated with modelling. We describe BON very briefly, and present an example of each type of BON diagram. We then specify the notation used throughout the paper and define the relation $cw$,"consistent-with", which

can be applied to development work products. Then we explain consistency checking of class diagrams against collaboration diagrams, specify well-formedness rules in BON and in PVS, and outline an implementation. We do the same for checking test drivers against collaboration diagrams. This is the key link between modelling and XP test drivers, which are interpreted as implementations or refinements of collaboration diagrams. We then briefly explain how the constraints are being implemented in our research prototype.

## 2 Background

We briefly summarize the methodology that integrates OO modelling and XP from [14], since the methodology provides the motivation for constructing the consistency constraints. Then, we outline BON and Eiffel, focussing 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.

### 2.1 A Methodology integrating XP and object-oriented modelling

A methodology that follows and supports the principles and practices of XP, while still allowing OO modelling to be used, was presented in [14]. 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 to allow 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.

Parts of the methodology are summarized in Fig. 1. The diagram uses UML's package and dependency notations to illustrate the work products delivered by the methodology, and their relationships.

A key element of Fig. 1 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 [14]. In this paper, we focus on the relationships involving collaboration diagrams and class diagrams. Specifically, we will describe how consistency between these work products can be established, directly via use of a metamodel and tools such as PVS, and indirectly via use of test drivers.

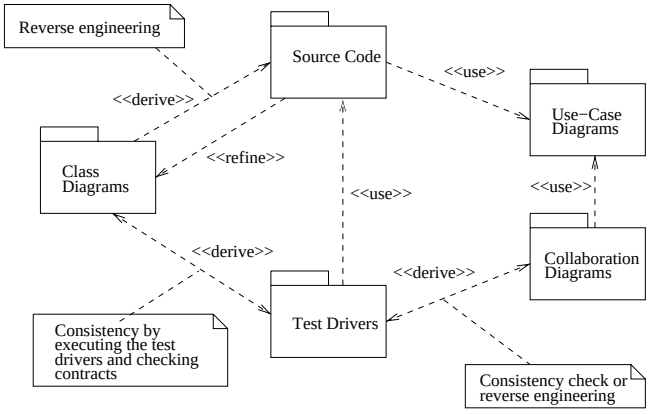


**Fig. 1.** Work products and their relationships

With this restricted set of work products, and with the relationships defined in Fig. 1, developers can follow the principles of XP where needed, and can also apply modelling where desired as well. Further, the tools and process supplied with the methodology can support the developers in switching between work products when needed.

### 2.2 BON and Eiffel

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. 2 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 [12].

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.

BON also provides notation for collaboration diagrams, showing the communication between objects. Fig. 3 shows an example. Numbers that annotate messages are

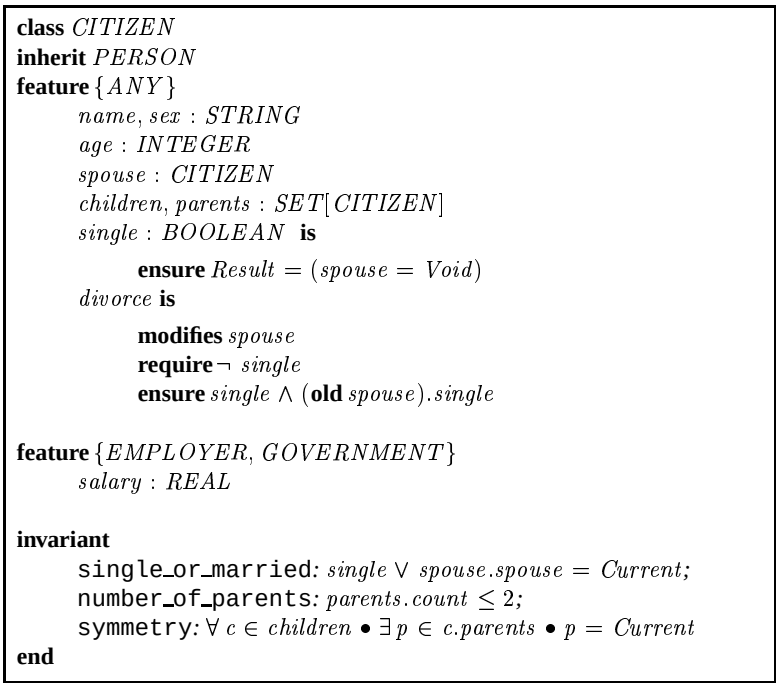


**Fig. 2.** Class *CITIZEN*

cross-referenced to a scenario box, detailing the purpose of the message. Messages in dynamic diagrams correspond to feature calls.

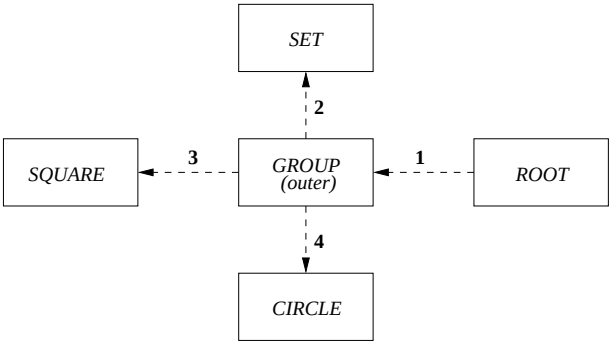


**Fig. 3.** BON collaboration diagram

Eiffel is a pure OO programming language with some similarities to Java. It emphasizes the use of design-by-contract, via the specification of pre- and postconditions for class features, and invariants for classes. It is syntactically very similar to the BON textual notation shown in Fig. 2, except that features can include bodies, and its assertion language is executable and supports only propositions, and not quantifiers (though the latest Eiffel implementation supports a notion of *agent* that increases the expressiveness of the executable assertion language substantially). Eiffel is semantically compatible with BON, in that BON models can be seamlessly mapped to Eiffel programs.

## 3 Notation and Foundations

The eventual goal of our work is to be able to check any *work product*, $WP$, against any other work product for consistency. A work product, in the context of the methodology of [14], 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 *"consistent-with"* relation $cw : WP \leftrightarrow WP$, which obeys several important properties that relate to the methodology of [14] (we assume that $cw$ is an infix relation):

- symmetry, i.e., $wp1 \ cw \ wp2 = wp2 \ cw \ wp1$.
- transitivity (which, as we discuss below, is useful for methodological support).

– reflexivity, i.e., a work product is always consistent with itself.

Transitivity of $cw$ is very useful, as the following illustrates. Suppose that $CD$, $DD$, $TD$, and $SC$ are work products representing, respectively, a class diagram, collaboration diagram, test driver, and source code. Suppose as well that we want to show that

$$CD \ cw \ DD$$

(i.e., that a class diagram is consistent with a collaboration diagram). One approach to this is:

$$(CD \ cw \ SC) \tag{1}$$
$$(DD \ cw \ TD) \tag{2}$$
$$(SC \ cw \ TD) \tag{3}$$

If $cw$ is transitive, then

$$(1) \wedge (2) \wedge (3) \rightarrow (CD \ cw \ DD)$$

It is straightforward to guarantee that a class diagram and source code are consistent (equation (1)); either forward engineer code from diagrams, or reverse engineer diagrams from code. Alternatively, an abstract syntax tree can be built for the code and it can be traversed, checking components against elements in the graph underpinning the class diagram. Similarly, checking source code against a test driver, equation (3), is also easy: compile the test driver and execute it. If it compiles, executes, and all tests are accepted, the two products are consistent. Equation (2), checking collaboration diagrams against test drivers, is more complicated, and will be described in the sequel; a sketch of an algorithm was presented in [14].

Thus, one approach to consistency checking of collaboration diagrams and class diagrams is by exploiting the transitive properties of the $cw$ relationship and by making use of test drivers as an intermediate product. But it is also desirable 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 (they may not, for example, be following the principles and guidelines of XP). In the next section, we specify consistency constraints for 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 a BON collaboration diagram against a set of 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 permission 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, 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).
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.

We consider these steps in order, skipping (1) since the BON CASE tool is described elsewhere [10]. The remaining constraints will be specified as an extension of the metamodel of BON first presented in [12]. Some of these constraints, e.g., those in (2), are easily implemented in a CASE tool. Since the BON CASE tool implements the BON metamodel, we thus have a way of ensuring that they are satisfied. 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.

Before specifying the rules in (2) and (3), we briefly recount the key parts of the BON metamodel from [12]. 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.


**Fig. 4.** BON metamodel, abstract view

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 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 [12] (with one extension - the notion of a *specification*). 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 [12], 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** $- ROUTINE$

$spec$ 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 in [11], 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 [12]. 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.

```
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 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 in the sequel. The formalization of specifications of a routine initially requires a new type, SPECTYPE, which is a record containing the initial and final state 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 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 [6]; it represents a packaged, indexable data structure of arbitrary but finite length. Here is an excerpt of its interface, presenting the routines that are relevant to this work. $item$ 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**
$size : INTEGER$
$item(i : INTEGER) : G$
$tail : SEQUENCE[G]$
$head : G$
$subseq(t : SEQUENCE[G]) : BOOLEAN$
$precedes(g1, g2 : G) : BOOLEAN$
**invariant** $size \geq 0$
**end** $-SEQUENCE[G]$

Now we can specify the concept of a message that appears in a collaboration diagram; messages were specified in [12], 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.

**class** $MESSAGE$ **feature**
$source, target : OBJECT$

$routine : ROUTINE$
$number : INTEGER$
**invariant** $number \geq 1$
**end** $-$ $MESSAGE$

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 [12]. 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 a collaboration diagram.

> **Aside.** BON obeys the *single model principle* [13], 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 $MODEL$, e.g., $DYNAMIC\_MODEL$, etc., as this can easily introduce inconsistencies.

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, via 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 features 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** $\{ANY\}$

$class\_diagram : EXTENDED\_MODEL$

$collab\_diagram : EXTENDED\_MODEL$

**invariant**

$msgs\_in\_rels$;

$calls\_linked\_to\_msgs$;

$objects\_in\_occurs$;

$objects\_in\_abs$;

$same\_lengths$

**end** $-EXTENDED\_MODEL$

The invariant clauses are as follows. The clause $msgs\_in\_rels$ says that each message in the $sequence$ is a relationship in the model.

$$\forall\, s \in sequence \mid \exists\, m \in rels \bullet m = s$$

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 $i$.

$$\begin{aligned} &\forall\, s1, s2 \in sequence \mid sequence.precedes(s1,\, s2) \bullet \\ &\exists\, c1, c2 \in calls \mid calls.precedes(c1,\, c2) \bullet \\ &\quad s1.routine = c1 \wedge s2.routine = c2 \end{aligned}$$

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

$$\begin{aligned} \forall\, m \in sequence \bullet \forall\, s \in m.source \bullet s \in occurs \wedge \\ \forall\, s \in m.target \bullet s \in occurs \end{aligned}$$

$objects\_in\_abs$ 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.

$$calls.length = sequence.length$$

The functions $class\_diagram$ and $collab\_diagram$ produce the two views of the model. These are defined as follows.

$$class\_diagram : EXTENDED\_MODEL$$

$$\textbf{ensure } Result.abs = \{a \in abs \mid a : STATIC\_ABS\};$$
$$Result.rels = \{r \in rels \mid r : STATIC\_REL\}$$

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

To express $EXTENDED\_MODEL$ in PVS, we could introduce a new subtype (representing the type of extended models), and then could map each routine of the BON class to a PVS function. 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 that with 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, e.g., 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)) })
```

We can now formally define the "consistent-with" relationship. Suppose $xm : EXTENDED\_MODEL$, and let $xm.cd$ and $xm.dd$ be short-hands for $xm.class\_diagram$ and $xm.collab\_diagram$, respectively. The formal definition of $xm.cd \ cw \ xm.dd$ is a number of constraints that must be contained in the invariant of $EXTENDED\_MODEL$. For each constraint, a PVS formulation is provided when it cannot be found in [12].

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

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

(Note that $m.cd.abs$, defined in [12], 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.

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

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 [12]. The constraint in [12] 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. Constraint (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.

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

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. In PVS, this is specified 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)))
```

What about calls after the first? For a call $i \geq 2$ to be enabled, the preceding 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:

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

(Recall that the definition of sequential composition is the following:

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

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 the above constraint in PVS is challenging. The problem 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.

```
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.

```
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)))))
```

Effectively, `convert` states that each element of the converted sequence is identical to the corresponding message's routine details. 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
     (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.

This is a specification of the consistency relation $\underline{cw}$ 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 specification of $cw$ above. The next subsection briefly suggests how the BON CASE tool might support this consistency checking.

### 4.1 Implementation and algorithms

An implementation of consistency checking of collaboration diagrams and class diagrams follows the following process. 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 [10] (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 $n + 1$.

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 [10].

## 5 Checking Collaboration Diagrams against Test Drivers

The approach described in the previous section shows how to check a collaboration diagram against a class diagram for consistency. When developing software using the methodology of [14], it is possible that developers will have constructed test drivers. These can be used not only for the usual testing purposes but also in the consistency checking process, as discussed in Section 3: consistency of class diagrams and collaboration diagrams can be tested indirectly via checking collaboration diagrams against test drivers. In this section we summarize how this latter check can be specified and implemented. To do so, we define the $cw$ relationship for test drivers and collaboration diagrams. We do this using the infrastructure set up for direct checking of collaboration diagrams and class diagrams. Consider the following test driver, written in Eiffel.

```
class TEST_DRIVER
creation make
feature r : REPOSITORY; s : SET; b : BUYER; a : ADDRESS;
feature
  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 previous section and simulate it directly as follows: on entry to each state, execute the next routine call appearing in the test driver. This is insufficient because the test driver may make calls to routines that do not

appear in the collaboration diagram (i.e., implementation details). Thus, we consider the following approach.

Define a test driver using a BON class as follows. The $occurs$ set describes the names of all objects appearing in **create** statements in the test driver, $calls$ is the sequence of routine calls appearing in the source text, and $source\_text$ is the sequence of strings that make up the body of the test driver.

$$
\begin{array}{l}
\textbf{class } TEST\_DRIVER \textbf{ feature} \\
\quad occurs : SET[OBJECT] \\
\quad calls : SEQUENCE[ROUTINE] \\
\quad source\_text : SEQUENCE[STRING] \\
\textbf{end } - TEST\_DRIVER
\end{array}
$$

The following well-formedness constraints apply for all test drivers; they are constraints that are part of the invariant for $TEST\_DRIVER$.

1. For each entity occurring in a test driver, there is a suitable instantiation and creation statement.

$$
\begin{array}{l}
\forall\, o \in occurs \bullet \exists\, i : 1, .. source\_text.length \bullet \\
\quad source\_text.item(i).includes(o.name) \wedge source\_text.item(i).includes(\text{“}create\text{”})
\end{array}
$$

2. If a message $i$ precedes a message $j$ in the sequence of test driver messages, then the routine call corresponding to $i$ in the source text precedes the routine call for $j$.

$$
\begin{array}{l}
\forall\, i, j : 1, .., calls.length \mid i < j \bullet \\
\quad \exists\, u, v : 1, .., source\_text.length \mid u < v \bullet \\
\qquad source\_text.item(u).includes(calls.item(i).text) \wedge \\
\qquad source\_text.item(v).includes(calls.item(j).text)
\end{array}
$$

(where a call $r.text$ for a routine $r$ produces a string representation of the routine call). Note that we do not have to worry about loops or selections in the test driver source text; all we have to concern ourselves with are calls. Consider the following example fragment of a test driver.

$$
\begin{array}{l}
\textbf{class } SIMPLE\_TEST\_DRIVER \textbf{ feature} \\
\ldots \\
make \textbf{ is do} \\
\quad \ldots \\
\quad \textbf{from } i := number\_of\_test\_runs; \\
\quad \textbf{until } i = 0; \\
\quad \textbf{loop}
\end{array}
$$

$contents.test1(i);\ contents.test2(i);\ frame.test3(i);$
$i := i - 1;$
**end**
**end**
**end** $-\ SIMPLE\_TEST\_DRIVER$

This test driver can generate a fixed number of messages sent, in order, to $contents$ and $frame$. The sequence of routine calls associated with this test driver will contain $number\_of\_test\_runs$ concatenations of the subsequence

$$< contents.test1, contents.test2, frame.test3 >$$

However, this repetition will not be captured in the collaboration diagram – it will show the subsequence once. Thus, its sequence of routine calls will contain the above subsequence exactly once, and the test driver is therefore an implementation of the collaboration diagram.
We can extract the calls very simply from the source text as follows. First, construct an abstract syntax tree (parse tree) from the test driver. Then, using an in-order traversal of the parse tree, concatenate all targeted routine calls that are encountered. This produces the sequence of calls.

We can now define $cw$ for test drivers and collaboration diagrams as follows. First, we declare the following variables.

$$dd : EXTENDED\_MODEL$$
$$td : TEST\_DRIVER$$

$dd\ cw\ td$ holds if and only if

$$td.occurs.includes(dd.occurs) \wedge dd.calls.subseq(td.calls)$$

The first conjunct 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. Here are some examples:

$$(< 1, 2, 4, 9, 16, 3, 8 >).subseq(< 1, 2, 3 >) = true$$
$$(< 1, 2, 3 >).subseq(< 1, 2 >) = true$$
$$(< 1, 2, 7, 9, 12 >).subseq(< 1, 2, 4 >) = false$$
$$(< 3, 5, 7, 9 >).subseq(<>) = true$$

The empty sequence, $<>$, is a subsequence of any other sequence. For our problem domain, this implies that the empty collaboration diagram (consisting of no messages)

is implemented by any test driver. This is consistent with the usual definitions of refinement. Note that the dual does not hold – an empty test driver (consisting of no routine calls) does not implement any collaboration diagram, except the empty one. Here is the formal specification of $subseq$.

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

$subseq$ makes use of the simple routine $includes$ which is as follows.

$$
\begin{array}{l}
includes(x : G) : BOOLEAN \\
\textbf{require } true \\
\textbf{ensure} \\
\quad size = 0 \rightarrow Result = false; \\
\quad size > 0 \rightarrow Result = ((x = head) \vee includes(a, tail))
\end{array}
$$

The implementation of $cw$ for test drivers and collaboration diagrams will construct the sequences corresponding to the routine calls appearing in the test driver, as well as those appearing in the messages of the collaboration diagram, and will check to see if the dynamic diagram sequence is a subsequence of the test driver sequence. It is inefficient to generate both sequences of routine calls and then test one against the other. Instead, we can carry out this check as follows. The dynamic diagram drives the implementation of the consistency check. As was done for testing class diagrams against collaboration diagrams, a finite state machine is constructed from the collaboration diagram (to recap: transitions of the machine are routine calls, and a state represents the execution of the body of a routine). Simulation of the diagram starts from a user-provided initial state, indicating initial values for variables. This could be extracted automatically from the test driver, e.g., by looking at assignment statements in creation routines, or by using the postcondition of a creation routine. The test driver is then parsed and an abstract syntax tree (AST) constructed and the root is marked. The simulation of the state machine begins. On entry to a state $a.f$ (representing the execution of a routine call $a.f$), the AST is traversed depth-first starting from the last mark. If a routine call $a.f$ is found in the AST, then the corresponding node is marked, and simulation of the state machine continues with the next transition according to sequence number. Other-

wise, simulation halts and a suitable message is displayed indicating at what point the test driver fails to implement the collaboration diagram.

## 6 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. [7] 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 [13]. 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] focusses 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.

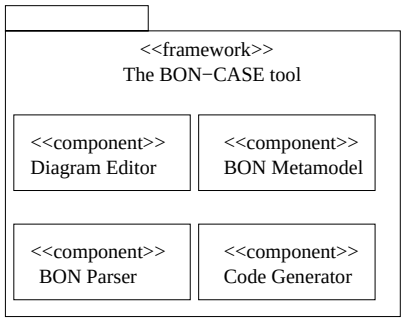


**Fig. 5.** Architecture of the BON CASE tool

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 [12] 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 be very useful for consistency checking of test drivers against collaboratin 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. E.C.R. Hehner. *A Practical Theory of Programming*, Springer-Verlag, 1993.
6. B. Meyer. *Eiffel The Language*, Prentice-Hall, 1992.
7. 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.
8. OMG Consortium, UML 1.4 Documentation, 2000. Available at www.omg.org.
9. S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS System Guide 2.3*, CSL, SRI International, September 1999.
10. R.F. Paige and L. Kaminskaya. A Tool-Supported Integration of BON and JML. Technical Report CS-TR-2001-04, Department of Computer Science, York University, July 2001.

11. 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.
12. 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.
13. R.F. Paige and J.S. Ostroff. The Single Model Principle. In *Proc. Requirements Engineering 2001*, IEEE Press, August 2001.
14. 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.
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.