Specification and Verification for Classes and Components

Bart Jacobs

Department of Computer Science,

University of Nijmegen

Email: bart@cs.kun.nl

URL: http://www.cs.kun.nl/~bart

and also

URL: http://www.cs.kun.nl/~bart/LOOP/

Contents

I. Components, objects, and their interfaces

II. The LOOP compiler and its input languages:

- CCSL, for Coalgebraic Class Specification Language
- JML, for Java Modeling Language
- Java

III. Demo: Example verification in Java.

Components, after Szyperski, Component Software (1997):

Software components are (binary) units with contractually specified interfaces and explicit context dependencies only. They can be produced, purchased, and deployed independently to form a functioning system.

Components are standard in all mature engineering disciplines.

Simple examples of components

- Applications in OS environment. E.g. in: `grep bart * | less`.
- Plug-ins, like helpers for Netscape or extensions for the MacOS.
- Compound documents: Microsoft's Visual Basic en OLE, Apple's OpenDoc, objects in HTML.
- Software libraries.

Current Standards

Corba, (D)COM, Java Beans, ...

Interfaces, non-technically

An interface of a component defines the component's access points, and describes how it connects to other components.

A component may have multiple interfaces, corresponding to different roles the component can play.

An interface specification is a contract between provider and client. This contract forms the basis for mutual interaction, because a component is typically developed without knowing its future clients.

In principle, the interface specification should remain stable over time, but the component implementing the specification may be revised.

Interface verification is needed to make sure that a component really implements its interface specification i.e. that it fulfils its contract. It is the task of the provider.

Standard view:

Objects

- have their own state, which is mutable
- encapsulate data + associated operations

Components

- are state-less and immutable. Thus they don't occur as multiple copies
- only provide operations (like modules, or abstract classes), and are separated from the data they act on.

Component interfaces & object interfaces

Szyperski, Component Software (1997), section 5.1.1:

> Most component implementations can be made to have their interfaces look like object interfaces

and

> Modeling all component interfaces as object interfaces, in which a single component may provide multiple object interfaces, helps simplify the grounds for discussion of semantics.

But what about state?

Once components are running, it becomes very artificial to keep a strict separation between data and operations.

Example

Consider ghostview as a component that can be launched by netscape. In itself, it is state-less. When it is displaying two documents in two separate displays, people talk about two ghostview-components.

Strictly speaking, there is only one component acting on two different documents (data).

For specification and verification purposes, it is most convenient to combine operations with data, so that the effects of the operations can be described in terms of the data.

Thus

Here, components will be treated as having state.

State in specifications

It is given by the values of the fields (also called attributes). Abstractly, in model based specifications, they are described via model (or ghost or specification) variables.

Example: Account

```
model int balance;
model int max_minus;

void credit(int change);
  requires: change >= 0
  ensures: balance = old(balance) + change

void debit(int change);
  requires: change >= 0  &  balance - change >= max_minus
  ensures: balance = old(balance) - change

invariant: balance >= max_minus
constraint: max_minus = old(max_minus)
```

Contractual specifiction for methods

A method specification involves two aspects:

- the type of the method, given by the types of the parameters and of the result (if any)
- the correctness assertions, typically given in terms of pre- and post-conditions

  (as popularised by the Eiffel language).

It is the client's resposability to make sure that the pre-condition holds when a method is invoked. And it is the provider's responsablity to guarantee that the post-condition holds when the method returns (assuming the pre-condition).

When a method is re-implemented (in a next version of a component), the pre-condition may be weakened and the post-condition may be strengthened.

Different versions of pre- and post-conditions may be used for normal and abnormal (exceptional) termination. (This is not in Eiffel.)

Modifies clauses tell which fields and parameters may have changed after a method invocation.

Contractual specifiction for interfaces

Specifications of interfaces involve:

- method specifications, as discussed before
- "global" assertions, like
  1. invariants: predicates on the state space which are maintained by all methods.

     Example: `size >= 0`.

  2. contraints: relations on the state space which describe a connection between what holds before and after all method invocations.

     Example: `max_size = old(max_size)`.

Implementation requirements

For each method and state x,

$$\mathtt{Pre}(\mathtt{old(x)}) \wedge \mathtt{Inv}(\mathtt{old(x)}) \Longrightarrow \mathtt{Post}(\mathtt{x}) \wedge \mathtt{Inv}(\mathtt{x}) \wedge \mathtt{Constr}(\mathtt{old(x), x})$$

[Aside: `old(x)` is `x@pre` in OCL.]

References as source of all problems

Specification and verification for components should be done in modular fashion.

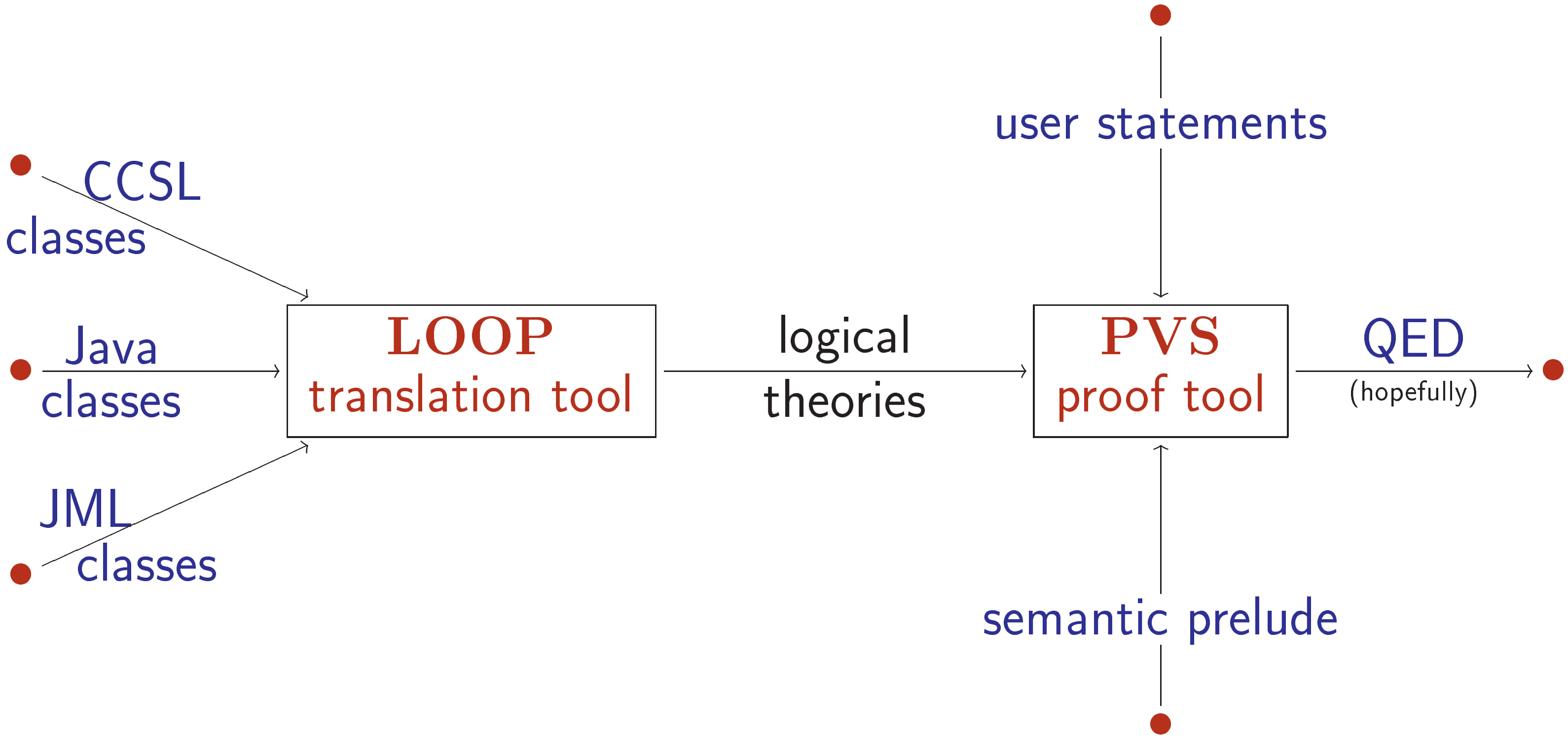
References create various kinds of problems, which ultimately all come down to representation exposure: some crucial integrity property, expressed e.g. as invariant, may be violated by an outsider holding a crucial reference.

This typically happens when one object calls a method on some other object with crucial references as parameters. Or when a crucial reference is returned as a result of a method.

Notorious cases are:

- callbacks, used for notification; they may modify the calling object.
- re-entrance: when during execution of one method another method of the same class is called.

There are several approaches to keep control over references, but there is no generally accepted solution.



The LOOP tool is implemented in Ocaml and has an Emacs interface.

It is possible to use also Isabelle as a back-end theorem prover.

CCSL, for Coalgebraic Class Specification Language is an experimental specifcation language for classes in object-oriented languages.

It involves parameterisation (by types and values), inheritance, aggregation.

The LOOP compiler automatically generates tailor-made definitions of invariant, bisimulation, $\Box$ (henceforth) and $\Diamond$ (eventually) for the CCSL classes.

The LOOP tool provides a semantics for CCSL classes in terms of so-called coalgebras in the logic of the back-end proof tool. (Similarly for Java and JML).

The LOOP tool is developed jointly at the universities of Nijmegen and Dresden, by:

Joachim van den Berg, Marieke Huisman, Bart Jacobs, Erik Poll (Nijmegen)

Ulrich Hensel, Hendrik Tews, Jan Rothe (Dresden).

```
BEGIN Queue[ A : Type ] : CLASSSPEC
    METHOD size : Self -> int;
           push : Self × A -> Self;
           pop : Self -> unit + (A × Self);     // unit = {*}
    ASSERTION
        size_push : FORALL(a:A) : size(push(x, a)) = size(x) + 1
        size_pop : CASES pop(x) OF
                        inl(*) : size(x) = 0,
                        inr(a, y) : size(y) = size(x) - 1
                    ENDCASES
        pop_push : FORALL(a:A) :
            CASES pop(push(x, a)) OF
                inl(*) : FALSE
                inr(b, y) : CASES pop(x) OF
                                inl(*) : b = a AND y ↔ x
                                inr(c, z) : b = c AND y ↔ push(z, a)
            ENDCASES ENDCASES
    CONSTRUCTOR empty : Self
    CREATION pop_empty : pop(empty) = inl(*)
END Queue
```

Topics in the use of CCSL

- Given a CCSL specification, construct a model (in PVS or Isabelle). This means
  1. defining a coalgebra consisting of a state space `Self = ...` with appropriate operations
  2. proving that the assertions are satisfied.

  (This often reveals omissions and errors in the specification.)

- Develop the theory of a specification, using e.g. the temporal logic of coalgebras for proving safety and liveness properties.

- Prove refinements between specifications.

JML, for Java Modeling Language is an extension of Java with correctness annotations, written after special comment signs, namely `//@`.

JML can be used in particular for interface specification, both for objects and components.

The main annotations are:

- Model variables, like in model based specification
- method specifications with pre- and post-conditions for normal and abnormal termination.
- invariants and constraints.

The development of JML started in the group of Gary Leavens at Iowa State University. The language is now combined with the annotation language used for static checking at Compaq System Research Center in Palo Alto. The semantics for the language is developed at Nijmegen.

JavaCard

JavaCard ia a relatively simple “sublanguage” of Java, designed for smartcards. (It has no threads, multi-dimensional arrays, floating-point numbers, ...)

JavaCard is an ideal target for application of formal methods.

JML for JavaCard

JML is used at Nijmegen for annotating the JavaCard API. This specification should form a basis for verification of JavaCard applets (programs), and for future extensions of the library.

Currently, the API classes contain only informal specifications. They should be improved and formalised, to enable verification.

Example: arrayCopy in `Util.java`

Type of the method:

```
public static final short arrayCopy(byte[] src, short srcOff,
                                    byte[] dest, short destOff, short length)
                          throws IndexOutOfBoundsException,
                                 NullPointerException,
                                 TransactionException
```

Type explanation:

```
Parameters: src - source byte array.
            srcOff - offset within source byte array to start copy from.
            dest - destination byte array.
            destOff - offset within destination byte array to start copy into.
            length - byte length to be copied.
Returns: destOff+length
Throws: IndexOutOfBoundsException -- if copying would cause access
                                     of data outside array bounds.
        NullPointerException -- if either src or dest is null.
        TransactionException -- if copying would cause the commit
                                capacity to be exceeded.
```

Informal explanation in `Util.java`:

```
If srcOff or destOff or length parameter is negative an
IndexOutOfBoundsException exception is thrown.

If srcOff+length is greater than src.length, the length of
the src array a IndexOutOfBoundsException exception is thrown
and no copy is performed.

If destOff+length is greater than dest.length, the length
of the dest array an IndexOutOfBoundsException exception is
thrown and no copy is performed.

If src or dest parameter is null a NullPointerException
exception is thrown.
  ...
```

```
... arrayCopy(byte[] src, short srcOff, byte[] dest, short destOff,
              short length) throws ...;
  //@ normal_behavior {
  //@   requires src != null && dest != null;
  //@   ensures ...; }
  //@ also
  //@ exceptional_behavior
  //@   {[
  //@       requires srcOff < 0 || destOff < 0 || length < 0;
  //@       signals (IndexOutOfBoundsException e) true;
  //@     also
  //@       requires srcOff + length > src.length;
  //@       signals (IndexOutOfBoundsException e) true;
  //@     also
  //@       requires destOff + length > dest.length;
  //@       signals (IndexOutOfBoundsException e) true;
  //@     also
  //@       requires src == null || dest == null;
  //@       signals (NullPointerException e) true;
  //@    ]}
```

Use of the semantics of JML

- An interface specification is translated by the LOOP tool into a predicate on a coalgebra, incorporating the contractual requirements.
- Models, theories and refinements can then be studied for such interface specifications, like in CCSL—using proof tools like PVS and Isabelle.
- Also one can prove that Java classes are implementations (i.e. models) of interfaces specifications, namely when their coalgebras satisfy the predicate of the specification.

Much work remains to be done here.

Verification of Java Programs

This is sofar the most succesful part of the LOOP project.

A denotational semantics for sequential Java has been elaborated in PVS and Isabelle. The basis is a non-trivial object memory model, involving explicit memory cells for storing objects in a heap-, stack or static part. The memory model serves as global state space.

Statements and expressions are translated as state transformer functions, which can either hang, terminate normally, or terminate abruptly. Language constructs like `if()then{}else{}` and `while` are defined on the basis of this representation.

During verification, automatic rewriting and decision procedures of the back-end proof tool are very important for all “trivial” steps.

For the non-trivial steps, a special Hoare logic has been developed.

The following example illustrates some semantic intricacies relating to mutual recursion and static initialisation. (It is from Jan Bergstra's lecture notes.)

```
class Yes {
    boolean result1, result2, result3;
    static boolean y = No.n;
    void m() {
        result1 = y;
        result2 = No.n;
        result3 = No.m();
    }
}

class No {
    static boolean n = !Yes.y;
    void m() { return n; }
}
```

What will be the values of `result1`, `result2` and `result3` after running `m` from `Yes` ??

In a recent, more substantial case study together with Marieke Huisman and Joachim van den Berg, a non-trivial invariant has been proved for Vector class from the standard Java class library.

The Vector class contains a private array field elementData, whose length is adapted automatically, according to needs. A private integer field elementCount keeps track of the number of valid components.

The predicate VectorIntegrity? that is shown to be an invariant is a conjunction of nine statements, the most important of which is:

$$\text{elementCount} \leq \text{elementData.length}$$

But also includes basic, but crucial, statements like:

$$\text{elementCount} \geq 0$$

$$\text{elementData} \neq \text{null}$$

For this case study, many classes from the Java class library, besides Vector, have been translated by the LOOP tool. This gave about 500K of PVS code.

Conclusions

- Based on a coalgebraic semantics, the LOOP tool translates CCSL class specifications, Java class implementations, and JML class specifications into PVS/Isabelle theories.

  The translation does not cover everything from Java. Excluded are, for example, threads, certain (runtime) exceptions, inner classes, API classes.

  It is not clear yet how much from JML will be covered.

- Using current technology, actual program verification seems feasible: automatic rewriting handles easy programs (without loops), and higher-level logics handle more complicated programs. But this kind of verification remains very labour- and knowledge-intensive.

Current and Future Work

- Advance the translation of JML.
- Specify the complete JavaCard API (45 classes), and show that the current implementation satisfies this implementation. Use this for verification of individual applets.
- Develop appropriate techniques for modular specification and verification, especially for components.

More info at URL: `http://www.cs.kun.nl/~bart/LOOP/`