Composition and Refinement of Behavioral Specifications

Dusko Pavlovic
Douglas R. Smith

Kestrel Institute
Palo Alto, California

www.kestrel.edu
Specifications and Morphisms

spec Partial-Order is
  sort E
  op le: E, E → Boolean
  axiom reflx is le(x,x)
  axiom trans is le(x,y) ∧ le(y,z) ⇒ le(x,z)
  axiom antis is le(x,y) ∧ le(y,x) ⇒ x = y
end-spec

spec Integer is
  sort Int
  op ≤: Int, Int → Boolean
  op 0 : Int
  op _+_: Int, Int → Int
  ...
end-spec

Specification morphism: a language translation that preserves provability
Specification Carrying Software

\[ \langle P, \models, S \rangle \]

- \( P \) = program
- \( S \) = specification
- \( \models \) = model relation
Specification Carrying Software

\[ A = \langle P_A, \models_A, S_A \rangle \]

\[ B = \langle P_B, \models_B, S_B \rangle \]

\[ f \]

\[ f_P \]

\[ f_S \]

\[ f_P(b) \models_A \alpha \]

\[ b \models_B f_S(\alpha) \]
Evolving specifications (especs)

Key ideas that link state machine concepts with logical concepts

1. States are models (structures satisfying axioms)

<table>
<thead>
<tr>
<th>State</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>datatypes</td>
<td>sets</td>
</tr>
<tr>
<td>variables</td>
<td>functions, values</td>
</tr>
<tr>
<td>properties</td>
<td>axioms, theorems</td>
</tr>
</tbody>
</table>

2. State transitions are finite model changes

Example: Updating an array/finite-function A

A : {1,2,3} →Nat

A(3) := 4

A : {1,2,3} →Nat
Evolving specifications (es specs)

3. Abstract states are sets of states
Specs denote sets of models
Specs represent abstract states

4. Abstract transitions are interpretations (in the opposite direction)!

\[ \text{correctness condition: } pre \vdash post(e) \]
Evolving specifications (especs)

5. Abstract Guarded Transitions

\[ g(x) \vdash x := e \]

These pairs of arrows are monics and epis of an abstract factorization system, with a general construction for composition and colimit. (AMAST02)
Espec for a GCD Algorithm

Each state has common structure:

\[ x_0, y_0 : \text{Pos} \]
\[ x, y, z : \text{Pos} \]

|x| y|
---|---|
6 | 9 |

axiom \( z = \text{gcd}(x_0, y_0) \)

\[ x > y \rightarrow x := x - y \]

\[ y > x \rightarrow y := y - x \]

\[ x = y \rightarrow z := x \]

\[ x > y \rightarrow x := x - y \]

axiom \( \text{gcd}(x_0, y_0) = \text{gcd}(x, y) \)

example run:

\[
\begin{array}{c|c|c|c|c}
  x & y & z \\
  9 & 6 & - & - \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
  x & y & z \\
  9 & 6 & 9 & 6 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
  x & y & z \\
  9 & 6 & 3 & 3 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
  x & y & z \\
  9 & 6 & 3 & 3 \\
\end{array}
\]

Kestrel Institute
GCD espec

espec GCD-base is
spec
op X-in,Y-in : Pos
op X,Y : Pos
op Z : Pos
op gcd : Pos, Pos -> Pos
axiom gcd-spec is
gcd(x,y) = z => (divides(z,x) & divides(z,y)
& forall(w:Pos)(divides(w,x) & divides(w,y) => w <= z))
end-spec

prog
stad One init[X-in,Y-in] is
end-stad

step initialize : One -> Loop is
  X := X-in
  Y := Y-in
end-step

step Loop1 : Loop -> Loop is
  X := X - Y
  cond X>Y
end-step

step Loop2 : Loop -> Loop is
  Y := Y - X
  cond Y>X
end-step

stad Loop is
thm loop-invariant is
  gcd(X-in,Y-in) = gcd(X,Y)
end-stad

stad Two fin[Z] is
axiom Z = gcd(X-in,Y-in)
axiom X = Y
end-stad

step Out : Loop -> Two is
  Z := X
  cond X = Y
end-step

Kestrel Institute
GCD espcs, states, and computation

GCD-base

One ← Loop → Two

Global spec
extends to
abstract state specs
denotes
states/models
# Control Constructs vs Logical Concepts

<table>
<thead>
<tr>
<th><strong>Command Language</strong></th>
<th><strong>Logical Concepts</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{P} x := e {Q}</code></td>
<td>interpretation $I: \text{Thy}_Q \to \text{Thy}_P$</td>
</tr>
<tr>
<td>skip</td>
<td>identity interpretation</td>
</tr>
<tr>
<td>sequencing $S1;S2$</td>
<td>composition $I_1 \circ I_2$</td>
</tr>
<tr>
<td>guarded command $g \to S$</td>
<td>conditional interpretation</td>
</tr>
<tr>
<td>if … fi</td>
<td>conditional interpretations with a common codomain</td>
</tr>
<tr>
<td>do … od</td>
<td>conditional interpretations with common domain and codomain</td>
</tr>
</tbody>
</table>
Espec Refinement

\[ x_0, y_0 : \text{Pos} \]
\[ z : \text{Pos} \]

\[ x_0, y_0 \]
\[ z := \text{gcd}(x_0, y_0) \]

\[ \text{ax } z = \text{gcd}(x_0, y_0) \]

\[ x := x_0 \]
\[ y := y_0 \]

\[ x > y \rightarrow x := x - y \]

\[ y > x \rightarrow y := y - x \]

\[ x = y \rightarrow z := x \]

\[ \text{ax } g\text{cd}(x_0, y_0) = g\text{cd}(x, y) \]
Espec Pushout

- pullback of underlying shapes
- pushout of global specs
- pushout of corresponding state specs
- transitions obtained via universality
Composition of Behavior: Mutual Exclusion

Service Spec-1
var u1 : boolean

Service Spec-2
var u2 : boolean

Working-1
u1 := false
Using-1
u1 := true

idle
b := false
Busy
b := true

Resource Invariants
¬(u1 ∧ u2)
b ⇔ (u1 ∨ u2)

Unit Capacity Resource

Abstract User process

Working-2
u2 := false
Using-2
u2 := true

W1, I, W2

U1, B, W2

W1, B, U2

The composed espec exhibits exactly the mutual exclusive behaviors
Colimit of especs

Axioms
1. \( \neg(u_1 \land u_2) \)
2. \( b \Rightarrow (u_1 \lor u_2) \)
3. \( u_1 \Rightarrow b \)
4. \( u_2 \Rightarrow b \)
Composition of Behavior: Mutual Exclusion

Service Spec-1
\[ \text{var } u_1 : \text{boolean} \]

Working-1
\[ u_1 := false \]
Using-1
\[ u_1 := true \]

Idle
\[ b := false \]
Busy
\[ b := true \]

Resource Invariants
\[ \neg (u_1 \land u_2) \]
\[ b \iff (u_1 \lor u_2) \]

Service Spec-2
\[ \text{var } u_2 : \text{boolean} \]

Working-2
\[ u_2 := false \]
Using-2
\[ u_2 := true \]

Abstract User process

Unit Capacity Resource

The composed espec exhibits exactly the mutual exclusive behaviors

Kestrel Institute
Espec Colimit

superposition of transitions

\[ a \to x := 0 \]

\[ b \to y := 1 \]

\[ a \land b \to x, y := 0, 1 \]
Continuous Re-Assembly of a Sensor Network

- Refinement of the logical architecture to mote components reveals unreliable communication
- At design-time, formally compose in active probes, gauges and an adaptivity scheme that at run-time re-architects the system to adapt to communication failures

Initially, motes are randomly strewn at a site

S - sensor
C - collector
Basic Architecture of a Sensor Net:
each sensor transmits data to
a designated data collector
A Simple Adaptive Architecture

heartbeat probe plus clock allows gauge to decide if a connector is alive, and to send adaptation information otherwise
Composing Architectures:
Basic architecture is composed with adaptive architecture

the composition is carried out by the automatic \textit{colimit} operation on diagrams of especs
Result: An Adaptive Sensor Net
Refinement of a Sensor Net Architecture

Sensor Net Architecture

Adaptive Sensor Net Architecture

msg service

Adaptive communication channel

colimit

espec compiler

C code
+ KNAL primitives

C compiler

Mote code under TinyOS

Kestrel Institute
Open Systems Composition

*Abadi-Lamport*

A system $M$, comprised of components $M_1,\ldots,M_n$, guarantees its services if

1. the environment satisfies its requirements $E$

2. $M$’s services follow from the services provided by $M_1,\ldots,M_n$

3. each component $M_i$ guarantees its services assuming that its environment ($E$ + the other components) satisfies $E_i$
Systems Specifications

*parameterization*

The environment model is a *parameter* espec to an espec:

- as an espec, it can specify operations, events, invariants, timing, resource constraints

- as a parameter, the system can exploit its properties and services, but cannot refine or modify them

- the assurance arguments for the system depend on the actual environment implementing the environment model

*i.e. there is a morphism from the model to the environment*
Open System Composition

```
<table>
<thead>
<tr>
<th>Environment 1 spec</th>
<th>Component 1 spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment 2 spec</th>
<th>Component 2 spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>
```
Example: Concurrent Garbage Collection

Given:
1. a finite DAG (directed acyclic graph)
2. some permanently rooted nodes
3. a Mutator process that mutates the graph
4. a Collector process that finds inaccessible nodes and makes them accessible
espec Directed-Rooted-Graph is
  spec
  sort Node
  sort Arc = {source : Node, target : Node}
  sort Directed-Rooted-Graph = { Nodes : set(Node),  Roots : set(Node),  Arcs : set(Arc)
                   | Roots ~= {}  & Roots ⊂ Nodes  & Arcs ⊆ Nodes x Nodes }
  var G : Directed-Rooted-Graph

  function accessible? (G:Directed-Rooted-Graph,  n:Node | n in G.Nodes) : Boolean
     = (n in G.Roots  or ex(m)(<m,n>  in G.Arcs & accessible?(G, m)))
  end-spec

  prog
  procedure CollectNode(n:Node | n in G.Nodes & ~accessible(G,n) ) is
    let r:Node  = some(s :Node)(s in G.Roots)
    G.Arcs := G.Arcs with (r ,n) \ {<n,m> | m in G.nodes}
  postcondition: accessible(G,n)
  end-procedure

  procedure ChangeArc(a:Arc, k:Node | a in G.Arcs & k in G.Nodes & accessible(G,k)) is
    G.Arcs := (G.Arcs without a) with <a.source,k>
  end-procedure

  end-espec

Kestrel Institute
Concurrent Garbage Collection

Theorem (Safety):
No accessible node is ever collected
Axiom (Liveness):
All inaccessible nodes are eventually collected.

Collector
Mutator
Problem Solving Structure

Complement Reduction Structure:
finite superset of feasible solutions plus a test for nonsolutions
* supports sieves

finite superset of feasible solutions
Sieve Example: Garbage Collection

Given:
1. a finite collection of nodes, each with some links to other nodes
2. some permanently rooted nodes

Find: all nodes inaccessible from the rooted nodes

Sieve interpretation
Finite superset of solutions $\mapsto$ all nodes
non-solutions $\mapsto$ accessible nodes

Sieve algorithm scheme
1. mark all nodes white
2. mark all accessible nodes black
3. collect the remaining white nodes

Accessibility:
$n \in G.roots \Rightarrow acc(n)$
$acc(n) \wedge \langle n, m \rangle \in G.arcs \Rightarrow acc(m)$

Primality:
$prime?(2)$
$prime?(n) \wedge plural?(i) \Rightarrow \neg prime?(n\times i)$
Using a Design Theory and Colimit to Construct a Refinement

Sieve theory

→

Garbage Collection Specification

Sieve Algorithm

→

Sieve Algorithm for Garbage Collection

colimit
Open System Composition

- Environment 1
  - Spec
  - Component 1
    - Spec
  - P

- Environment 2
  - Spec
  - Component 2
    - Spec
  - P
A Road Map – Specification Expressiveness

Basic Results
- composition
- refinement
- design theories
- automation

HO Logic → Behavioral
- Functional Lisp, C, Java
- Imperative Lisp, C

Resource Modeling
- Schedulers

Behavioral + Resource Modeling
- RT/Embedded Systems

Continuous Vars
- Hybrid Systems
A Road Map – Specification Expressiveness

- **HO Logic**
  - Functional
    - Lisp, C, Java
- **Behavioral**
  - Imperative
    - Lisp, C
- **Object-Oriented Modeling**
  - Idiomatic Java
- **Stochastic Processes**
  - Hybrid Systems
Extras
Calculating a Colimit in SPEC

spec BINARY-RELATION is
sort E
  op _br_: E, E → Boolean
end-spec

spec TRANSITIVE -RELATION is
sort E
  op _tr_: E, E → Boolean
  axiom transitivity is
    a tr b ∧ b tr c ⇒ a tr c
end-spec

spec REFLEXIVE-RELATION is
sort E
  op _rr_: E, E → Boolean
  axiom reflexivity is
    a rr a
end-spec

spec PREORDER-RELATION is
sort E
  op ≤: E, E → Boolean
  axiom reflexivity is
    a ≤a
  axiom transitivity is
    a ≤b ∧ b ≤c ⇒ a ≤c
end-spec
Calculating a Colimit in **SPEC**

Collect equivalence classes of sorts and ops from all specs in the diagram.

**BINARY-RELATION**

**REFLEXIVE-RELATION**

**TRANSITIVE-RELATION**

**PREORDER-RELATION**
Hybrid Especs
modeling continuous behavior

fuel pump service

<table>
<thead>
<tr>
<th>var</th>
<th>$l$ : real</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>$l_{\text{max}}$ : real</td>
</tr>
<tr>
<td>ax</td>
<td>$l \in [0, l_{\text{max}}]$</td>
</tr>
</tbody>
</table>

import fuel pump service

fuel pump

<table>
<thead>
<tr>
<th>pumping</th>
</tr>
</thead>
</table>
| $ax$ $l' = -5$
| $l' := -5$ |
| not pumping |
| $ax$ $l' \in [0, 100]$ |
| $l' := 0$ |

<table>
<thead>
<tr>
<th>idling</th>
</tr>
</thead>
</table>
| $ax$ $l' = 0$
| $l' := 0$ |

<table>
<thead>
<tr>
<th>refilling</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ax$ $l' \in [0, 100]$</td>
</tr>
<tr>
<td>$l' := 0$</td>
</tr>
</tbody>
</table>
Hybrid Espects
modeling continuous behavior

vehicle fuel reqt

| var f : real |
| const f_max : real |
| ax f ∈ [0, f_max ] |

fueling connector

import vehicle fuel reqt
import fuel pump service

fueling

| ax f’ ∈ [0, 10 ] |

not fueling

| ax f’ ∈ [-20, 0 ] |

fuel pump service

| var l : real |
| const l_max : real |
| ax l ∈ [0, l_max ] |

pumping

| ax l’ = -5 |
| l’ := -5 |

not pumping

| ax l’ ∈ [0, 100 ] |

import fuel pump service

| ax l’ ∈ [0, 100 ] |