Model-based Specifications in Larch/C++

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at

Model-based Specifications

C.A.R. Hoare: "Proof of Correctness of Data Representations" (1972).

- Verification of abstract datatype implementations.
  - Complements pre/post-condition reasoning about computations.
- Specification uses abstraction function $\mathcal{A} : C \rightarrow A$.
  - Maps concrete representations (objects of type `Stack`) to abstract values (mathematical "stacks").
  - Client of an ADT can reason about its operations without actually knowing its implementation.
- Verification uses inverse concretization function $\mathcal{C} : A \rightarrow \mathbb{P}(C)$.
  - Maps abstract values to (sets of) concrete values.
    - $\forall c \in C : c \in \mathcal{C}(\mathcal{A}(c))$.
    - $\forall a \in A, c \in \mathcal{C}(a) : \mathcal{A}(c) = a$.
  - Implementation of ADT must prove that its operations satisfy the properties expressed in the specification.

Example

```
interface Stack { void push(Elem e); Elem pop(); }
```

$\{\mathcal{A}(s) = S\}$
`s.push(e);`
$\{\mathcal{A}(s) = push(S, \mathcal{A}(e))\}$

$\{\mathcal{A}(s) = S \wedge \neg isEmpty(S)\}$
`e = s.pop();`
$\{\mathcal{A}(e) = top(S) \wedge \mathcal{A}(s) = pop(S)\}$

Pre/post-conditions in terms of abstract mathematical values.

Example (Contd)

```
class ArrayStack implements Stack
{
  Elem[] array; int n;
  ...
}
```

- $\mathcal{A}$: `ArrayStack` $\rightarrow$ *Stack*.
  - $\mathcal{A}(\texttt{array}, \texttt{n}) :=$ (informal sketch)
    $push(\ldots push(empty, \mathcal{A}(\texttt{array[0]}))\ldots, \mathcal{A}(\texttt{array[n-1]}))$.
- $\mathcal{C}$: *Stack* $\rightarrow \mathbb{P}($`ArrayStack`$)$.
  - $\mathcal{C}(empty) := \{\langle \texttt{array}, 0\rangle \mid \texttt{Elem[] array}\}$.
  - $\mathcal{C}(push(s, e)) :=$
    $\{\langle \texttt{array}, l+1\rangle : \exists a . \langle a, l\rangle \in \mathcal{C}(s) \wedge$
    $\forall\, 0 \leq i < l . \texttt{array}[i] = a[i] \wedge \texttt{array}[l] = \mathcal{C}(e)\}$

Must prove that $\mathcal{C}$ is inverse of $\mathcal{A}$.

Example (Contd'2)

$\texttt{class ArrayStack \{ ... void push(Elem e) \{ } \textit{body} \texttt{ \} ... \}}$

$\{\mathcal{A}(\texttt{array},\texttt{n}) = S\}\ \textit{body}\ \{\mathcal{A}(\texttt{array},\texttt{n}) = \textit{push}(S, \mathcal{A}(\texttt{e}))\}$

$\{\langle\texttt{array},\texttt{n}\rangle \in \mathcal{C}(S)\}\ \textit{body}\ \{\mathcal{A}(\texttt{array},\texttt{n}) = \textit{push}(S, \mathcal{A}(\texttt{e}))\}$

- Case $S = \textit{empty}$:

  $\{\texttt{n} = 0\}\ \textit{body}\ \{\ldots\}$

- Case $S = \textit{push}(s, e)$:

  $\{\exists l, a\ .\ \texttt{n} = l + 1 \land \langle a, l\rangle \in \mathcal{C}(s)\ \land$
  $\quad \forall\, 0 \le i < l\ .\ \texttt{array}[i] = a[i] \land \texttt{array}[l] = \mathcal{C}(e)\}$
  $\textit{body}$
  $\{\ldots\}$

Model-based Specification Languages

Abstract model specifies vocabulary used in pre/post-conditions.

- VDM-SL (Vienna Development Method Specification Language)
  - Started in the IBM laboratory in Vienna in the mid-1970s.
  - (Sort of) functional language to specify models.
- Z
  - Started at Oxford University (Hoare and others) in the late 1970s.
  - Set theory and first-order predicate logic to specify models.
- Larch: `http://www.sds.lcs.mit.edu/spd/larch`
  - Started at MIT in the late 1970s.
  - Larch Shared Language (LSL) to specify algebraic data types.
  - Several behavioral interface languages to specify modules in specific programming languages (including language-specific features).
    - LCL (for C), Larch/Ada, Larch/CLU, Larch/Smalltalk, Larch/C++.

ISO standards for VDM-SL (1996) and for Z (2002).

Larch/C++

- Behavioral interface specification language for C++.
  - Gary T. Leavens, Iowa State University, 1993-1999.
  - `http://www.cs.iastate.edu/~leavens/larchc++.html`.
- Shared layer: LSL traits.
  - Extensible specifications of ADTs.
  - Loose interpretation of algebraic specifications.
- Interface layer: Larch/C++ specification modules.
  - Specification of C++ classes.
  - Includes features dealing with state, aliasing, termination, etc.
- Larch/C++ tools.
  - `lcpp`: parser and type checker.
  - `lcpp2html`: generation of HTML pages.
  - LP: prover for reasoning about LSL traits.

Predecessor of the Java Modeling Language (JML).

Example: Four Sided Figures

Leavens: "An Overview of Larch/C++ Behavioral Specifications for C++ Modules", 1999.

```
// Quadrilateral.h
#include "QuadShape.h"

class Quadrilateral : virtual public QuadShape {
public:
  Quadrilateral(Vector v1, Vector v2, Vector v3, Vector v4,
                Vector pos) throw();
  //@ behavior {
  //@   requires isLoop(\<v1,v2,v3,v4\>);
  //@   modifies edges, position;
  //@   ensures liberally edges' = \<v1,v2,v3,v4\> /\ position' = pos;
  //@ }
};
```

The interface layer.

Example: Four Sided Figures (Contd'2)

```
// QuadShape.h
#include "Vector.h"

//@ uses FourSidedFigure;

/*@ abstract @*/ class QuadShape {
public:
  //@ spec Vector edges[4];
  //@ spec Vector position;
  //@ invariant isLoop(edges\any);
```

```
virtual Vector GetVec(int i)
  const throw();
//@ behavior {
//@ requires between(1, i, 4);
//@ ensures result = edges^[i-1];
//@ example i = 1 /\ result = edges^[0]; }

virtual Vector GetPosition()
  const throw();
//@ behavior {
//@   ensures result = position^; }  };
```

```
  virtual Move(const Vector& v) throw();
  //@ behavior {
  //@   requires assigned(v, pre);
  //@   requires redundantly assigned(edges, pre)
  //@              /\ assigned(position, pre) /\ isLoop(edges^);
  //@   modifies position;
  //@   trashes nothing;
  //@   ensures liberally position' = position^ + v^;
  //@   ensures redundantly liberally edges' = edges^;
  //@   example liberally position^ = 0:Vector /\ position' = v^; }
```


Example: Four Sided Figures (Contd'3)

```
% FourSidedFigure.lsl
FourSidedFigure(Scalar): trait

  includes
    PreVector(Scalar, Vector for Vec[T]),
    int, Val_Array(Vector)

  introduces
    isLoop: Arr[Vector] -> Bool
    \<__,__,__,__\>:
      Vector, Vector, Vector, Vector
      -> Arr[Vector]

  asserts
   \forall e: Arr[Vector], v1,v2,v3,v4:Vector
     isLoop(e) == (e[0] + e[1] + e[2] + e[3] = 0:Vector);
     \<v1,v2,v3,v4\>
        == assign(assign(assign(assign(create(4), 0,v1), 1,v2), 2,v3), 3,v4);
```

```
implies
 \forall e: Arr[Vector],
    v1,v2,v3,v4:Vector
    size(\<v1,v2,v3,v4\>) == 4;
    (\<v1,v2,v3,v4\>)[0] == v1;
    (\<v1,v2,v3,v4\>)[1] == v2;
    (\<v1,v2,v3,v4\>)[2] == v3;
    (\<v1,v2,v3,v4\>)[3] == v4;
    allAllocated(\<v1,v2,v3,v4\>);
 converts
   isLoop:Arr[Vector] -> Bool,
   \<__,__,__,__\>:
     Vector, Vector, Vector, Vector
     -> Arr[Vector]
```

The shared layer.

Example: Four Sided Figures (Contd'4)

```
% PreVector.lsl
PreVector(T): trait

  assumes RingWithUnit, Abelian(* for \circ),
          TotalOrder, CoerceToReal(T)

  includes PreVectorSpace(T), Real

  introduces
     __ \cdot __: Vec[T], Vec[T] -> T
    length: Vec[T] -> T

  asserts
    \forall u,v,w: Vec[T], a, b: T

      % the inner product is bilinear
      (u + v) \cdot w == (u \cdot w) + (v \cdot w);
      u \cdot (v + w) == (u \cdot v) + (u \cdot w);
      (a * u) \cdot v == a * (u \cdot v);
      (a * u) \cdot v == u \cdot (a * v);
```

```
    % ... and is commutative
    u \cdot v == v \cdot u;

    % ... and is positive definite
    (u \cdot u) >= 0;
    (u \cdot u = 0) == (u = 0);

    approximates(length(u),
      sqrt(toReal(u \cdot u)));
implies
  PreVectorSig(T)
  converts
     __ \cdot __: Vec[T], Vec[T] -> T
```


Example: Four Sided Figures (Contd'5)

```
% PreVectorSpace.lsl
PreVectorSpace(T): trait
  assumes RingWithUnit, Abelian(* for \circ)

  includes
    AbelianGroup
      (Vec[T] for T, + for \circ,
       0 for unit, - __ for \inv),
    DistributiveRingAction
      (T for M, Vec[T] for T)

  implies
    AC(+ for \circ, Vec[T] for T),
    Idempotent(- __, Vec[T])
    \forall u,v,w: Vec[T], a, b: T
      a * (u + v) == (a * u) + (a * v);
      (a + b) * u == (a * u) + (b * u);
      (a * b) * u == a * (b * u);
      1 * u == u;
      u - v == u + (- v);
      (u + v = u + w) => v = w;
      0 * u == 0:Vec[T];
```

```
      -(a * u) == (-a) * u;
      -(a * u) == a * (-u);
      (-a) * (-u) == a * u;
      (a \neq 0 /\ a * u = a * v) =>
        u = v;

    converts
      0: -> Vec[T],
      __+__: Vec[T], Vec[T] -> Vec[T],
      __*__: T, Vec[T] -> Vec[T],
      - __: Vec[T] -> Vec[T],
      __ - __: Vec[T], Vec[T] -> Vec[T]

PreVectorSig(T): trait
  introduces
    __ + __: Vec[T], Vec[T] -> Vec[T]
    __ * __: T, Vec[T] -> Vec[T]
    0: -> Vec[T]
    - __: Vec[T] -> Vec[T]
    __ - __: Vec[T], Vec[T] -> Vec[T]
    __ \cdot __: Vec[T], Vec[T] -> T
    length: Vec[T] -> T
```

Example: Four Sided Figures (Contd'6)

```
edsger2!448> lcpp
Usage: lcpp [preprocessor-options] [checker-options] file1.h [file2.lh ...]
    The checker-options are:
        --no-verbose    (don't print verbose messages)
        --no-LSL        (don't run the LSL checker)
        --keep-LSL      (keep LSL trait files if they have errors)
    The currently understood preprocessor options are:
         -ansi -Dmacro[=defn] -Umacro -Aquestion[(answer)] -nostdinc++ -undef
         -I dir -H dir -include file -imacros file -iprefix prefix
         -iwithprefix dir -idirafter dir
```

Syntax and type checking; no verification!

Example: Four Sided Figures (Contd'7)

```
edsger2!447> lcpp Quadrilateral.h

LCPP_builtins is up to date.

Checking Quadrilateral.h ...

Checking trait: Scalar
Finished checking LSL traits

Checking trait: PreVector(Scalar,Vector for Vec[T])
Finished checking LSL traits

NoContainedObjects(Vector) is up to date.

Checking trait: FourSidedFigure
Finished checking LSL traits

NoContainedObjects(Shear) is up to date.

Quadrilateral.h 0 warnings; 0 syntax & 0 semantic errors!
```


Proving LSL Properties

```
LP (the Larch Prover), Release 3.1b (98/06/09) logging to
'/usr3/Larch/lp3.1b/samples/list1.lplog' on 18 October 2005 16:18:26.

LP0.1.9: declare sorts Element, List
LP0.1.10: declare variables e: Element, x, y, z: List
LP0.1.11: declare operators
  null   :                -> List
  cons   : Element, List -> List
  append : List, List    -> List
  rev    : List          -> List
  ..

LP0.1.15: assert
  sort List generated by null, cons;
  append(null, x) = x;
  append(cons(e, y), z) = cons(e, append(y, z));
  rev(null) = null;
  rev(cons(e, y)) = append(rev(y), cons(e, null))
  ..
```


Proving LSL Properties (Contd)

```
LP0.1.22: prove rev(rev(x)) = x by induction

Attempting to prove conjecture theorem.1: rev(rev(x)) = x
Creating subgoals for proof by structural induction on 'x'
Basis subgoal:
  Subgoal 1: rev(rev(null)) = null
Induction constant: xc
Induction hypothesis:
  theoremInductHyp.1: rev(rev(xc)) = xc
Induction subgoal:
  Subgoal 2: rev(rev(cons(e, xc))) = cons(e, xc)

Attempting to prove level 2 subgoal 1 (basis step) for proof by induction on x
Level 2 subgoal 1 (basis step) for proof by induction on x
[] Proved by normalization.

Attempting to prove level 2 subgoal 2 (induction step) for proof by induction
on x
Added hypothesis theoremInductHyp.1 to the system.
Suspending proof of level 2 subgoal 2 (induction step) for proof by induction
on x
```

Proving LSL Properties (Contd'2)

```
LP0.1.24: % We need a lemma about rev(append(x, y)).
LP0.1.26: prove rev(append(x, y)) = append(rev(y), rev(x)) by induction on x

Attempting to prove level 3 lemma theorem.2:
  rev(append(x, y)) = append(rev(y), rev(x))

Creating subgoals for proof by structural induction on ‘x’
Basis subgoal:
  Subgoal 1: rev(append(null, y)) = append(rev(y), rev(null))
Induction constant: xc1
Induction hypothesis:
  theoremInductHyp.2: rev(append(xc1, y)) = append(rev(y), rev(xc1))
Induction subgoal:
  Subgoal 2: rev(append(cons(e, xc1), y)) = append(rev(y), rev(cons(e, xc1)))

Attempting to prove level 4 subgoal 1 (basis step) for proof by induction on x
Suspending proof of level 4 subgoal 1 (basis step) for proof by induction on x
```


Proving LSL Properties (Contd'3)

```
LP0.1.28: % We need another lemma, which we obtain by generalization.
LP0.1.30: prove append(x, null) = x by induction

Attempting to prove level 5 lemma theorem.3: append(x, null) = x
Creating subgoals for proof by structural induction on ‘x’
Basis subgoal:
  Subgoal 1: append(null, null) = null
Induction constant: xc1
Induction hypothesis:
  theoremInductHyp.2: append(xc1, null) = xc1
Induction subgoal:
  Subgoal 2: append(cons(e, xc1), null) = cons(e, xc1)

Attempting to prove level 6 subgoal 1 (basis step) for proof by induction on x
Level 6 subgoal 1 (basis step) for proof by induction on x
[] Proved by normalization.
```


Proving LSL Properties (Contd'4)

```
Attempting to prove level 6 subgoal 2 (induction step) for proof by induction
on x
Added hypothesis theoremInductHyp.2 to the system.
Level 6 subgoal 2 (induction step) for proof by induction on x
[] Proved by normalization.

Level 5 lemma theorem.3
[] Proved by structural induction on ‘x’.

Attempting to prove level 4 subgoal 1 (basis step) for proof by induction on x
Level 4 subgoal 1 (basis step) for proof by induction on x:
  rev(append(null, y)) = append(rev(y), rev(null))
[] Proved by normalization.

Attempting to prove level 4 subgoal 2 (induction step) for proof by induction
on x: rev(append(cons(e, xc1), y)) = append(rev(y), rev(cons(e, xc1)))
Added hypothesis theoremInductHyp.2 to the system.
Suspending proof of level 4 subgoal 2 (induction step) for proof by induction
on x
```


Proving LSL Properties (Contd'5)

```
LP0.1.32: % We need another lemma (the associativity of append)
LP0.1.35: prove append(append(x, y), z) = append(x, append(y, z)) by induction o
n x

Attempting to prove level 5 lemma theorem.3:
  append(append(x, y), z) = append(x, append(y, z))
Creating subgoals for proof by structural induction on ‘x’
Basis subgoal:
  Subgoal 1: append(append(null, y), z) = append(null, append(y, z))
Induction constant: xc2
Induction hypothesis:
  theoremInductHyp.3: append(append(xc2, y), z) = append(xc2, append(y, z))
Induction subgoal:
  Subgoal 2: append(append(cons(e, xc2), y), z)
               = append(cons(e, xc2), append(y, z))

Attempting to prove level 6 subgoal 1 (basis step) for proof by induction on x
Level 6 subgoal 1 (basis step) for proof by induction on x
[] Proved by normalization.
```

Proving LSL Properties (Contd'6)

```
Attempting to prove level 6 subgoal 2 (induction step) for proof by induction
on x
Added hypothesis theoremInductHyp.3 to the system.
Level 6 subgoal 2 (induction step) for proof by induction on x
[] Proved by normalization.

Level 5 lemma theorem.3
[] Proved by structural induction on ‘x’.

Attempting to prove level 4 subgoal 2 (induction step) for proof by induction
on x: rev(append(cons(e, xc1), y)) = append(rev(y), rev(cons(e, xc1)))
  Current subgoal:
    append(append(rev(y), rev(xc1)), cons(e, null))
      = append(rev(y), append(rev(xc1), cons(e, null)))

Level 4 subgoal 2 (induction step) for proof by induction on x
[] Proved by normalization.

Level 3 lemma theorem.2: rev(append(x, y)) = append(rev(y), rev(x))
[] Proved by structural induction on ‘x’.
```


Proving LSL Properties (Contd'7)

```
Attempting to prove level 2 subgoal 2 (induction step) for proof by induction
on x: rev(rev(cons(e, xc))) = cons(e, xc)
  Current subgoal: rev(append(rev(xc), cons(e, null))) = cons(e, xc)

Level 2 subgoal 2 (induction step) for proof by induction on x
[] Proved by normalization.

Conjecture theorem.1: rev(rev(x)) = x
[] Proved by structural induction on ‘x’.

LP0.1.36: qed
All conjectures have been proved.
```


The Java Modeling Language

- Behavioral interface specification language for Java.
  - Gary T. Leavens et al., Iowa State University, since 1999.
  - `http://www.jmlspecs.org`
- Fully embedded into the Java language.
  - No separation between shared layer and interface layer anymore.
  - All specifications expressed in (an extended version of) Java.
- Considerable community support.
  - `jml`: syntax and type checking.
  - `jmldoc`: document generation.
  - JMLEclipse: plugin for the Eclipse IDE.
  - ESC/Java2: extended static checking of JML specifications.

Java programmer needs not learn a new expression language, but distinction between model and representation gets blurred.

A Stack Model

```
public /*@ pure @*/ class IntStackModel
{
  // IntStackModel() is default constructor

  //@ public model boolean isempty();
  //@ public model IntStackModel push(int e);
  //@ public model int top();
  //@ public model IntStackModel pop();

  /*@ public invariant
    @  (\forall IntStackModel s, s2; s != null;
    @     (\forall int e, e2; ;
    @        !new IntStackModel().equals(s.push(e)) &&
    @        (s.push(e).equals(s2.push(e2)) ==> s.equals(s2) && e == e2) &&
    @        new IntStackModel().isempty() &&
    @        !s.push(e).isempty() &&
    @        e == s.push(e).top() &&
    @        s.equals(s.push(e).pop())));
    @*/
}
```

A Stack Implementation

```
public class IntStack // "IntStack.jml"
{
  /*@ public model
    @   non_null IntStackModel stackM;
    @ public initially stackM.isempty();
    @
    @ represents stackM <- toModel();
    @ public model
    @   pure IntStackModel toModel(); @*/

  /*@ public normal_behavior
    @   assignable stackM;
    @   ensures stackM.isempty(); @*/
  public IntStack();

  /*@ public normal_behavior
    @   assignable \nothing;
    @   ensures \result <==> stackM.isempty(); @*/
  public /*@ pure @*/ boolean isempty();
```

```
  /*@ public normal_behavior
    @   assignable stackM;
    @   ensures stackM ==
    @     \old(stackM.push(e)); @*/
  public void push(int e);

  /*@ public normal_behavior
    @   requires !stackM.isempty();
    @   assignable stackM;
    @   ensures \result ==
    @     \old(stackM.top())
    @   && stackM ==
    @     \old(stackM.pop()); @*/
  public int pop(int e);
}
```

See course on “Formal Methods in Software Development”.