Specware Technologies

Douglas R Smith

Kestrel Institute and Kestrel Technology
Palo Alto, California
www.kestrel.edu
www.kestreltechnology.com
Code Generation by Refinement

Requirements

\[ \text{semiformal link} \]

Specification

\[ \text{Property-preserving refinements from design theories} \]

Key ideas
- specifications
- refinement
- design theories
- composition

Code
Specifications and Morphisms/Interpretations

spec **Partial-Order** is
    type 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
    type Int
    op ≤: Int, Int → Boolean
    op 0 : Int
    op _+_ : Int, Int → Int
    ...
end-spec

**Specification morphism:** a language translation that preserves provability

le(x,x) translates to x ≤ x
Software Development by Refinement

Code generation is accomplished via a logic morphism from SPEC to the logic of a programming language.
Specification Language: MetaSlang

• types:
  – products: $P, Q$
  – coproducts: $P+Q$
  – function sorts: $P \rightarrow Q$
  – subtypes defined using predicates: $P|I$
  – quotients defined using equivalence relations : $P/\equiv$
  – type axioms: $\text{Even-integers} = \text{Integer} | \text{even}?$
  – polymorphic types

• function signatures

• optional definitions, using patterns

• higher-order axioms and theorems

• executable subset similar to ML
A = spec
  type Even = Nat | even?
def even? n = (n div 2 = 0)
  op f : Even \rightarrow Boolean
  \ldots f(expr) \ldots
  axiom fa(n:Even) f(n) \Rightarrow n>0
  \ldots

\begin{align*}
  \begin{cases}
    \text{Even} \leftrightarrow E \\
    f \leftrightarrow g \\
    \ldots
  \end{cases}
\end{align*}

B = spec \ldots

\begin{align*}
  A \quad \text{even? is a well-defined function}\\
  A, \text{ context} & \quad \text{even?}(expr)\\
  B \quad g : E \rightarrow Boolean\\
  B \quad \text{fa(n:E)} \ g(n) \Rightarrow n>0
\end{align*}
Assurance Aim

Let $S_0 \rightarrow S_1 \rightarrow \cdots \rightarrow S_n$ be a derivation.

If (1) the proof obligations generated for each spec $S_i$ \(i=0,1,\ldots,n\) are provable and (2) the proof obligations generated for each morphism are provable and (3) the translation to executable code preserves the definitions

then (1) the executable code terminates on all legal inputs and
(2) the code computes functions that satisfy the specified properties in $S_0$.

Concerns:
- correctness of the code generators and compilers
- correctness of underlying computation substrate
Composing Specifications: the Colimit operation

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

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

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

spec PREORDER-RELATION is
  type 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**: $E$ with $br$.
- **Transitive Relation**: $E$ with $tr$.
- **Reflexive Relation**: $E$ with $rr$.
- **Preorder Relation**: $\{E, E, E\}$ renamed $E$; $\{br, rr, tr\}$ renamed $\leq$.

Axioms:
- For binary relation: $x tr y \land y tr z \Rightarrow x tr z$.
- For preorder relation: $x \leq x$, $x \leq y \land y \leq z \Rightarrow x \leq z$. 


Structure of a Specification for Scheduling

\[
\text{Time, Quantity} \quad \text{Resource} \quad \text{Task} \quad \text{Reservation} = \text{Resource} \times \text{Task} \times \text{Time} \quad \text{1-Sort} \\
\text{Schedule} = \text{Set(Reservation)} \quad \text{Set} \\
\text{Scheduler}
\]
Structuring a Spec via Colimits

Spec 1-SORT
  type E

Spec LINEAR ORDER
  type L
  op ≤: L,L → Boolean
  ...

Spec TIME
  type Time
  op time-le
  ...

{L→Time, ≤→ time-le}

coproduct

TIME + QUANTITY

Spec LINEAR ORDER

LINEAR ORDER
  op ≤

GROUP

po

LINEAR ORDER + GROUP

extend

LINEARLY ORDERED GROUP

rename

QUANTITY

Spec TIME

type Time

op time-le

extend

TIME + QUANTITY
Constructing Refinements

1. Library of Refinements

Global Search Algorithm → Set → Sequence → Resource
Global Search

2. Library of Refinement Generators

- Rewrite Simplification
- Context-dependent Simplification
- Finite Differencing
- Case Analysis
- Partial Evaluation

Global Search Algorithm → Scheduling\(_0\) → Scheduling\(_1\) → Scheduling\(_2\) → Context-dependent Simplification → Scheduling\(_3\) → Finite Differencing → Scheduling\(_4\) → Set → Sequence
Planware: Synthesis of High PerformanceSchedulers

- Model Construction Tool
  - Library of resource and task models
  - TPFDD: Strategic Airlift, Aircrews, Fuel tracking, MOG

- Model of Scheduling Problem
- Planware Scheduler Generator
  - Customized Scheduler (in MetaSlang)
  - Optimization and Code Generation
    - Customized Scheduler (in CommonLisp)

- Customized Scheduler
  - TPFDD data, Resource data
  - Schedule

- 890 lines
- 20,000 LOC
- 88,000 LOC
Java Card Applet Generator

Domain-specific language (DSL)

- domain = smart cards
  - ISO 7816 commands/responses
  - cryptography
  - personal identification numbers
  - ...

(applet spec) \rightarrow (automatic) \rightarrow applet code

- productivity
- high assurance
Independent Certification

applet spec → GENERATOR → applet code

proof → CHECKER → yes/no
FORGES: Stateflow to C

Compiler based on a partial evaluator constructed with stepwise refinement
Results

“The surprising result for us and Kestrel was the quality and size of the code generated. It has taken both dSpace and the MathWorks many years to develop their respective code generation tools. Kestrel took less than two years. In addition, because it is based on an analytic approach to generating the code generator, it is relatively easy to extend the supported Stateflow language and create a new code generator. We believe this approach is extremely promising and hope that commercial tool vendors will take notice.”

— Bill Milam, Ford Research
State Machine Foundations in Specware

1. Nature of State Machines and behavior
   - discrete systems
   - communication protocols
   - hybrid systems
   - resource systems
⇒ nodes represent activities and invariant structure

2. Systems Specification and Design
   - contravariance of system versus environment
   - system parameter as requirements on environment

   ![Diagram showing the relationship between environment specifications and agents with parameters and services](image)
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

\[
\begin{array}{ccc}
A & 5 & 8 & 1 \\
A & 5 & 8 & 4 \\
\end{array}
\]

\[
\text{true} \rightarrow A(3) := 4
\]
Evolving specifications (especs)

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)!

\[
x := e
\]

pre(x) \rightarrow post(x)

correctness condition:
\[ pre(x) \vdash post(e) \]

spec before is
\[
\ldots
\text{var } x : \ldots
\text{ax } pre(x)
\ldots
\text{end-spec}
\]

\{ e \leftarrow x \}

spec after is
\[
\ldots
\text{var } x : \ldots
\text{ax } post(x)
\ldots
\text{end-spec}
\]
Especs, states, and computation

Base

One ↔ Loop ↔ Two

global spec
extends to
abstract state specs
denotes
states/models
Guarded Commands

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

is represented as the compound arrow:

\[
\text{spec} \ A \ is \ \\
\quad \ldots \ \\
\quad \text{var} \ x : \ldots \ \\
\quad \text{ax} \ \text{pre}(x) \ \\
\quad \ldots \ \\
\quad \text{end-spec} \ \\
\]

\[
\text{spec} \ AB \ is \ \\
\quad \text{import} \ A \ \\
\quad \text{ax} \ g(x) \ \\
\quad \text{end-spec} \ \\
\]

\[
\text{spec} \ B \ is \ \\
\quad \ldots \ \\
\quad \text{var} \ x : \ldots \ \\
\quad \text{ax} \ \text{post}(x) \ \\
\quad \ldots \ \\
\quad \text{end-spec} \ \\
\]

\[
\{ e \leftrightarrow x \} \ \\
\]
Accord Specs and Refinement

\[ B_{\text{abs}} = \langle \text{spec}_{\text{abs}}, \text{behavior}_{\text{abs}} \rangle \]

\[ B_{\text{con}} = \langle \text{spec}_{\text{con}}, \text{behavior}_{\text{con}} \rangle \]
Espec Refinement

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

\[ x, y, z \in \text{Pos} \]

\[ \text{spec refinement} \]

\[ \text{behavior refinement (simulation)} \]

\[ x_0, y_0 \rightarrow \text{skip} \]
\[ x_0, y_0 \rightarrow y := y - x \]
\[ x_0, y_0 \rightarrow x := x - y \]
\[ x, y, z \rightarrow z := \gcd(x_0, y_0) \]
\[ x, y, z \rightarrow x := y - x \]
\[ x, y, z \rightarrow x := x - y \]

\[ \text{ax } z = \gcd(x_0, y_0) \]
\[ \text{ax } \gcd(x_0, y_0) = \gcd(x, y) \]
Parametric Accord Specs and Refinement

\[ P_{\text{abs-p}} = \langle \text{spec}_{\text{abs-p}}, \text{behavior}_{\text{abs-p}} \rangle \xrightarrow{p} B_{\text{abs}} = \langle \text{spec}_{\text{abs}}, \text{behavior}_{\text{abs}} \rangle \]

\[ P_{\text{con-p}} = \langle \text{spec}_{\text{con-p}}, \text{behavior}_{\text{con-p}} \rangle \xrightarrow{p} B_{\text{con}} = \langle \text{spec}_{\text{con}}, \text{behavior}_{\text{con}} \rangle \]
Refinement Theorem

If \( A \rightarrow B \)
then every run/trace of B maps to a run/trace of A;
i.e. \( \text{traces}(B) \subseteq \text{traces}(A) \).

but, does B behave like A in all environments?

This theorem suffices for the case of showing that
a computation satisfies an abstract property,
but more is needed to model computational refinement.
Computational Refinement Theorem

If A refines to B as in the figure and progress conditions are satisfied then \( \text{traces}(B) \subseteq \text{traces}(A) \)
and for every trace of A from initial state \( a_0 \)
there is a trace of B from an initial state \( b_0 \)
that maps to \( a_0 \)
\( i.e. \) for every environment in which A behaves properly,
so does B
System Composition Problem

Specify and compose a system comprised of a
1. mission-controller component
2. radar unit
3. communication channel
Mission-Controller

- Requests radar images frequently
- Requires a 5ms response time at most

\[ \text{MC-Env} = \]
\[ \text{event RadarRequest : MC-Radar-Parameters} \]

\[ \text{Mission-Controller} = \]
\[ \text{event RadarResult : MC-Radar-Response} \]
\[ \text{var res : MC-Radar-Response} \]
\[ \text{var parms : MC-Radar-Parameters} \]
\[ \text{var mc1 : Clock = 0} \]

\[ \text{mc1 := 0,} \]
\[ !\text{radarRequest(parms)} \]
\[ \text{mc1 \leq 5ms} \]
\[ \text{?\text{radarResult(res)}} \]
Radar Component

- Requires a minimum separation of request of 1ms (i.e. 1000Hz max rate)
- Offers a 0.5ms maximum response time

\[
\text{Radar-Env} = \\
\text{event RadarInfo : Radar-Response}
\]

\[
\text{Radar} = \\
\text{import Radar-Env} \\
\text{event GetSignal : Radar-Parameters} \\
\text{var radar-parms : Radar-Parameters} \\
\text{var radar-result : Radar-Response} \\
\text{vars rc1, rc2 : Clock = 0}
\]

1ms \leq rc1 \rightarrow \\
rc1 := 0, rc2 := 0, \\
?\text{GetSignal(radar-parms)}

Axiom: \text{rc2 \leq 0.5ms}

!\text{RadarInfo(radar-result)}
Communication Channel/Connector

- Handles messages at rates up to 500kHz
- Offers a 0.001ms one-way transmission time

```
CC-Env1 =
output event out1 : Out1

CommunicationChannel =
import CC-Env1, CC-Env2
input event in1 : In1
input event in2 : In2
var m : In1
var n : In2
const durRP : Time
vars c1, c2, c3 : Clock

CC-Env2 =
output event out2 : Out2
```

A

0.002ms + durRP ≤ c1 →
c1 := 0,
?in1(m)

B

c1 ≤ .001ms

c2 := 0,
!out1(glue1(m))

c2 ≤ durRP

C

c3 := 0,
?in2(n)

c3 ≤ .001ms

D

!out2(glue2(n))
System Composition Diagram
Mission-Control-System =
  event RadarRequest : MC-Radar-Parameters
  event RadarResult : MC-Radar-Response
  event GetSignal : Radar-Parameters
  event RadarInfo : Radar-Response
  var parms : MC-Radar-Parameters
  var rr : Radar-Response
  op glue1 : MC-Radar-Parameters \to\ Radar-Parameters
  op glue2 : Radar-Response \to\ MC-Radar-Response
  vars mc1, c1, c2, c3, rc1, rc2 : Clock

radarResult(glue2(rr))

axiom c3 \leq .001ms
axiom mc1 \leq 5ms

\text{c3} := 0,
RadarInfo(rr)

0.502ms \leq c1 \to
\text{c1} := 0, \text{mc1}:=0,
radarRequest(parms)

axiom c1 \leq .001ms
axiom mc1 \leq 5ms

1ms \leq rc1 \to
\text{rc1} := 0, \text{rc2} := 0, \text{c2} := 0,
getSignal(glue1(parms))

axiom c2 \leq .5ms
axiom mc1 \leq 5ms
axiom rc2 \leq .5ms
# Functional versus Behavioral Specifications

<table>
<thead>
<tr>
<th>functional specifications</th>
<th>behavioral specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>types</td>
<td>classes</td>
</tr>
<tr>
<td>functions</td>
<td>procedures</td>
</tr>
<tr>
<td>axioms</td>
<td>axioms</td>
</tr>
<tr>
<td>Specification units</td>
<td>Module units</td>
</tr>
</tbody>
</table>

- **Product, Sum, Function, Subtype, Quotient**
- **Axioms**
- **Import, Parameterize, Refine, Compose by Colimit**
Example: Points and Pixels

<table>
<thead>
<tr>
<th>Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>float x,y;</td>
</tr>
<tr>
<td>void clear() {x = 0; y=0}</td>
</tr>
<tr>
<td>void move(x:float, y:float){ ... }</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

extends

<table>
<thead>
<tr>
<th>Pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color color;</td>
</tr>
<tr>
<td>void clear() {super.clear(); color = null}</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Classes, Inheritance, Implementations

Overriding is not semantically acceptable in Specware
Class Refinement

Overriding is not semantically acceptable in Specware
Issue: How to Handle Nonfunctional and Cross-Cutting Concerns wrt Composition and Refinement?

A concern is \textit{cross-cutting} if its manifestation cuts across the dominant hierarchical structure of a program/system.

\textbf{Examples}

\begin{itemize}
\item Log all errors that arise during system execution
\item Enforce a system-wide error-handling policy
\item Disallow unauthorized data accesses
\item Enforce timing and resource constraints on a system design
\end{itemize}
Policy Enforcement Approach

where and how does the policy apply?

Policy Conditions → System
Policy Constraint → Refined System

showing where the policy applies requires sound static analysis

What does the policy prescribe?
Security Design Patterns

“Design Patterns capture the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces.”

Design Pattern for Protected Systems
aka Reference Monitor

Client, Resource, policy axiom

Client, Guard, Policy, Resource

Local Network with Users, Personnel Database

Local Network with Access Control on Personnel Database