An Introduction to PVS
Metamodelling with PVS

Richard Paige

Department of Computer Science

University of York, York, U.K.

paige@cs.york.ac.uk

and

Department of Computer Science

York University, Toronto, Canada.

PVS: What Is It?

A verification system with

- a general-purpose formal specification language, associated with a *theorem prover*, *model checker*, and related tools (browser, doc. generator).

Freely distributed by SRI, currently on v2.4

- Runs on Solaris and Linux, UI based on Emacs and Tcl/Tk
- Used in both academia and industry
- Rich specification language, powerful prover, expressive libraries, wealth of support.
- **Applications**: safety critical systems, hardware, mathematics, distributed algorithms

Overview

- Introduction to the PVS specification language
- Look-and-feel of the prover.
  - Some key prover commands.
- Several little examples.
- Using PVS for
  - meta-modelling
  - expressing object-oriented models (particularly BON)
  - conformance and consistency checking

PVS Specification Language

... is an enriched typed λ-calculus.

- If you're comfortable with functional programming, you'll be comfortable with PVS.
- Key aspects:
  - Type constructors for restricting the domain and range of operations.
  - Rich expression language.
  - Parameterized and hierarchical specification.

Types

Base types: eg., **`bool, int, nat`**

Function types, eg., **`[int -> [bool -> int]]`**

Enumeration types **`{a,b,c}`**

Tuple types **`[A,B]`**

Record types **`[#a:A, b:B #]`**

Mutually recursive data types (ADTs).

Predicate subtypes:

- **`A: TYPE = {x:B | p(x)}`**
- **`A: TYPE = (p)`**

More on Types

Lots of predefined subtypes, eg.,

```
nat: TYPE = { n:int | n >=0 }
subrange(n,m:int): TYPE =
  { i:int | n<=i & i<=m }
```

Dependent types allow later types to depend on earlier ones.

```
date:TYPE =
  [# month:subrange(1,12),
     day:subrange(1,num_of_days(month))
  #]
```

Predicate subtypes are used to constrain domain/range of operations and to define partial functions.

Expressions

- Higher-order logic **(&, OR, =>, .., FORALL, EXISTS)**
- Conditionals
  - **IF c THEN e1 ELSE e2 ENDIF**
  - **COND c1->e1, c2->e2, c3->e3 ENDCOND**
- Record overriding
  - **id WITH [(0):=42,(1):=12]**
- Recursive functions

```
fac(n:nat): RECURSIVE nat =
  IF n=0 THEN 1 ELSE n*fac(n-1) ENDIF
  MEASURE n
```

- Inductive definitions, tables

Type Correctness Conditions (TCCs)

- PVS must check that the expressions that you write are well-typed.

```
fac(n:nat): RECURSIVE nat =
  IF n=0 THEN 1 ELSE n*fac(n-1) ENDIF
MEASURE n
```

Function **`fac`** is well-typed if

- **`n/=0 => n-1>=0`** (the argument is a nat)
- **`n/=0 => n-1<n`** (termination).

The type checker (M-x tc) generates type correctness conditions (TCCs)

Example TCCs for factorial

```
fac_TCC1: OBLIGATION
  FORALL (n:nat): n/=0 => n-1 >= 0

fac_TCC2: OBLIGATION
  FORALL (n:nat): n/=0 => n-1 < n
```

TCCs (Continued)

Expressions are only considered to be well-typed after all TCCs have been proven.

- Type checking in PVS is ***undecidable*** (because of predicate subtypes).

+ The PVS prover will automatically discharge most TCCs that crop up in practice.

Why aren't there more TCCs in preceding, eg., for **`n*fac(n-1)`** of type **`nat`**?

Suppressing TCC Generation

The type checker “knows” that

```
JUDGEMENT *(i,j) HAS_TYPE nat
JUDGEMENT 1 HAS_TYPE posint
```

Judgements are a means for controlling the generation of TCCs.

Inference is carried out behind-the-scenes.

Judgements can be arbitrarily complex and useful.

```
JUDGEMENT inverse(f:(bijective?[D,R]))
  HAS_TYPE (bijective?[R,D])
JUDGEMENT union(a:(nonempty?), b:set)
  HAS_TYPE (nonempty?)
```

Theories

- Specifications are built from *theories.*
- Declarations introduce types, variables, constants, formulae, etc.

```
 div: THEORY      % natural division
  BEGIN
    posnat: TYPE = { n:nat | n>0 }
   a: VAR nat; b: VAR posnat
    below(b): TYPE = { n:nat | n<b }
   div(a,b): [ nat, below(b) ] % tuple

   divchar: AXIOM
    LET (q,r) = div(a,b) IN a=q*b+r
END div
```

Theories (II)

- Theories may be parametric in types, constants, and functions.

```
wf_induction[T:TYPE,<:(well_founded?[T])]: THEORY
```

- Theories are hierarchical and can import others.

```
IMPORTING wf_induction[nat, <]
```

- The built-in prelude and loadable libraries provide standard specs and proven facts for a large number of theories.

Example: Division Algorithm

```
euclid: THEORY
BEGIN
  div(a:nat, b:nat): RECURSIVE [nat,below(b)] =
    IF a<b THEN (0,a)
    ELSE LET (q,r)=div(a-b,b) IN (q+1,r)
    ENDIF
    MEASURE a
END euclid
```

- Type checking (M-x tcp) yields two TCCs

```
% proved - complete
div_TCC1: OBLIGATION FORALL (a,b:nat)
    a>=b IMPLIES a-b>=0;
% unfinished
div_TCC2: OBLIGATION FORALL (a,b:nat)
    a>=b IMPLIES a-b<a;
```

Division Algorithm (Corrected)

```
euclid: THEORY
BEGIN
  div(a:nat, b:posnat): RECURSIVE
   [nat,below(b)] =
    IF a<b THEN (0,a)
    ELSE LET (q,r)=div(a-b,b) IN (q+1,r)
    ENDIF
    MEASURE a
END euclid
```

- Type checking yields

```
     2 TCCs, 2 proved, 0 unproved
```

which does not necessarily mean div is correct!

Division
Alternative Specification

```
div: THEORY
BEGIN
  a: VAR nat; b: VAR posnat; q: VAR nat
  rem(a,b,q): TYPE =
    { r:below(b) | a=q*b+r }
  div(a,b): RECURSIVE
    [# q:nat, r: rem(a,b,q) #] =
    IF a<b THEN
      (# q:=0, r:=a #)
    ELSE
      LET rec=div(a-b,b) IN
        (# q:=rec’q+1, r:=rec’r #)
    ENDIF
    MEASURE a
END div
```

Division TCCs

```
div_TCC1: OBLIGATION
  FORALL (a,b): a<b IMPLIES a<b AND a=a

div_TCC2: OBLIGATION
  FORALL (a,b): a>=b IMPLIES a-b >= 0

div_TCC3: OBLIGATION
  FORALL (a,b): a>=b IMPLIES a-b<a
```

- All TCCs are proved automatically by the typechecker.

Animation

- Instead of doing full verification, functions can be validated in PVS via execution:
  - M-x pvs-ground-evaluator

```
<GndEval> "div(234565123,23123543)"
; cpu time (total)  0 msec user, 0 msec system
==>
  (# q:=101, r:= 10167280 #)
```

- **Question**: is this useful in metamodel validation?

Design Elements in the PVS Prover

- Heuristic automation for “obvious” cases.
- Leave the human free to concentrate on and direct steps that require real insight.
- Sequent calculus presentation

```
{-1}      A
{-2}      B
[-3]      C
|-----------------
[1]       S
{2}       T
```

- Intuitive interpretation: **`A & B & C => S OR T`**
- PVS maintains proof tree of sequents.

Interaction

- Basic tactics exist to manipulate these sequents.
- Propositional rules
  - **`(flatten)`, `(split)`, `(lift-if)`**
- Quantifier rules
  - **`(skolem)`, `(inst)`**
- Tactic language **`(try)`, `(then)`, `(repeat)`** for defining higher-level proof strategies.

```
(defstep prop ()
  (try (flatten) (prop) (try (split) (prop)
  (skip))) ...)
```

Automation

- Automate (almost) everything that is decidable!
- Propositional calculus **`(prop), (bddsimp)`**
- Equality reasoning with uninterpreted function symbols

`x=y & f(f(f(x))) = f(x) => f(f(f(f(f(y))))) = f(x)`

- Model checking **`(model-check)`**
- Automated instantiation and skolemization **`(skosimp)`**
- Workhorse: **`(grind)`**
  - combination of simplifications, rewriting, propositional reasoning, decision procedures, quantifier reasoning.
- Induction strategies.

Prover Infrastructure

- Browsing facilities locate and display definitions and find formulae that reference a name.
- Proof replay, stepping, editing.
- Graphical display of proof trees.
- Lemmas can be proved in any order.
- Introduce/modify lemmas on the fly.
- Proof chain analysis keeps you honest!

Metamodelling

- A modelling language (eg., BON, UML, OCL) consists of
  - a notation (syntax and presentation style)
  - a metamodel: well-formedness constraints
- A metamodel captures the rules that “good” (well-formed) models in the language must obey.
- **Examples:**
  - Associations are directed between from a class or cluster to a class or cluster.
  - Classes cannot inherit from themselves.

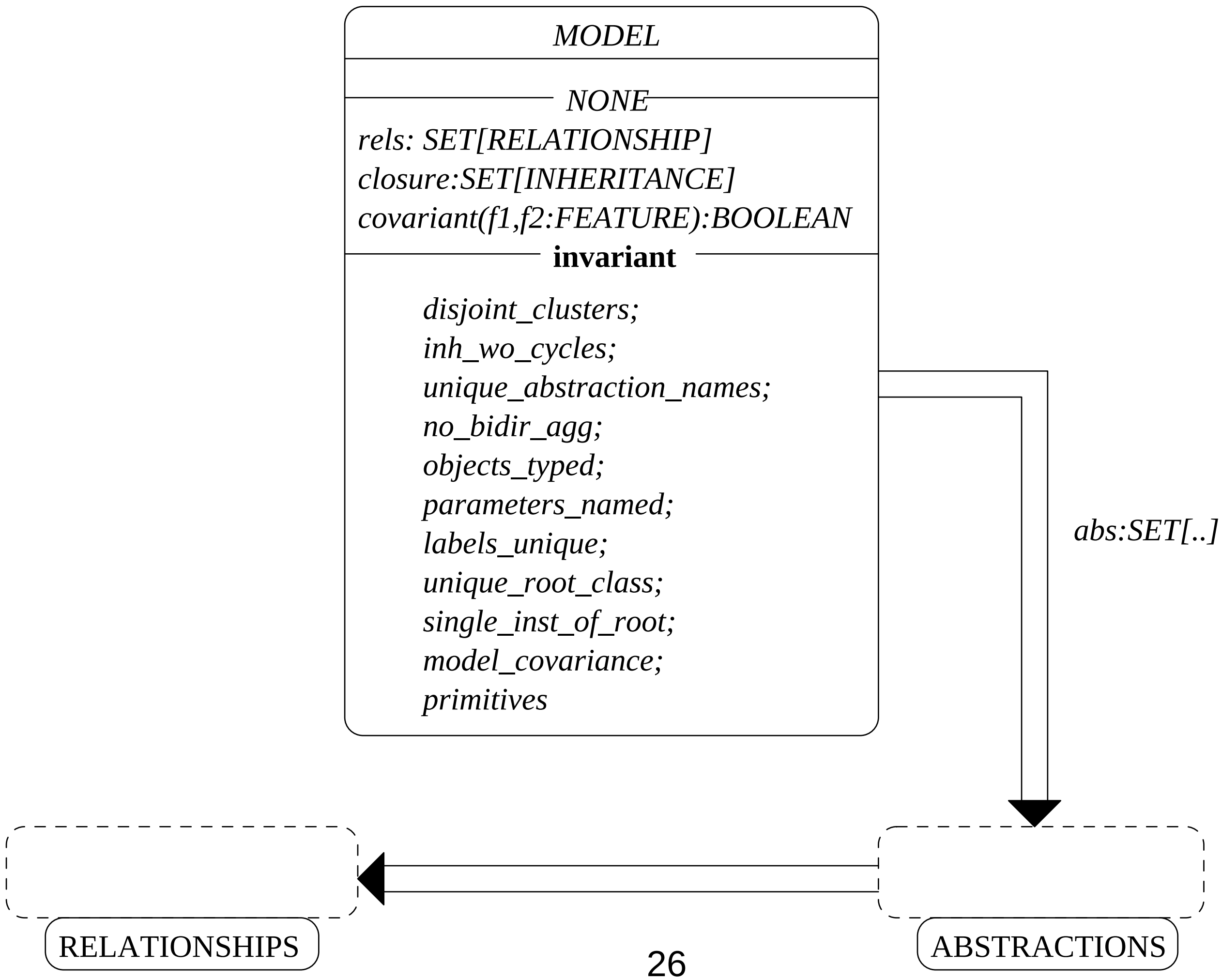
Metamodelling

- Distinction between well-formedness rules (semantic/contextual analysis) and syntactic rules (grammar/tokens) is fuzzy.
  - 2uworks.org RFP for UML 2.0 includes both abstract syntax and contextual analysis rules in metamodel.
- If a metamodel is viewed as a specification to be given to tool builders, then this is not unreasonable.
  - ...but it can make your metamodel **much** larger and thus in need of better structuring mechanisms.

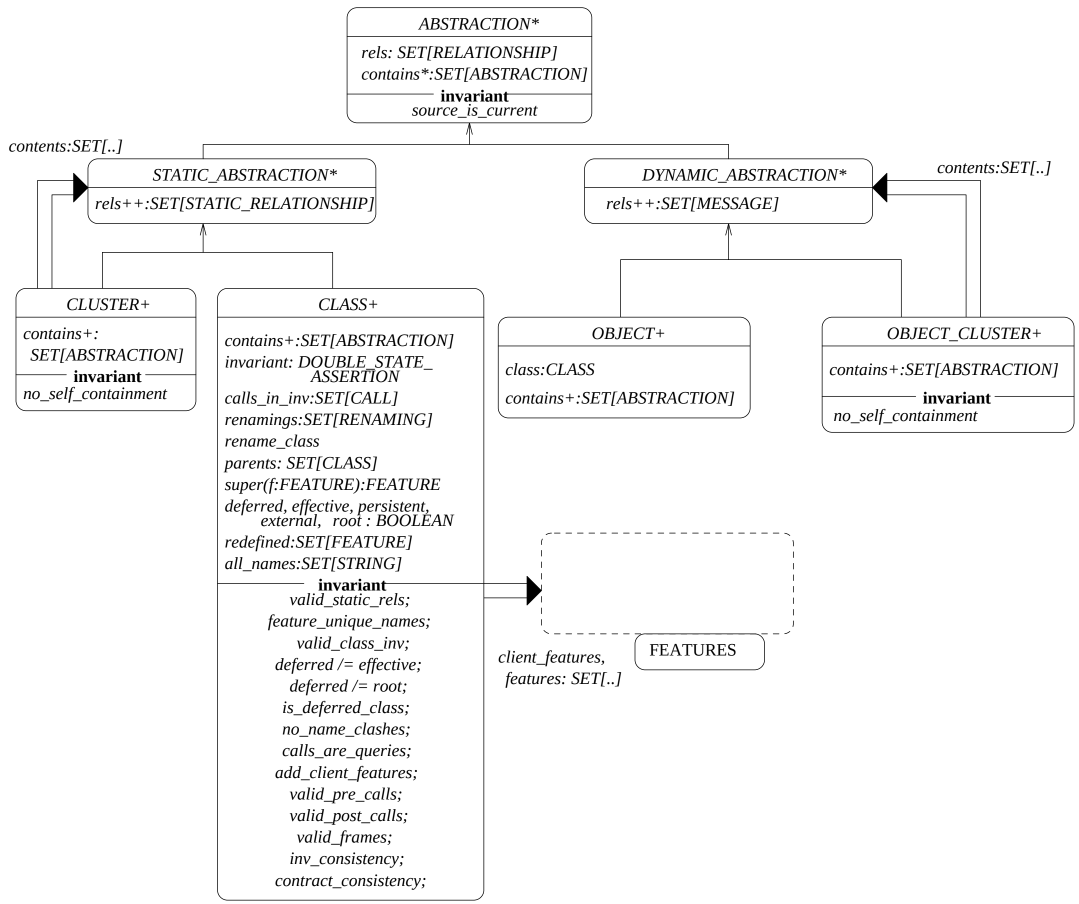
Metamodelling with PVS

- Using a tool like PVS to express a metamodel has a number of benefits:
  - Machine-checkable syntax.
  - Type checker.
  - Prover can be used to validate metamodel.
  - Ground evaluator can be used for testing.
  - Built-in theories can simplify the process of expressing the metamodel.
- But metamodels are usually expressed in OO languages ... and PVS is not OO!

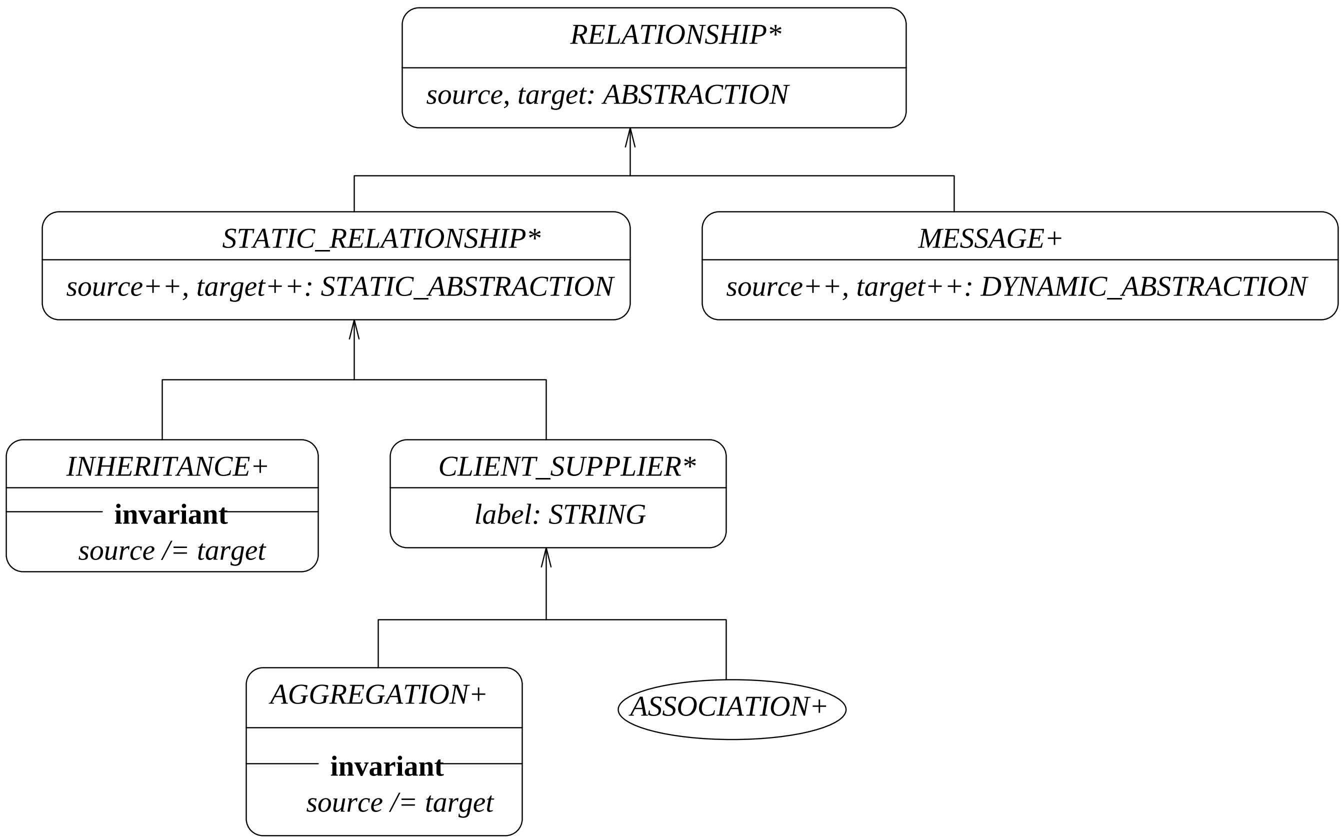
Typical Metamodel for BON



Abstractions Cluster



Relationships Cluster



Expressing the BON Metamodel in PVS

- Easiest approach: map the BON specification of the metamodel directly into PVS.
- Key questions to answer:
  - How to represent classes and objects in PVS?
  - How to represent client-supplier and inheritance?
  - How to represent the class invariants?
  - How to represent clusters?
  - How to represent features of classes?
- Answering such questions will let us represent not only the BON metamodel in PVS, but BON models as well!
- **Question**: how does an instantiated metamodel compare with a model in PVS for reasoning?

Basic Approach

- Specify class hierarchies as PVS types and subtypes.

```
ABSTRACTION: TYPE+
STATICABS, DYNABS: TYPE+ FROM ABSTRACTION
CLUSTER, CLASS: TYPE+ FROM STATICABS

OBJECT, OBJECTCLUSTER: TYPE+ FROM DYNABS

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

- Features of BON classes become functions:

```
deferred_class: [ CLASS -> bool]
class_features: [ CLASS -> set[FEATURE] ]
feature_frame:  [ FEATURE -> set[QUERY] ]
```

What is a BON Model?

- A BON model, in PVS, is just a record.

```
MODEL: TYPE+ =
  [# abst:set[ABS], rels: set[REL] #]
```

- Note that all abstractions (static and dynamic) are combined into one set.
- Projections from this to produce different views.

Clusters and Invariants

- Note that the BON metamodel has a number of clusters (Abstractions and Relationships).
- These are mapped to PVS theories.
  - *Is there any need to parameterize these theories?*
- What about the invariant clauses of classes in the metamodel?
- These can be mapped to PVS axioms.
  - In general, we'd like to avoid axioms when possible since they can introduce inconsistency.
  - Use definitions if possible.

Example Axioms

```
% Inheritance relations cannot be from an abstraction to itself.
% A class cannot be its own parent.

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

% Clusters cannot contain themselves.

no_nesting_of_clusters: AXIOM
(FORALL (cl:CLUSTER) : not member(cl,cluster_contents(cl)))

% A deferred feature cannot also be effective.

deferred_not_effective: AXIOM
(FORALL (c:CLASS): (FORALL (f:FEATURE):
  (NOT (deferred_feature(c,f) IFF effective_feature(c,f)))))
```

Example Axioms (II)

```
% All feature calls that appear in a precondition obey the
% information hiding model.

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

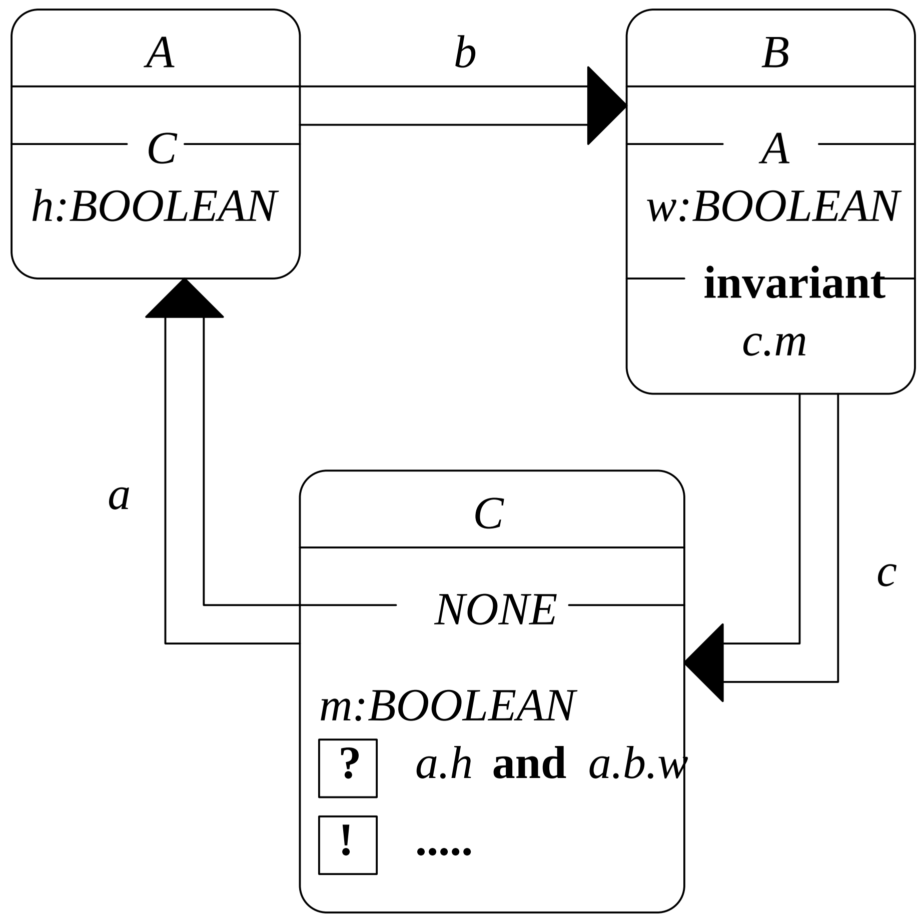
Type and Conformance Checking

- Running the type checker over the existing metamodel theories generates approximately 7 TCCs that are automatically proved.
- Earlier versions did not type check and revealed errors and omissions.
- *What can we now do with the metamodel?*
  - Conformance checking
  - Extension to view consistency checking.

Conformance Checking

- *Does a BON model satisfy the metamodel constraints?*
  - In practice this is implemented via a constrained GUI and by suitable algorithms (eg., no cycles in inheritance graph -> cycle detection algorithm).
  - In practice and *in general* it cannot be implemented fully automatically.
- **Approach 1:** express a BON model in PVS and check that it satisfies the axioms.
  - If it does not, counterexamples will be generated, though sometimes they will be difficult to interpret.
- **Approach 2:** express that a BON model cannot exist, and show that fails to satisfy an axiom. (Often easier.)

Example


A
C
h:BOOLEAN
b
B
A
w:BOOLEAN
invariant
c.m
a
c
C
NONE
m:BOOLEAN
? a.h and a.b.w
! .....

PVS Theory

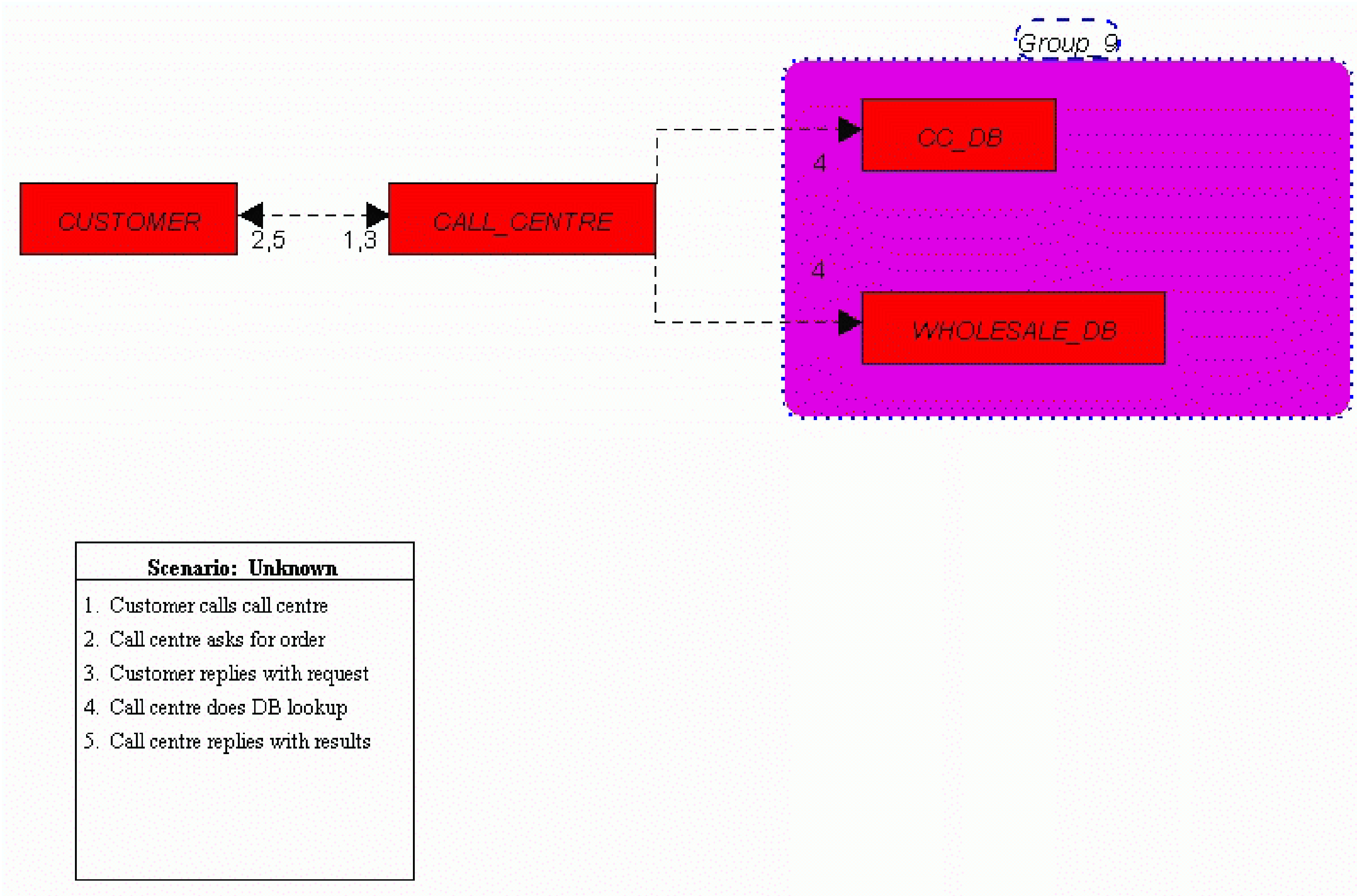
```
info2: THEORY
BEGIN
  IMPORTING metamodel
  a, b, c: VAR CLASS
  h, w, m: VAR QUERY
  ea, eb, ec: VAR ENTITY
  xm: VAR MODEL
  call1, call2, call_anon: VAR DIRECT_CALL
  call3: VAR CHAINED_CALL

  test_info_hiding: CONJECTURE
    (NOT (EXISTS (xm:MODEL): EXISTS (a,b,c: CLASS):
     EXISTS (h,w,m: QUERY): (EXISTS (ea,eb,ec:ENTITY):
     EXISTS (call1, call2, call_anon: DIRECT_CALL):
     EXISTS (call3: CHAINED_CALL):
     member(c, accessors(h)) AND member(a,accessors(w)) AND
     empty?(accessors(m)) AND call_entity(call2)=ec AND
     call_entity(call2) = ec AND call_entity(call_anon)=eb AND
     call_entity(call3) = ea AND member(call1,calls_in_pre(m)) AND
     member(call3, calls_in_pre(m)) AND
     member(call_anon,calls_in_pre(m)) AND
     member(call2, calls_in_inv(b)))))
 END info2
```

View Consistency

- BON provides two views of systems:
  - *static* (architectural) view, represented using class diagrams and contracts.
  - *dynamic* (message-passing) view, represented using collaboration diagrams
- The views may be constructed separately and thus may be inconsistent.
- Examples:
  - object in dynamic view has no class in static view
  - message in dynamic view is not enabled (precondition of routine in static view is not *true*)

BON Dynamic Diagrams



Extension of Metamodel

- In general, checking view consistency will require theorem proving support.
  - **Key check**: prove that message *i* in the dynamic view has its precondition enabled by preceding messages *1,..,i-1*
- Effectively we want to show that for a collaboration diagram *cd* with sequence of calls *cd.calls*,

$$\forall i:2,..,cd.calls.length \bullet \exists\, cd.occurs \bullet$$
$$(init;\ cd.calls(1).spec\ ;\ ..\ ;\ cd.calls(i\text{-}1).spec$$
$$\Rightarrow \quad cd.calls(i).pre)$$

Expression in PVS

- ... is non-trivial.
- Need the following:
  - formalization of specifications (pre- and poststate) as new PVS type **SPECTYPE**
  - formalization of sequencing ;
  - formalization of specification state
- Add extra functions to the metamodel:
  - projection of static and dynamic views
  - sequence of routine calls in dynamic view

Specifications and Routines

- Each routine is formalized as a SPECTYPE.

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

- Given a routine and its pre/poststate we can produce a **SPECTYPE** using function

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

- Axiom needed to combine pre/postcondition of the routine into a single predicate.

Additional Infrastructure

- Two functions are needed:
  - **`seqspecs`**: the sequential composition of two **`SPECTYPE`**s
  - **`seqspecsn`**: lifted version of **`seqspecs`** to finite sequences

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

View Consistency Axiom

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

Just Off the Press...

- ... there is a small example of a consistency checking attempt in PVS in

R. Paige, J. Ostroff, P. Brooke, "Theorem Proving Support for View Consistency Checking", submitted to *L'Objet*, July 2002. (Draft available from the authors.)