Axiomatic Specifications in ✓eriFun

Andreas Schlosser Christoph Walther Markus Aderhold

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Programmiermethodik, Technische Universität Darmstadt, Germany
http://www.informatik.tu-darmstadt.de/pm

3rd International *Verification*-Workshop
Seattle, August 15–16, 2006

The √eriFun System

√eriFun at a Glance

- Interactive inductive theorem prover for the verification of programs
- Freely generated polymorphic data types
- First-order procedures operating on these data types based on recursion, case analyses, and functional composition
- Statements (called "lemmas") about the data types and the procedures, defined by universal quantifications
- Powerful induction heuristics
- Semi-automated termination analysis

Specifications

What is a Specification?

- Signature (operators and domains)
- Description of behavior (informally or formally)

Why Use Specifications?

- Modularization
  - Organize large programs
  - Interface definitions (Design by Contract)
  - Reuse code/proofs
- Abstraction vs. Implementation/Instantiation

Capabilities of Specifications

- Using simple specifications
- Parameterizing specifications
- Building hierarchies
  - Inheritance
  - Renaming
  - Multi-Usage
  - Sharing
- Instantiation
- Modeling non-free data types

Specifying a Total Order

```
specification TotalOrder
  domain @O
  operator le : @O, @O → bool
  axiom reflexivity <= ∀x:@O le(x, x)
  axiom antisymmetry <= ∀x, y:@O le(x, y) ∧ le(y, x) → x = y
  axiom transitivity <= ∀x, y, z:@O le(x, y) ∧ le(y, z) → le(x, z)
  axiom totality <= ∀x, y:@O le(x, y) ∨ le(y, x)
```

Using a Total Order

```
function ordered(k:list[ℕ]):bool <=
if k = ø
  then true
  else if tl(k) = ø
    then true
    else if hd(k) ≤ hd(tl(k))
      then ordered(tl(k))
      else false
end end end
```

Using a Total Order

```
function ordered[T:TotalOrder](k:list[@O]):bool <=
if k = ø
  then true
  else if tl(k) = ø
    then true
    else if le(hd(k), hd(tl(k)))
      then ordered[T](tl(k))
      else false
end end end
```

Capabilities of Specifications

- Using simple specifications
- Parameterizing specifications
- Building hierarchies
  - Inheritance
  - Renaming
  - Multi-Usage
  - Sharing
- Instantiation
- Modeling non-free data types

Specifying a Priority Queue

specification $\mathit{PriorityQueue}[T\!:\!\mathit{TotalOrder}]$
  domain $Q[@O]$
  operator $\mathit{new} : Q[@O]$
  operator $\mathit{ins} : @O, Q[@O] \rightarrow Q[@O]$
  operator $\mathit{min} : Q[@O] \rightarrow @O$
  operator $\mathit{dm} : Q[@O] \rightarrow Q[@O]$
  operator $\mathit{size} : Q[@O] \rightarrow \mathbb{N}$
  axiom $\mathit{min\ ins\ new} <= \forall v{:}@O\ \mathit{min}(\mathit{ins}(v, \mathit{new})) = v$
    $\vdots$

Using a Priority Queue

$\texttt{function}\ makeQueue[P{:}PriorityQueue](l{:}list[@O]){:}Q[@O] <=$
$\textbf{\textit{if}}\ l = \emptyset\ \textbf{\textit{then}}\ new\ \textbf{\textit{else}}\ ins(hd(l), makeQueue[P](tl(l)))\ \textbf{\textit{end}}$

$\texttt{function}\ makeList[P{:}PriorityQueue](q{:}Q[@O]){:}list[@O] <=$
$\textbf{\textit{if}}\ q = new\ \textbf{\textit{then}}\ \emptyset\ \textbf{\textit{else}}\ min(q) :: makeList[P](dm(q))\ \textbf{\textit{end}}$

$\texttt{function}\ sort[P{:}PriorityQueue](l{:}list[@O]){:}list[@O] <=$
$makeList[P](makeQueue[P](l))$

Using a Priority Queue

$\texttt{function}$ $\mathit{makeQueue}[P{:}\mathit{PriorityQueue}](l{:}\mathit{list}[@O]){:}Q[@O] <=$
$\textbf{\textit{if}}\ l = \emptyset\ \textbf{\textit{then}}\ \mathit{new}\ \textbf{\textit{else}}\ \mathit{ins}(\mathit{hd}(l), \mathit{makeQueue}[P](\mathit{tl}(l)))\ \textbf{\textit{end}}$

$\texttt{function}$ $\mathit{makeList}[P{:}\mathit{PriorityQueue}](q{:}Q[@O]){:}\mathit{list}[@O] <=$
$\textbf{\textit{if}}\ q = \mathit{new}\ \textbf{\textit{then}}\ \emptyset\ \textbf{\textit{else}}\ \mathit{min}(q) :: \mathit{makeList}[P](\mathit{dm}(q))\ \textbf{\textit{end}}$

$\texttt{function}$ $\mathit{sort}[P{:}\mathit{PriorityQueue}](l{:}\mathit{list}[@O]){:}\mathit{list}[@O] <=$
$\mathit{makeList}[P](\mathit{makeQueue}[P](l))$

$\texttt{lemma}$ $\mathit{sort\ sorts}[P{:}\mathit{PriorityQueue}] <= \forall l{:}\mathit{list}[@O]$
$\mathit{ordered}[T(P)](\mathit{sort}[P](l))$

$\texttt{lemma}$ $\mathit{sort\ permutes}[P{:}\mathit{PriorityQueue}] <= \forall n{:}@O, l{:}\mathit{list}[@O]$
$\mathit{occurs}(n, l) = \mathit{occurs}(n, \mathit{sort}[P](l))$

Capabilities of Specifications

- Using simple specifications
- Parameterizing specifications
- Building hierarchies
  - Inheritance
  - Renaming
  - Multi-Usage
  - Sharing
- Instantiation
- Modeling non-free data types

Specifying Algebraic Structures

```
specification Monoid
  domain @M
  operator op : @M, @M → @M
  operator neut : @M
  axiom left neut <= ∀x:@M op(neut, x) = x
  axiom right neut <= ∀x:@M op(x, neut) = x
  axiom op assoc <= ∀x, y, z:@M op(op(x, y), z) = op(x, op(y, z))
```

`specification` *Monoid*

Inheritance

`specification` $Group[M{:}Monoid(@M, op, neut)]$
  `operator` $inv : @G \to @G$
  `axiom` *inv op right* $<= \forall x{:}@G \ \ op(x, inv(x)) = neut$
  `lemma` *inv op left* $<= \forall x{:}@G \ \ op(inv(x), x) = neut$
  `lemma` *inv inv* $<= \forall x{:}@G \ \ inv(inv(x)) = x$

Specifying Algebraic Structures

`specification` *Monoid*

Inheritance

`specification` $Group[M{:}Monoid(@M, op, neut)]$

Multi-Usage and Renaming

`specification` $GroupHomomorphism[G_1{:}Group(@G_1, op_1, neut_1, inv_1),$
$G_2{:}Group(@G_2, op_2, neut_2, inv_2)]$

`operator` $h : @G_1 \to @G_2$
`axiom` $homomorphism <= \forall x, y{:}@G_1\ op_2(h(x), h(y)) = h(op_1(x, y))$
`lemma` $h\ keeps\ neut <= h(neut_1) = neut_2$
`lemma` $h\ keeps\ inv <= \forall x{:}@G_1\ h(inv_1(x)) = inv_2(h(x))$

Specifying Algebraic Structures

specification *Monoid*

Inheritance

specification $Group[M{:}Monoid(@M, op, neut)]$

Multi-Usage and Renaming

specification $GroupHomomorphism[G_1{:}Group(@G_1, op_1, neut_1, inv_1),$
$G_2{:}Group(@G_2, op_2, neut_2, inv_2)]$

Sharing

specification $RingUnit[G{:}Group(@R, plus, zero, minus),$
$M{:}Monoid(@R, mult, one)]$

axiom *plus commutativity* $<= \forall x, y{:}@R\ plus(x, y) = plus(y, x)$

axiom *left distributivity* $<= \ldots$ axiom *right distributivity* $<= \ldots$

Capabilities of Specifications

- Using simple specifications
- Parameterizing specifications
- Building hierarchies
  - Inheritance
  - Renaming
  - Multi-Usage
  - Sharing
- Instantiation
- Modeling non-free data types

Instantiating Monoid

$\texttt{function}\ \mathit{fold}[M{:}\mathit{Monoid}](l{:}\mathit{list}[@M]){:}@M <=$
$\textbf{\textit{if}}\ l = \emptyset\ \textbf{\textit{then}}\ \mathit{neut}\ \textbf{\textit{else}}\ \mathit{op}(\mathit{hd}(l), \mathit{fold}[M](\mathit{tl}(l)))\ \textbf{\textit{end}}$

$\texttt{lemma}\ \mathit{fold\ append}[M{:}\mathit{Monoid}] <= \forall l_1, l_2{:}\mathit{list}[@M]$
$\mathit{op}(\mathit{fold}[M](l_1), \mathit{fold}[M](l_2)) = \mathit{fold}[M](\mathit{append}(l_1, l_2))$

$\texttt{instance}\ \mathit{Plus} <= \mathit{Monoid}(\mathbb{N}, +, 0)$

Instantiating Monoid

function $fold[M{:}Monoid](l{:}list[@M]){:}@M$ <=
if $l = \emptyset$ ***then*** $neut$ ***else*** $op(hd(l), fold[M](tl(l)))$ ***end***

lemma $fold\ append[M{:}Monoid]$ <= $\forall l_1, l_2{:}list[@M]$
$op(fold[M](l_1), fold[M](l_2)) = fold[M](append(l_1, l_2))$

instance $Plus$ <= $Monoid(\mathbb{N}, +, 0)$

function $fold[Plus](l{:}list[\mathbb{N}]){:}\mathbb{N}$ <=
if $l = \emptyset$ ***then*** 0 ***else*** $hd(l) + fold[Plus](tl(l))$ ***end***

lemma $fold\ append[Plus]$ <= $\forall l_1, l_2{:}list[\mathbb{N}]$
$fold[Plus](l_1) + fold[Plus](l_2) = fold[Plus](append(l_1, l_2))$

Capabilities of Specifications

- Using simple specifications
- Parameterizing specifications
- Building hierarchies
  - Inheritance
  - Renaming
  - Multi-Usage
  - Sharing
- Instantiation
- Modeling non-free data types

Specifying Integers I

Naive Specification

specification *Integer*
  domain $@I$
  operator *zero* : $@I$
  operator *succ* : $@I \rightarrow @I$
  operator *pred* : $@I \rightarrow @I$
  axiom *pred succ* $<= \forall i{:}@I \;\; pred(succ(i)) = i$
  axiom *succ pred* $<= \forall i{:}@I \;\; succ(pred(i)) = i$
  axiom *succ not id* $<= \forall i{:}@I \;\; succ(i) \neq i$
  axiom *pred not id* $<= \forall i{:}@I \;\; pred(i) \neq i$
  axiom *succ injective* $<= \forall i_1, i_2{:}@I \;\; succ(i_1) = succ(i_2) \rightarrow i_1 = i_2$
  axiom *pred injective* $<= \forall i_1, i_2{:}@I \;\; pred(i_1) = pred(i_2) \rightarrow i_1 = i_2$

Specifying Integers II

Addition on $\mathbb{N}$

$\texttt{function} +(x, y{:}\mathbb{N}) <=$
if $?0(x)$
  then y
  else $succ(pred(x) + y)$
end

Addition on $\mathbb{Z}$

$plus[I](zero, y) = y$
$plus[I](x, y) =$
  $succ(plus[I](pred(x), y))$
$plus[I](x, y) =$
  $pred(plus[I](succ(x), y))$

Specifying Integers II

Addition on $\mathbb{N}$

$\texttt{function} +(x, y{:}\mathbb{N}) <=$
if $?0(x)$
  then y
  else $succ(pred(x) + y)$
end

Addition on $\mathbb{Z}$

$plus[I](zero, y) = y$
$plus[I](x, y) =$
  $succ(plus[I](pred(x), y))$
$plus[I](x, y) =$
  $pred(plus[I](succ(x), y))$

Advanced Specification

$\texttt{specification}$ *Integer*
  $\texttt{operator}$ $sign : @I \rightarrow int\_sign$
  $\texttt{operator}$ $abs : @I \rightarrow \mathbb{N}$
  $\texttt{axiom}$ *sign definition* $<= \forall i{:}@I \ \ldots$
  $\texttt{axiom}$ *abs definition* $<= \forall i{:}@I \ \ldots$

$\texttt{structure}$ $int\_sign <=$
  $pos, neut, neg$

Definition of Addition on Integers

$\texttt{function}\ plus[I{:}Integer](x, y{:}@I){:}@I \ <=$
case $sign(x)$ **of**
  $pos : succ(plus[I](pred(x), y)),$
  $neut : y,$
  $neg : pred(plus[I](succ(x), y))$
end

Termination Hypotheses Using $\lambda x, y.abs(x)$

$sign(x) = pos \rightarrow abs(x) > abs(pred(x))$
$sign(x) = neg \rightarrow abs(x) > abs(succ(x))$

Proving Properties about Integers

Induction Axiom Schema Derived from the Definition of *plus*

$$\frac{\begin{array}{c}\forall x : @I \;\; sign(x) = neut \to \phi[x] \\ \forall x : @I \;\; sign(x) = pos \wedge \phi[pred(x)] \to \phi[x] \\ \forall x : @I \;\; sign(x) = neg \wedge \phi[succ(x)] \to \phi[x]\end{array}}{\forall x : @I \;\; \phi[x]}$$

Lemmas Proved by Induction

`lemma` $plus\ associative[I{:}Integer] <= \forall x, y, z{:}@I$
$\quad plus[I](x, plus[I](y, z)) = plus[I](plus[I](x, y), z)$

`lemma` $plus\ commutative[I{:}Integer] <= \forall x, y{:}@I$
$\quad plus[I](x, y) = plus[I](y, x)$

Instantiation with Integers

```
function uminus[I:Integer](x:@I):@I <=
case sign(x) of
  pos : pred(uminus[I](pred(x))),
  neut : x,
  neg : succ(uminus[I](succ(x)))
end
lemma plus uminus is zero[I:Integer] <= ∀x:@I
  plus[I](x, uminus[I](x)) = zero
```

```
instance IntegersGroup[I:Integer] <=
  Group(@I, zero, plus[I], uminus[I])
```

Semantics of Procedure *plus*

Example (Signature Extended *plus* with Translated Recursive Call)

$\texttt{function}\ plus^*(zero{:}@I, succ, pred{:}@I \rightarrow @I, sign{:}@I \rightarrow int\_sign,$
$\quad abs{:}@I \rightarrow \mathbb{N}, x, y{:}@I){:}@I <=$

$\textbf{\textit{case}}\ sign(x)\ \textbf{\textit{of}}$

$\quad pos : succ(plus^*(zero, succ, pred, sign, abs, pred(x), y)),$

$\quad neut : y,$

$\quad neg : pred(plus^*(zero, succ, pred, sign, abs, succ(x), y))$

$\textbf{\textit{end}}$

Semantics of Lemma *plus commutative*

Example (Lemma *plus commutative*)

$$\texttt{lemma}\ \mathit{plus\ commutative} <= \forall \mathit{zero}{:}@I, \mathit{succ}, \mathit{pred}{:}@I \rightarrow @I,$$
$$\mathit{sign}{:}@I \rightarrow \mathit{int\_sign}, \mathit{abs}{:}@I \rightarrow \mathbb{N}$$
$$\mathit{pred\ succ} \wedge \mathit{succ\ pred} \wedge \mathit{succ\ not\ id} \wedge \mathit{pred\ not\ id} \wedge$$
$$\mathit{succ\ injective} \wedge \mathit{pred\ injective} \wedge \mathit{sign\ definition} \wedge \mathit{abs\ definition} \rightarrow$$
$$\forall x, y{:}@I\ \mathit{plus}^*(\mathit{zero}, \mathit{succ}, \mathit{pred}, \mathit{sign}, \mathit{abs}, x, y) =$$
$$\mathit{plus}^*(\mathit{zero}, \mathit{succ}, \mathit{pred}, \mathit{sign}, \mathit{abs}, y, x)$$

Conclusion

- Integrated into an experimental version of the √eriFun
- High degree of user interaction upon reasoning involving *axioms*
  → Integration of a first-order theorem prover
- High flexibility in combination of specifications

	√eriFun	ACL2	IMPS	PVS	Isabelle/HOL	Locales	MAYA	Nuprl	Coq
inheritance	✓	✗	•	✓	•	✓	✓	✓	✓
parameterization	✓	✗	✗	✓	•	✓	✓	✓	✓
type operator vars	✓	✗	✗	✗	✗	✗	✗	✗	✗
referencing	✓	✗	✗	✗	✗	✗	✗	✗	✗
multi-usage	✓	✗	✗	✓	✗	✓	✓	•	✓
sharing	✓	✗	✗	✗	✗	✓	✓	•	✗
instantiation	✓	•	✓	✓	✓	✓	✓	✗	✓

✓: full support •: partial support ✗: no support

- Induction proofs over non-freely generated data types

Capabilities of Other Systems

	VeriFun	ACL2	IMPS	PVS	Isabelle/ HOL	Locales	MAYA	Nuprl	Coq
inheritance	✓	✗	•	✓	•	✓	✓	✓	✓
parameterization	✓	✗	✗	✓	•	✓	✓	✓	✓
type operator vars	✓	✗	✗	✗	✗	✗	✗	✗	✗
referencing	✓	✗	✗	✗	✗	✗	✗	✗	✗
multi-usage	✓	✗	✗	✓	✗	✓	✓	•	✓
sharing	✓	✗	✗	✗	✗	✓	✓	•	✗
instantiation	✓	•	✓	✓	✓	✓	✓	✗	✓

✓: full support •: partial support ✗: no support

Specifying Integers II

$\texttt{structure}$ $int\_sign <= pos, neut, neg$

$\texttt{specification}$ $Integer$

$\quad\texttt{operator}$ $sign : @I \longrightarrow int\_sign$

$\quad\texttt{operator}$ $abs : @I \longrightarrow \mathbb{N}$

$\quad\texttt{axiom}$ $sign\ definition <= \forall i{:}@I\ (i = zero \leftrightarrow sign(i) = neut) \wedge$

$\quad\quad (sign(pred(zero)) = neg) \wedge (sign(succ(zero)) = pos) \wedge$

$\quad\quad (sign(i) = pos \longrightarrow sign(succ(i)) = pos \wedge sign(pred(i)) \neq neg) \wedge$

$\quad\quad (sign(i) = neg \longrightarrow sign(pred(i)) = neg \wedge sign(succ(i)) \neq pos)$

$\quad\texttt{axiom}$ $abs\ definition <= \forall i{:}@I$

$\quad\quad \mathbf{case}\{sign(i);$

$\quad\quad\quad pos : abs(i) = {}^{-}(abs(succ(i))) \wedge abs(i) = {}^{+}(abs(pred(i))),$

$\quad\quad\quad neut : abs(i) = 0,$

$\quad\quad\quad neg : abs(i) = {}^{+}(abs(succ(i))) \wedge abs(i) = {}^{-}(abs(pred(i)))\}$

Definition (Specifications, Environments, and Instances)

- *Specification* $s = (S, \mathcal{E}, \mathcal{V}_T, \mathcal{V}_O, \mathcal{AX})$
- *Specification conversion* $\gamma := \langle \xi, \sigma \rangle$,
  γ is an *s-conversion* if $dom(\xi) \subseteq \mathcal{V}_T$ and $dom(\sigma) \subseteq \mathcal{V}_O$
- *Environment* $\mathcal{E} = \langle e_1{:}(s_1, \gamma_1), \ldots, e_n{:}(s_n, \gamma_n) \rangle$
  where each γ_i is an s_i-conversion

Specifications et al.

Definition (Specifications, Environments, and Instances)

- *Specification* $s = (S, \mathcal{E}, \mathcal{V}_T, \mathcal{V}_O, \mathcal{AX})$
- *Specification conversion* $\gamma := \langle \xi, \sigma \rangle$,
  γ is an *s-conversion* if $dom(\xi) \subseteq \mathcal{V}_T$ and $dom(\sigma) \subseteq \mathcal{V}_O$
- *Environment* $\mathcal{E} = \langle e_1{:}(s_1, \gamma_1), \ldots, e_n{:}(s_n, \gamma_n) \rangle$
  where each γ_i is an s_i-conversion

Example (*Monoid* and *Group*)

$$m = (Monoid, \emptyset, \langle @M \rangle, \langle op : @M, @M \rightarrow @M, neut : @M \rangle,$$
$$\{\forall x{:}@M \;\; op(neut, x) = x, \forall x{:}@M \;\; op(x, neut) = x,$$
$$\forall x, y, z{:}@M \;\; op(op(x, y), z) = op(x, op(y, z))\})$$

$$g = (Group, \langle M{:}(m, \langle \{@M/@G\}, \{\} \rangle) \rangle, \emptyset, \langle inv : @M \rightarrow @M \rangle, \ldots)$$

Referencing

Definition (Environment Terms)

Set $\mathcal{T}_\mathcal{E}$ of *specification typed environment terms* in $\mathcal{E}$ is defined as the smallest set satisfying (1)–(2). To each environment term $e \in \mathcal{T}_\mathcal{E}$, a corresponding specification conversion $\gamma_\mathcal{E}$ is assigned.

1. $a_s \in \mathcal{T}_\mathcal{E}$ for each $a \in dom(\mathcal{E})$ with $\mathcal{E}(a) = (s, \gamma)$;
   and $\gamma_\mathcal{E}(a_s) := \gamma$
2. $b(a_s)_{s'} \in \mathcal{T}_\mathcal{E}$ for each $a_s \in \mathcal{T}_\mathcal{E}$ with $s = (S, \mathcal{E}_s, \mathcal{V}_T, \mathcal{V}_O, \mathcal{AX})$ and for each $b \in dom(\mathcal{E}_s)$ with $\mathcal{E}_s(b) = (s', \gamma')$;
   and $\gamma_\mathcal{E}(b(a_s)_{s'}) := \gamma_\mathcal{E}(a_s) \circ \gamma'$

Sometimes an environment term e_s is denoted as $e{:}s$.

Signature Extended Procedures

Signature Extending

Semantics of `function` $func[\mathcal{E}](x_1{:}\tau_1, \ldots, x_m{:}\tau_m){:}\tau$
with $\mathcal{E} := \langle e_1{:}(s_1, \gamma_1), \ldots, e_n{:}(s_n, \gamma_n)\rangle$ is semantics of
`function` $func^*(\gamma_1(\mathcal{V}_O(s_1)), \ldots, \gamma_n(\mathcal{V}_O(s_n)), x_1{:}\tau_1, \ldots, x_m{:}\tau_m){:}\tau$

Translating Procedure Calls

Procedure call $func[a_1{:}s_1, \ldots, a_n{:}s_n](t_1, \ldots, t_m)$
in environment $\mathcal{E}$ with $a_i{:}s_i \in \mathcal{T}_\mathcal{E}$ is translated into call
$func^*(\gamma_\mathcal{E}(a_1)(\mathcal{V}_O(s_1)), \ldots, \gamma_\mathcal{E}(a_n)(\mathcal{V}_O(s_n)), t_1, \ldots, t_m)$

True Modulo Axioms

Definition (Semantics of Lemmas With Environments)

Lemma `lemma` $lem[\mathcal{E}]$ <= $\forall x_1{:}\tau_1, \ldots, x_m{:}\tau_m\ b$
with $\mathcal{E} := \langle e_1{:}(s_1, \gamma_1), \ldots, e_n{:}(s_n, \gamma_n)\rangle$ is *true* iff

for all σ_O with $dom(\sigma_O) = \bigcup_{i=1}^{n} \gamma_i(\mathcal{V}_O(s_i))$
assigning the first-order variables *arbitrary* terminating $\mathcal{L}$-procedures holds:

$eval_P(\sigma_\xi(\sigma_O(b))) = true$ for all σ_ξ if
$eval_P(\sigma'_{\xi'}(\sigma_O(ax))) = true$ for all $\sigma'_{\xi'}$
for all axioms $\forall z_1{:}\upsilon_1, \ldots, z_k{:}\upsilon_k\ ax \in \bigcup_{i=1}^{n} \gamma_i(\mathcal{AX}(s_i))$