Specware Technologies

Douglas R Smith

Kestrel Institute and Kestrel Technology
Palo Alto, California
www.kestrel.edu
www.kestreltechnology.com

Code Generation by Refinement

Requirements

semiformal link

Specification

...

Property-preserving refinements from design theories

Code

Key ideas

- specifications
- refinement
- design theories
- composition

Specifications and Morphisms/Interpretations

spec **Partial-Order** is
type E
op le: E, E $\rightarrow$ Boolean
axiom reflx is le(x,x)
axiom trans is le(x,y) $\wedge$ le(y,z) $\Rightarrow$ le(x,z)
axiom antis is le(x,y) $\wedge$ le(y,x) $\Rightarrow$ x = y
end-spec

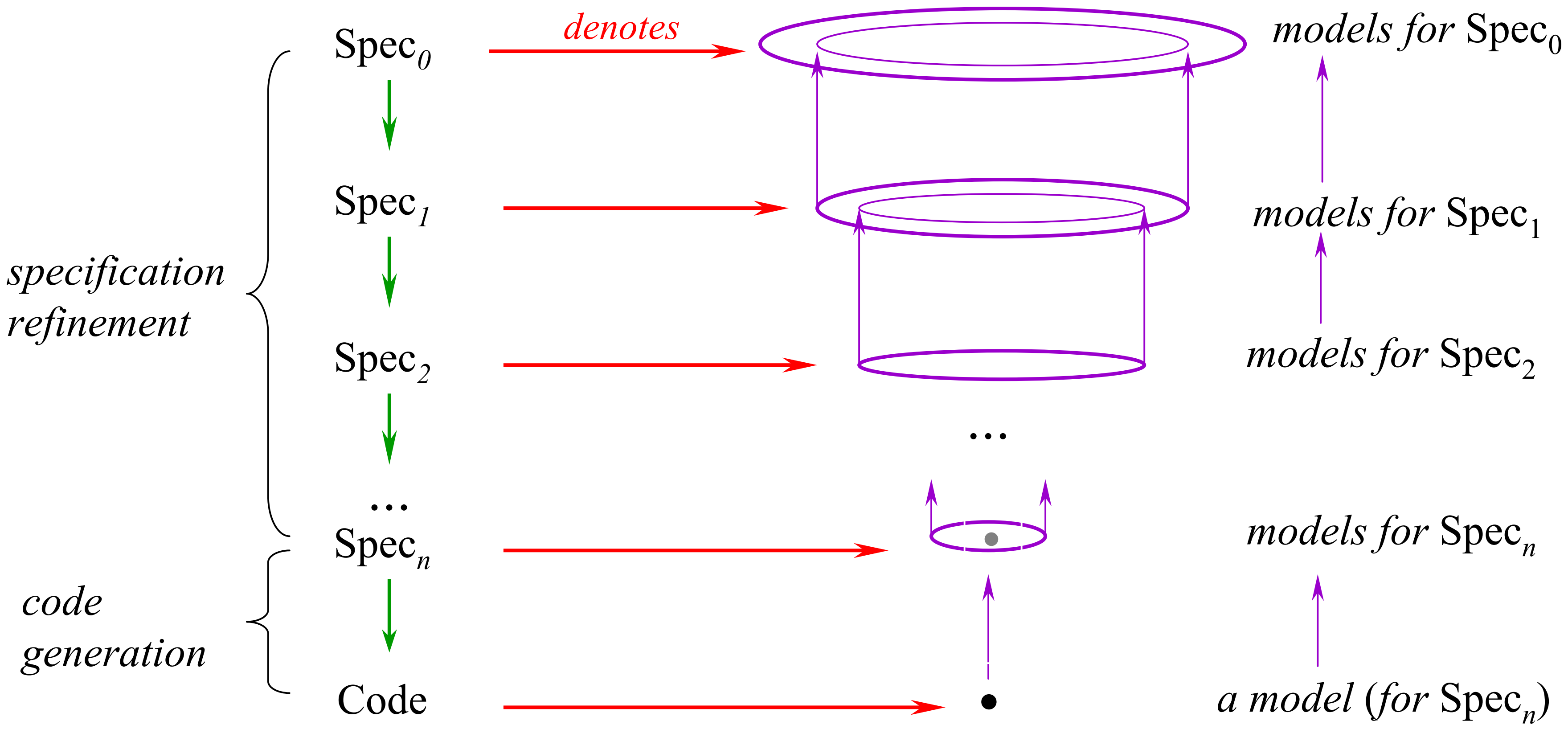
E $\mapsto$ Int
le $\mapsto$ $\leq$
axioms $\mapsto$ thms

spec **Integer** is
type Int
op $\leq$: Int, Int $\rightarrow$ Boolean
op 0 : Int
op _+_ : Int , Int $\rightarrow$ Int
…
end-spec

Specification morphism: a language translation that preserves provability

le(x,x) translates to $\mathbf{x} \leq \mathbf{x}$

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: *Even-integers* = *Integer* | *even?*
  - polymorphic types
- function signatures
- optional definitions, using patterns
- higher-order axioms and theorems
- executable subset similar to ML

Proof Obligations

A = spec
  type Even = Nat | even?
  def even? n = (n div 2 = 0)
  op f : Even → Boolean
  … f(expr) …
  axiom fa(n:Even) f(n) ⇒ n>0
  …

A □ even? is a well-defined function

A, context □ even?(expr)

↓

{ Even ↦ E
f ↦ g
… }

B □ g : E → Boolean

B □ fa(n:E) g(n) ⇒ n>0

B = spec …

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 $i=0,1,..,n$ 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 $\_br\_ : E, E \to Boolean$
end-spec

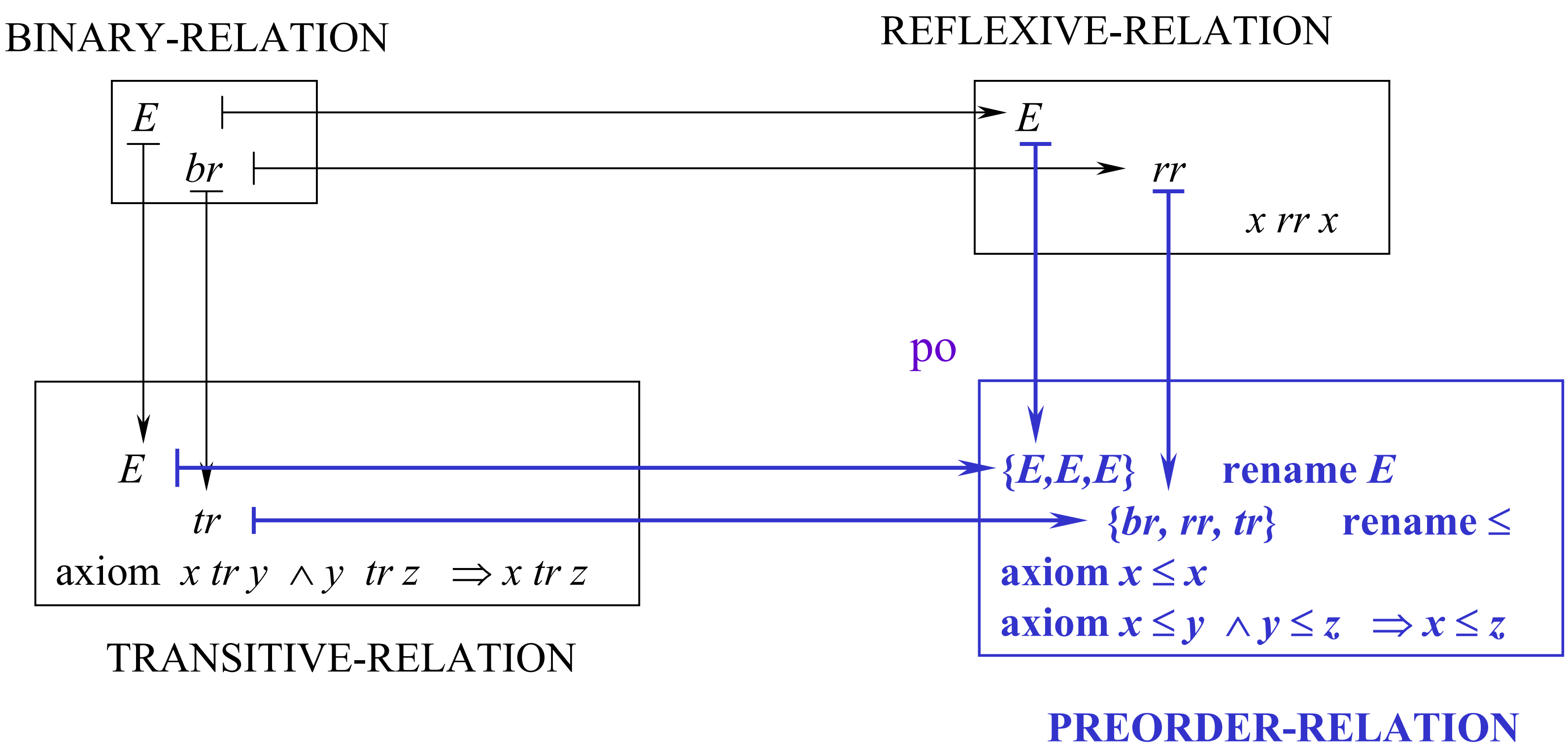
spec REFLEXIVE-RELATION is
  type E
  op $\_rr\_ : E, E \to Boolean$
  axiom reflexivity is $a\ rr\ a$
end-spec

spec TRANSITIVE -RELATION is
  type E
  op $\_tr\_ : E, E \to$ Boolean
  axiom transitivity is
  $a\ tr\ b \wedge b\ tr\ c \Rightarrow a\ tr\ c$
end-spec

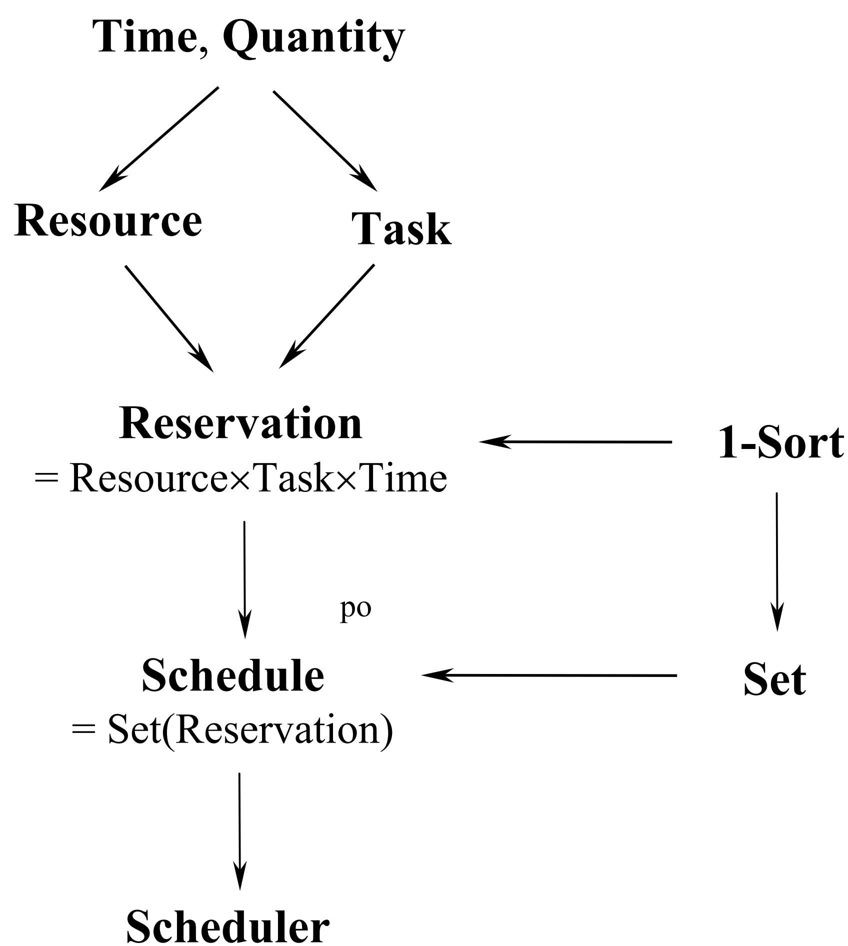
spec PREORDER-RELATION is
  type E
  op $\leq : E, E \to Boolean$
  axiom reflexivity is
  $a \leq a$
  axiom transitivity is
  $a \leq b \wedge b \leq c \Rightarrow a \leq c$
end-spec

Calculating a Colimit in **SPEC**

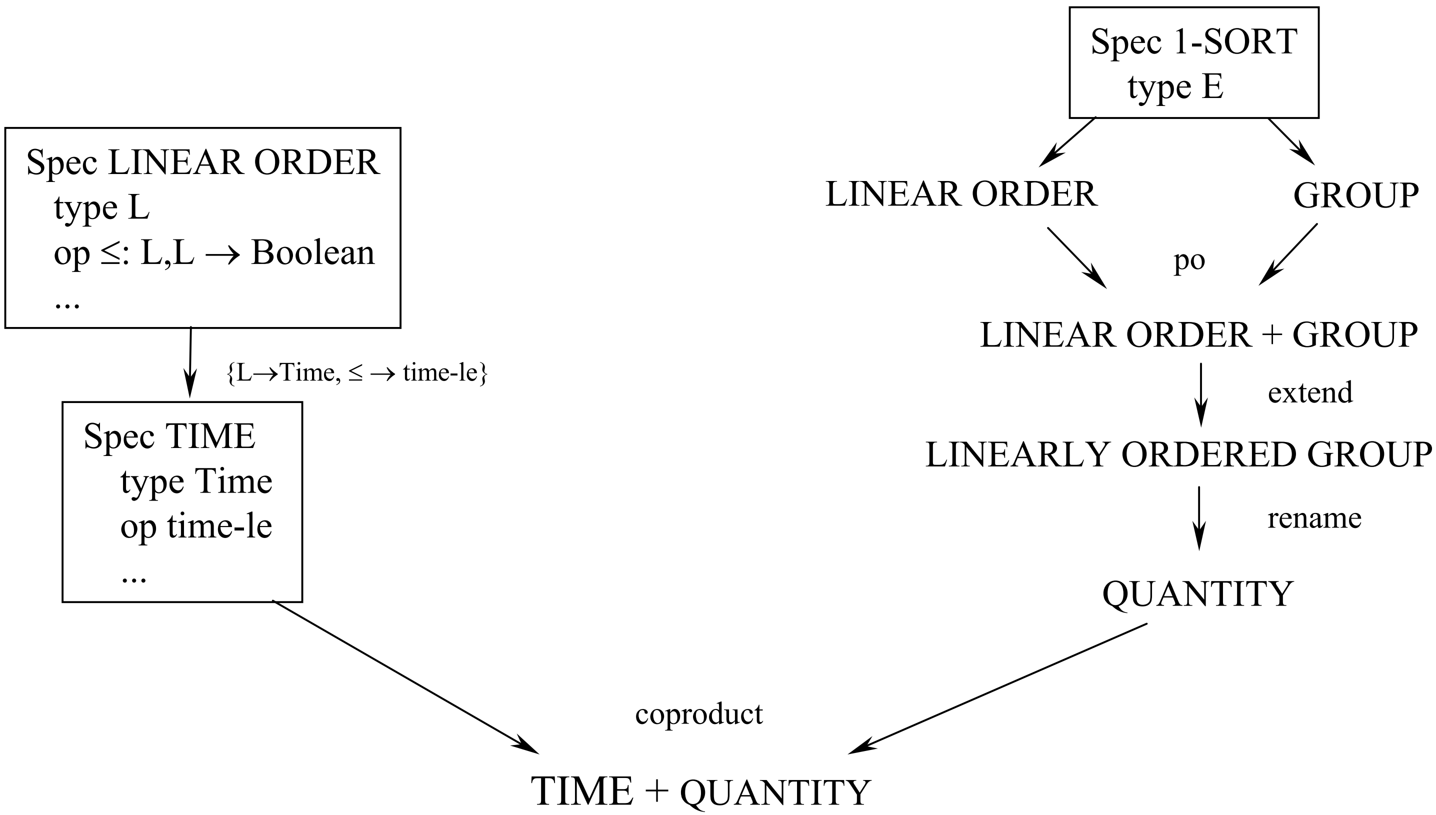
Collect equivalence classes of sorts and ops from all specs in the diagram.



Structure of a Specification for Scheduling

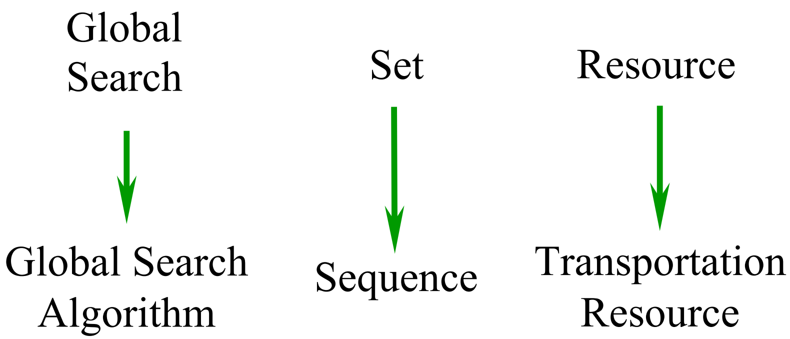


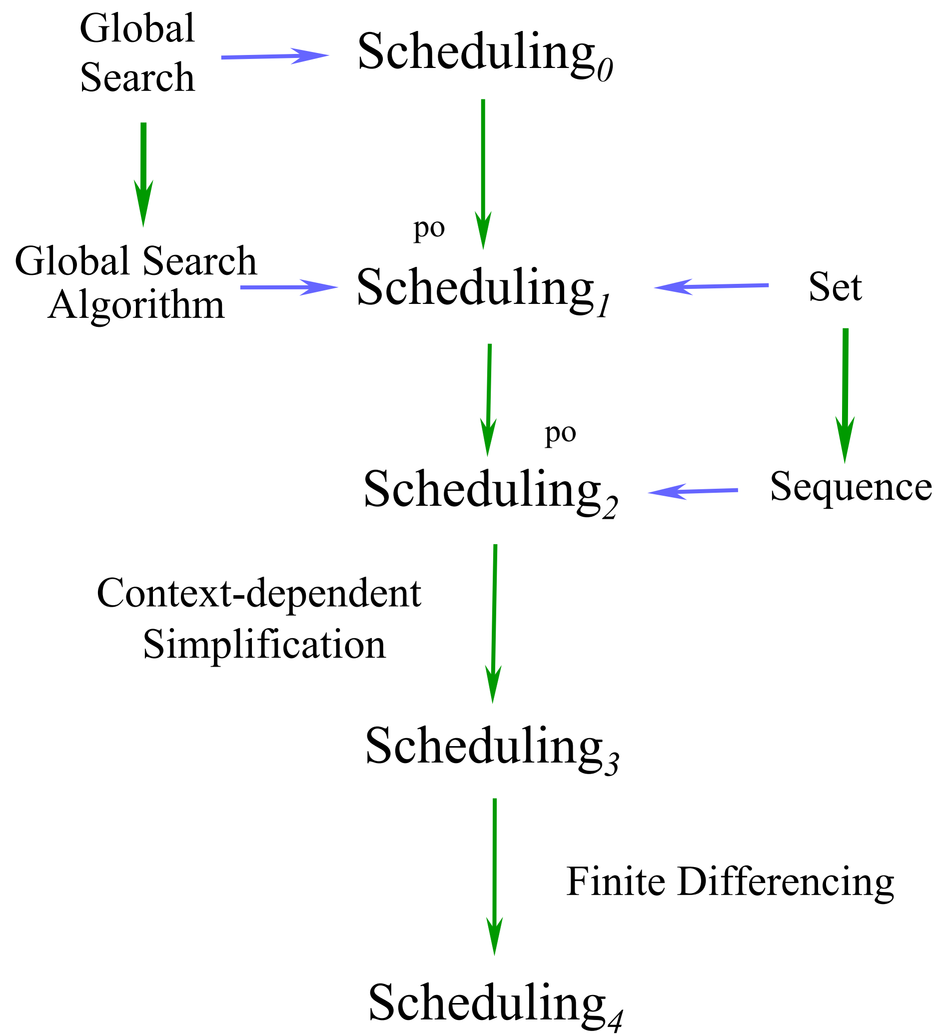
Structuring a Spec via Colimits



Constructing Refinements

1. Library of Refinements

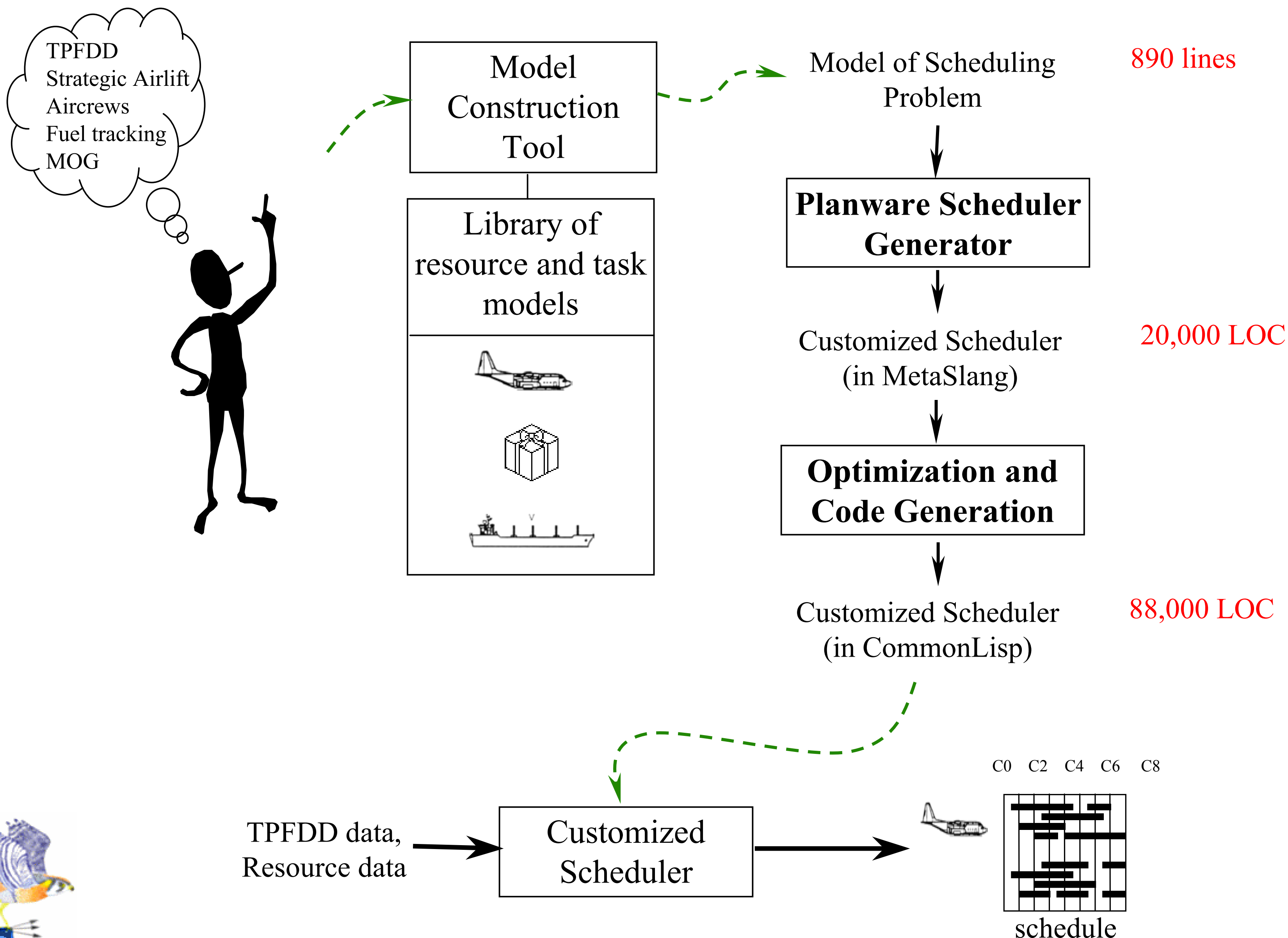




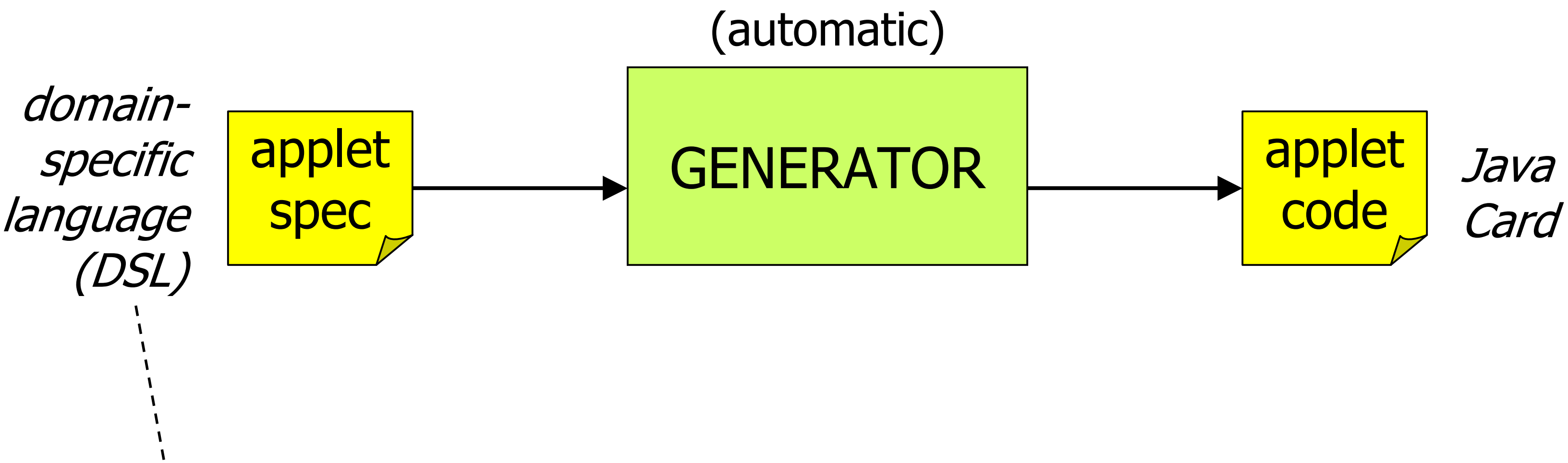
2. Library of Refinement Generators

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

Planware: Synthesis of High Performance Schedulers



Java Card Applet Generator

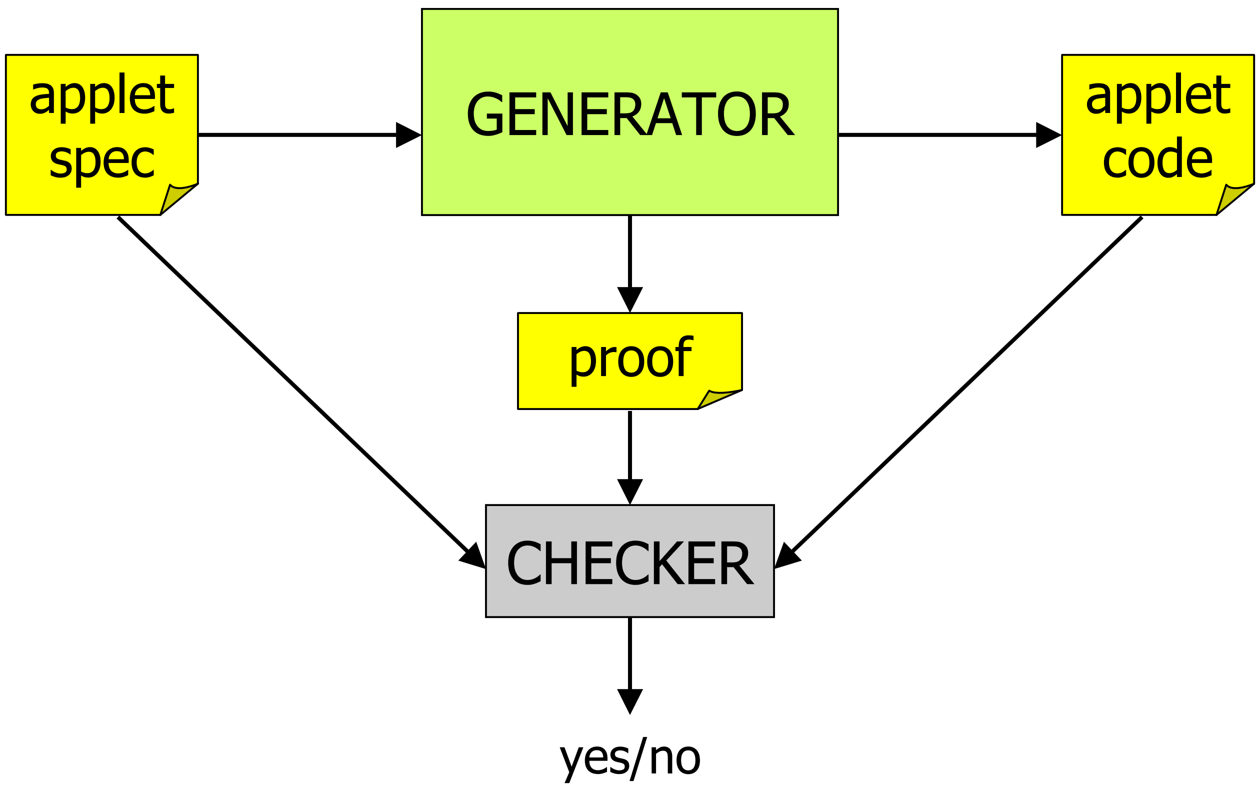


domain = smart cards

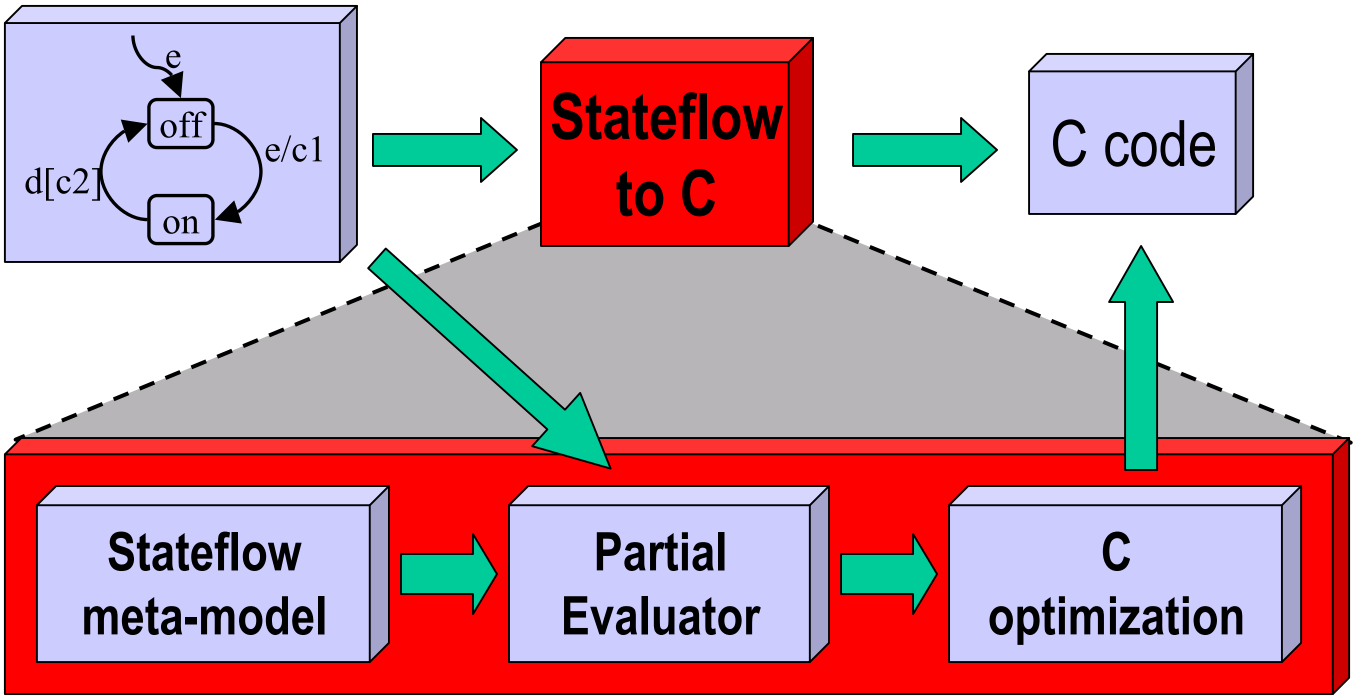
- ISO 7816 commands/responses
- cryptography
- personal identification numbers
- …

- productivity
- high assurance

Independent Certification



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

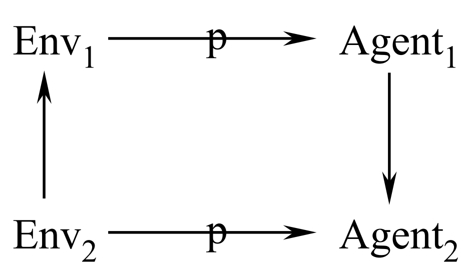
State Machine Foundations in Specware

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

parameter,
environment spec,
required services



body,
system/component/agent spec,
offered services

Evolving specifications (especs)

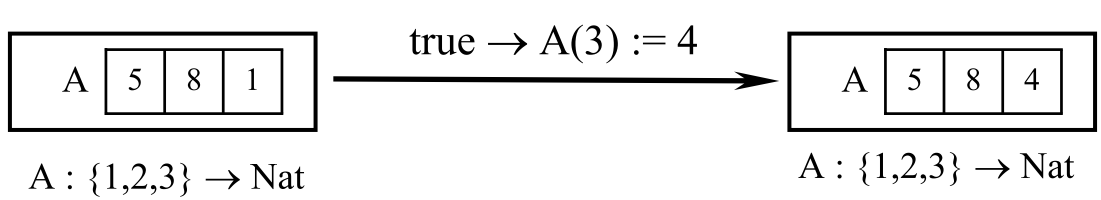
Key ideas that link state machine concepts with logical concepts

1. States are models (structures satisfying axioms)

State	Model
datatypes	sets
variables	functions, values
properties	axioms, theorems

2. State transitions are finite model changes

Example: Updating an array/finite-function A

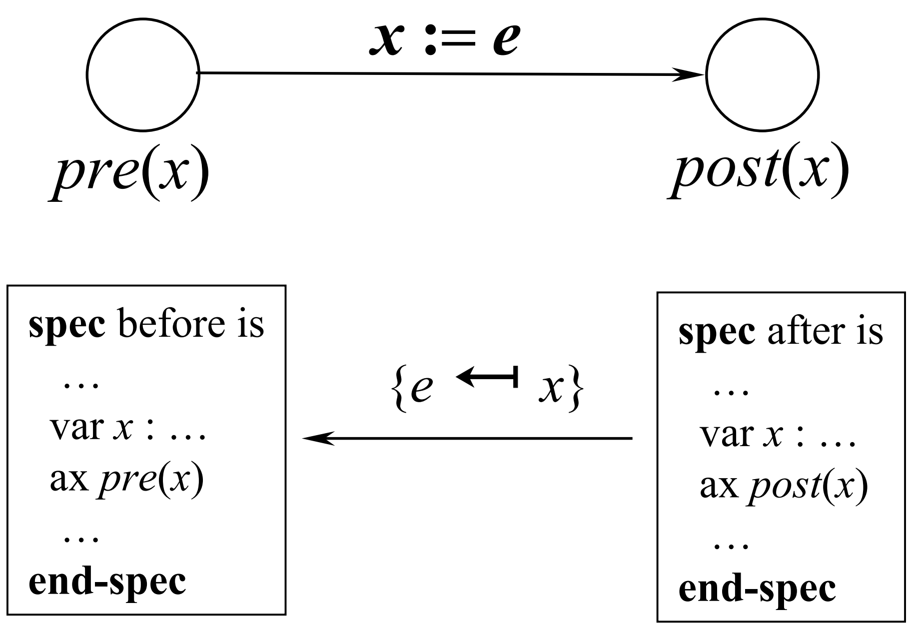


Evolving specifications (especs)

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

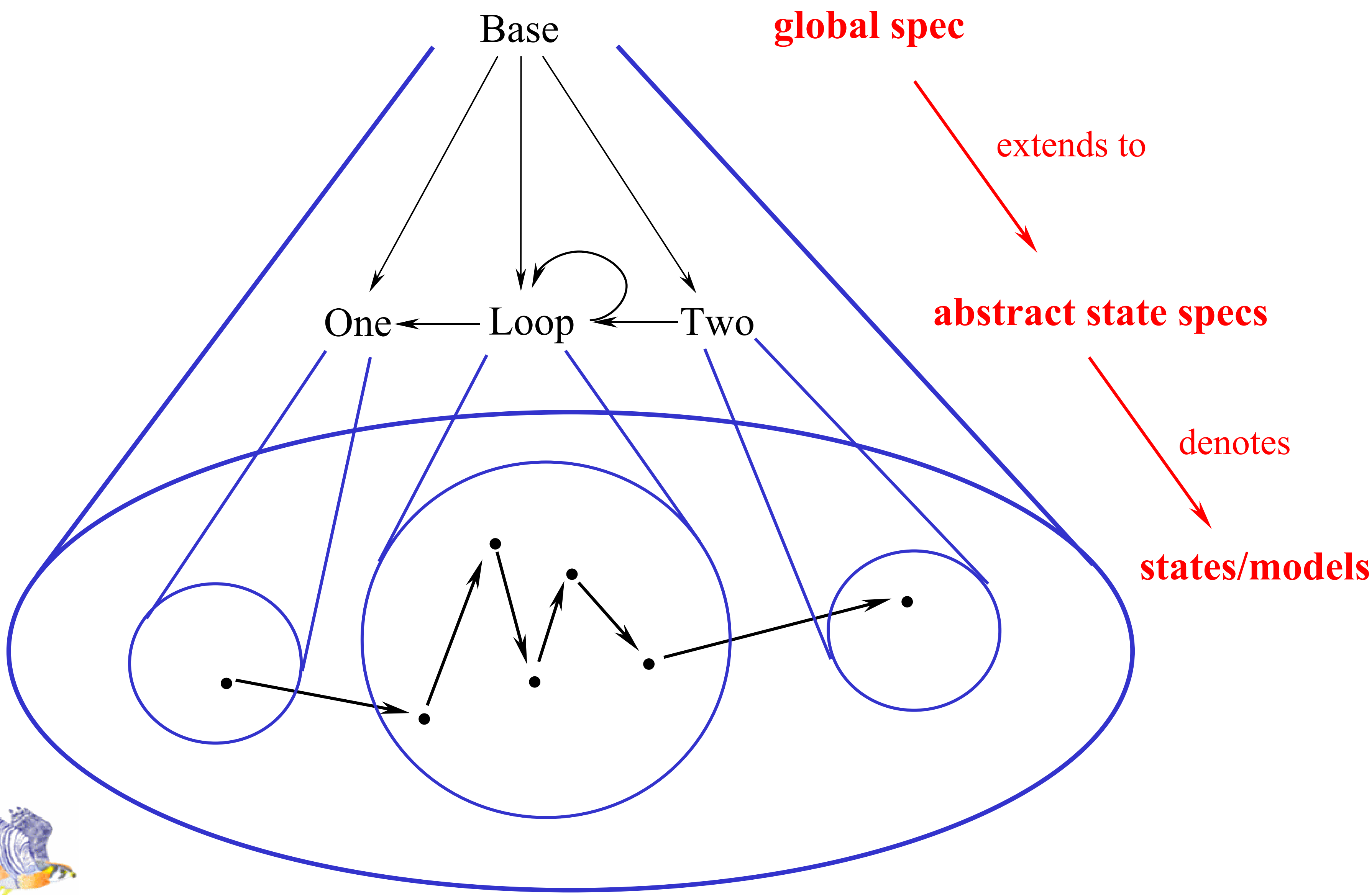
   Specs represent abstract states

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

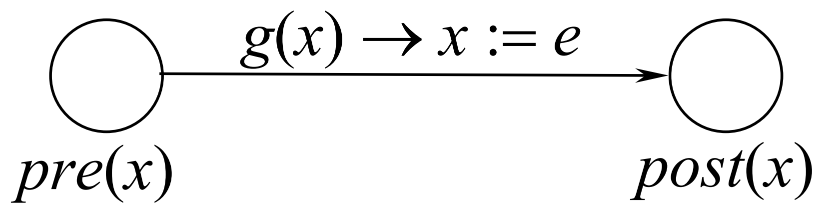


correctness condition:
$pre(x) \vdash post(e)$

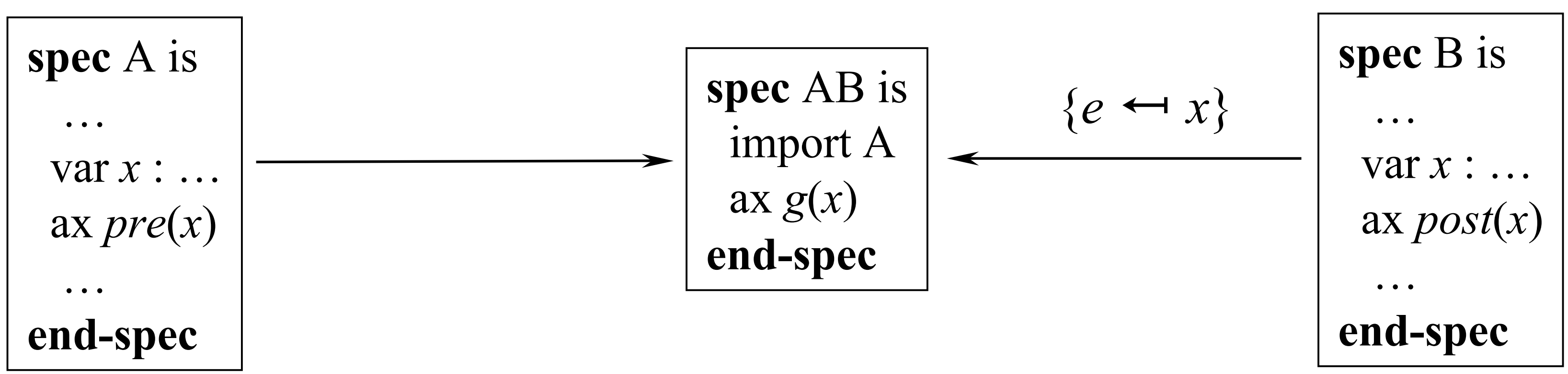
Especs, states, and computation



Guarded Commands

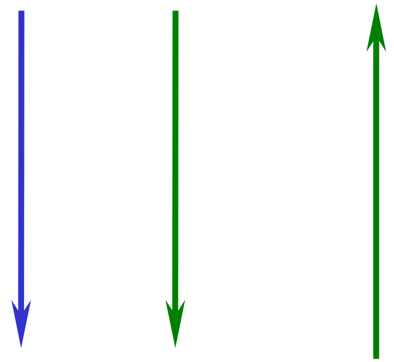


is represented as the compound arrow:



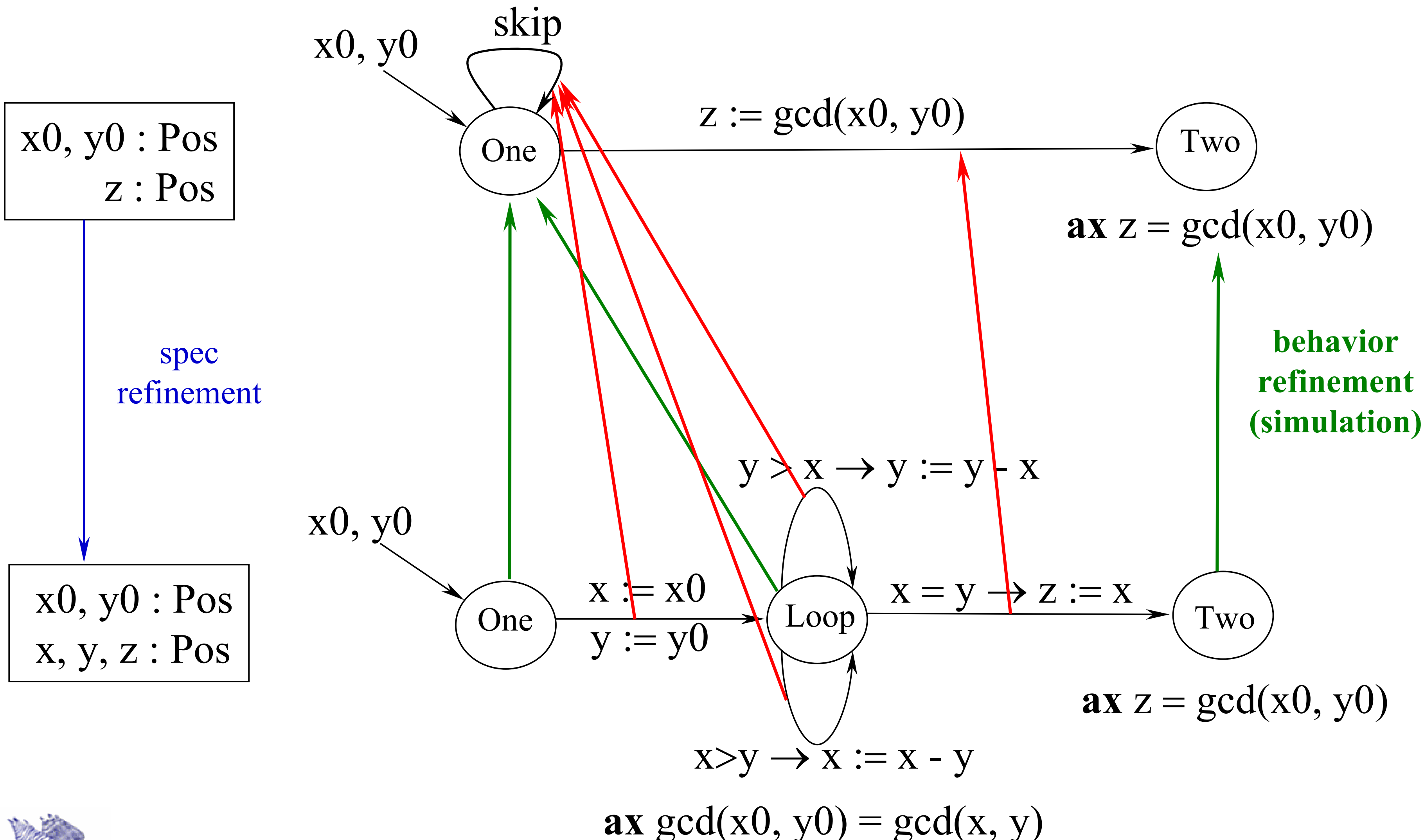
Accord Specs and Refinement

$$B_{abs} = \langle\, spec_{abs} ,\, behavior_{abs} \,\rangle$$



$$B_{con} = \langle\, spec_{con} ,\, behavior_{con} \,\rangle$$

Espec Refinement



Parametric Accord Specs and Refinement

$$P_{abs\text{-}p} = \langle\, spec_{abs\text{-}p}, behavior_{abs\text{-}p} \,\rangle \xrightarrow{\;p\;} B_{abs} = \langle\, spec_{abs}, behavior_{abs} \,\rangle$$

$$P_{con\text{-}p} = \langle\, spec_{con\text{-}p}, behavior_{con\text{-}p} \,\rangle \xrightarrow{\;p\;} B_{con} = \langle\, spec_{con}, behavior_{con} \,\rangle$$

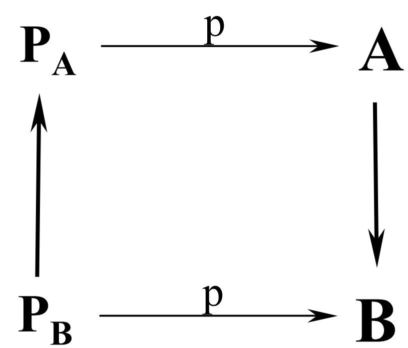
Refinement Theorem

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

and for every trace of A from initial state a_0

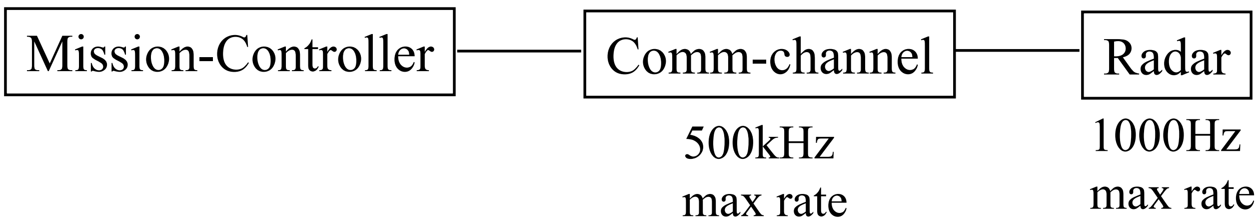
there is a trace of B from an initial state b_0

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

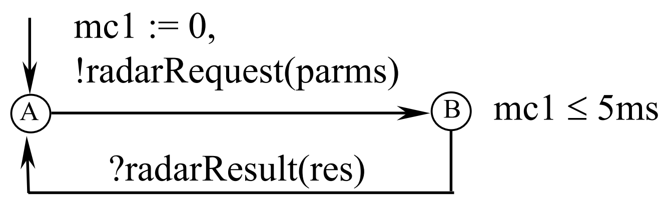
- Requests radar images frequently
- Requires a 5ms response time at most

MC-Env =
event RadarRequest : MC-Radar-Parameters

required service of the environment

Mission-Controller =
event RadarResult : MC-Radar-Response
var res : MC-Radar-Response
var parms : MC-Radar-Parameters
var mc1 : Clock = 0

offered service of the component

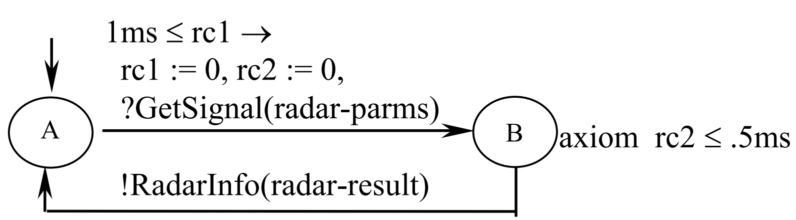


Radar Component

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

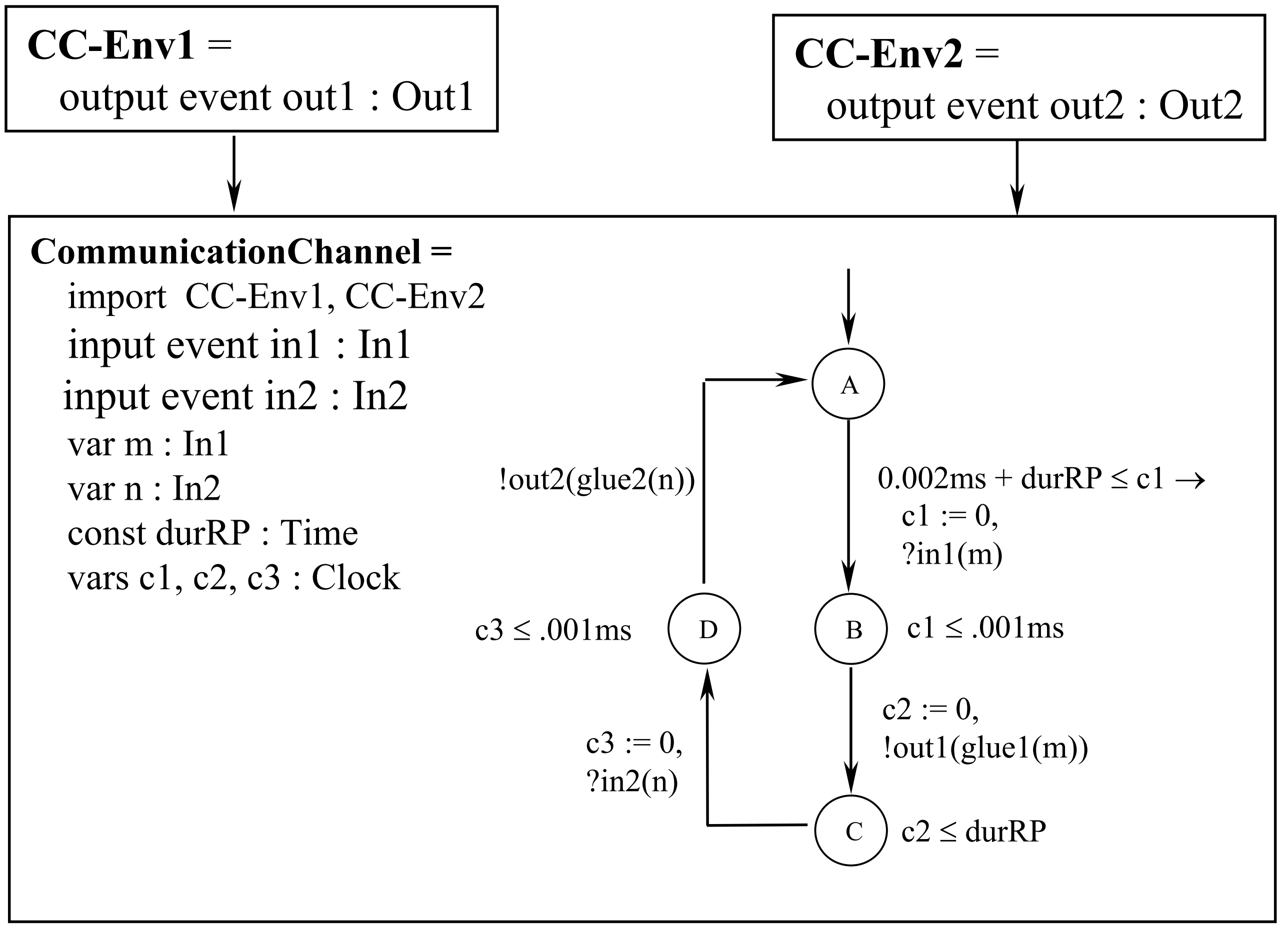
Radar-Env =
event RadarInfo : Radar-Response

Radar =
import Radar-Env
event GetSignal : Radar-Parameters
var radar-parms : Radar-Parameters
var radar-result : Radar-Response
vars rc1, rc2 : Clock = 0

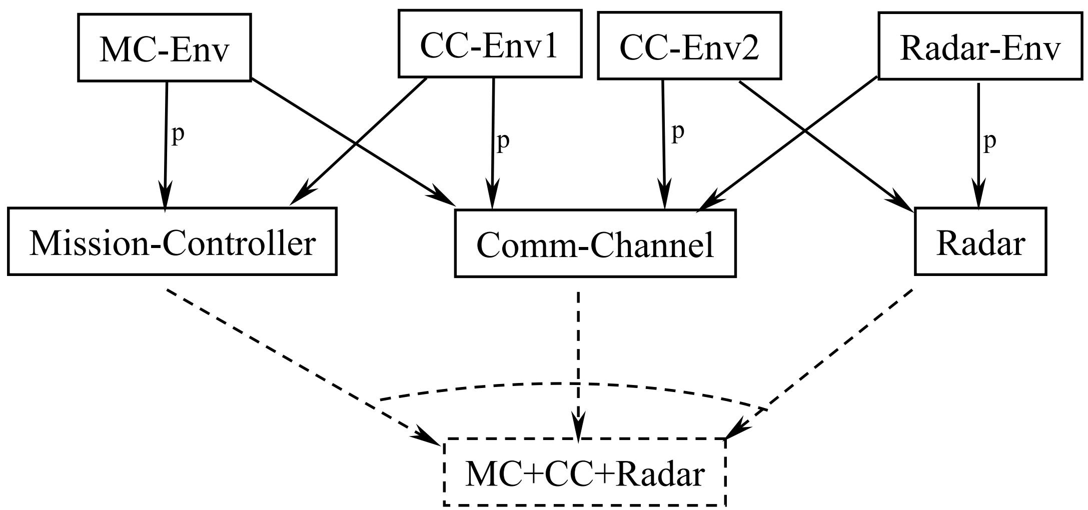


Communication Channel/Connector

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

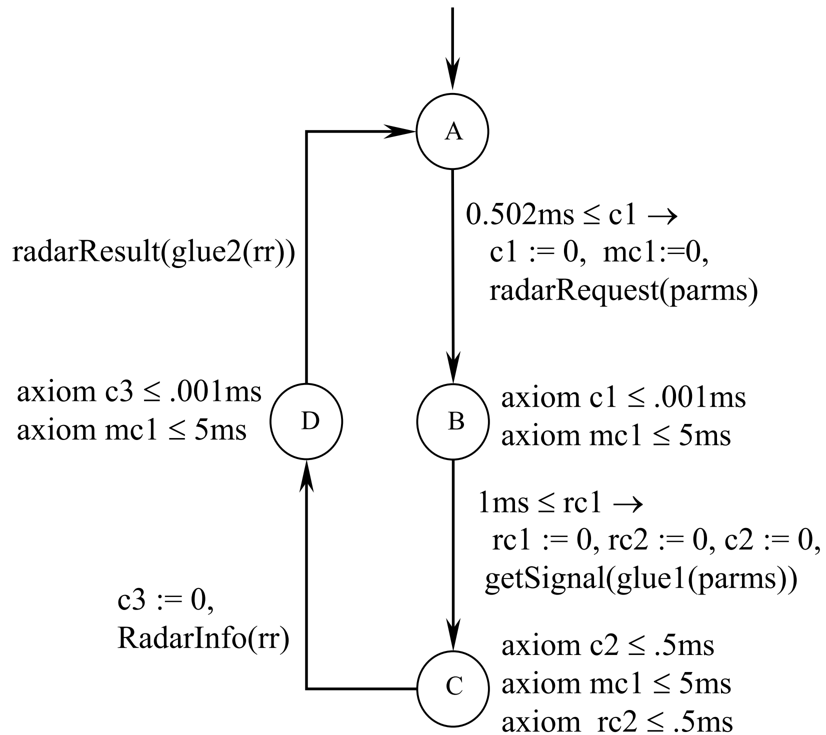


System Composition Diagram



Simplified Colimit

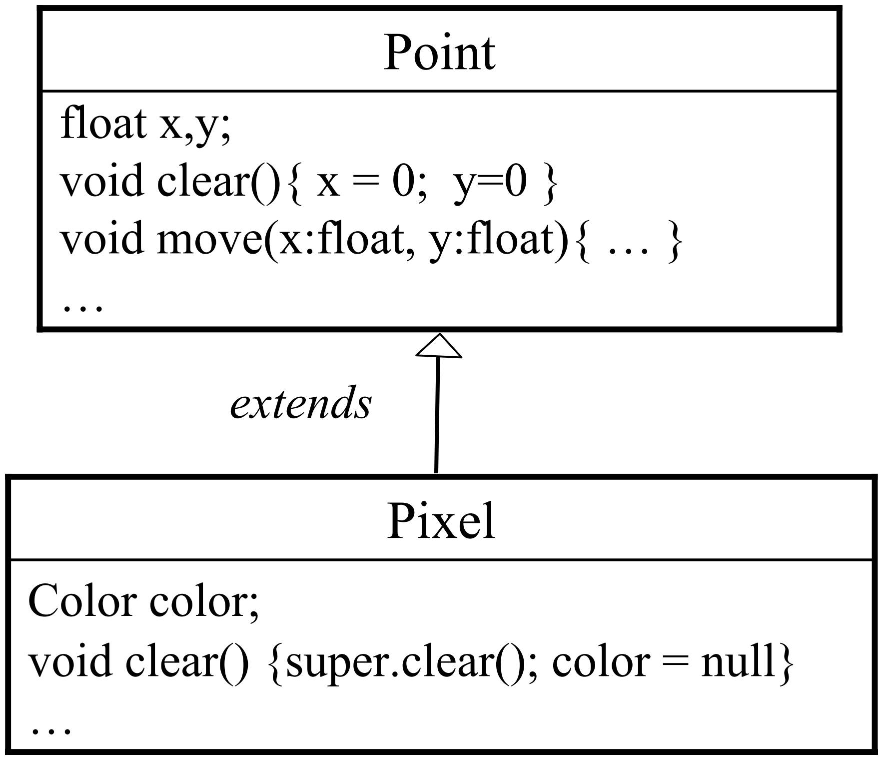
Mission-Control-System =
event RadarRequest : MC-Radar-Parameters
event RadarResult : MC-Radar-Response
event GetSignal : Radar-Parameters
event RadarInfo : Radar-Response
var parms : MC-Radar-Parameters
var rr : Radar-Response
op glue1 : MC-Radar-Parameters → Radar-Parameters
op glue2 : Radar-Response → MC-Radar-Response
vars mc1, c1, c2, c3, rc1, rc2 : Clock



Functional versus Behavioral Specifications

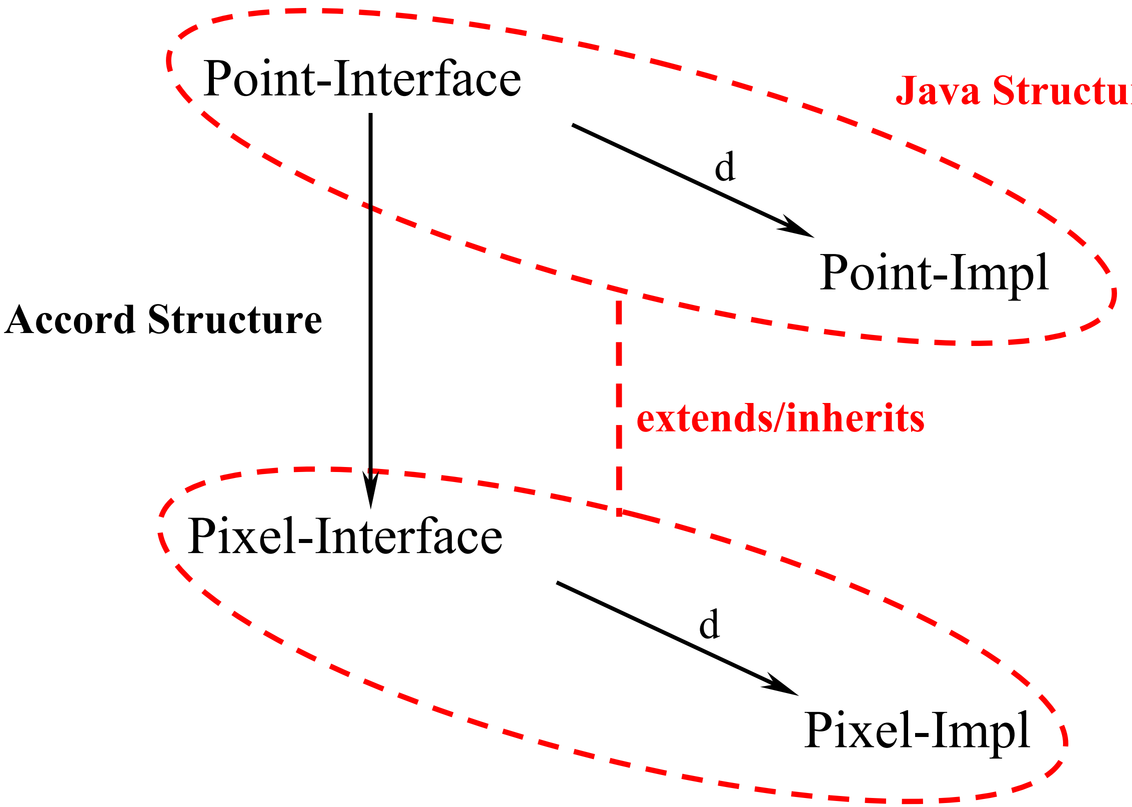
	functional specifications	behavioral specifications	
product, sum, function, subtype, quotient	types	classes	product, sum, function, extension
	functions	procedures	
	axioms	axioms	
	Specification units	Module units	import, parameterize, refine, compose by colimit

Example: Points and Pixels



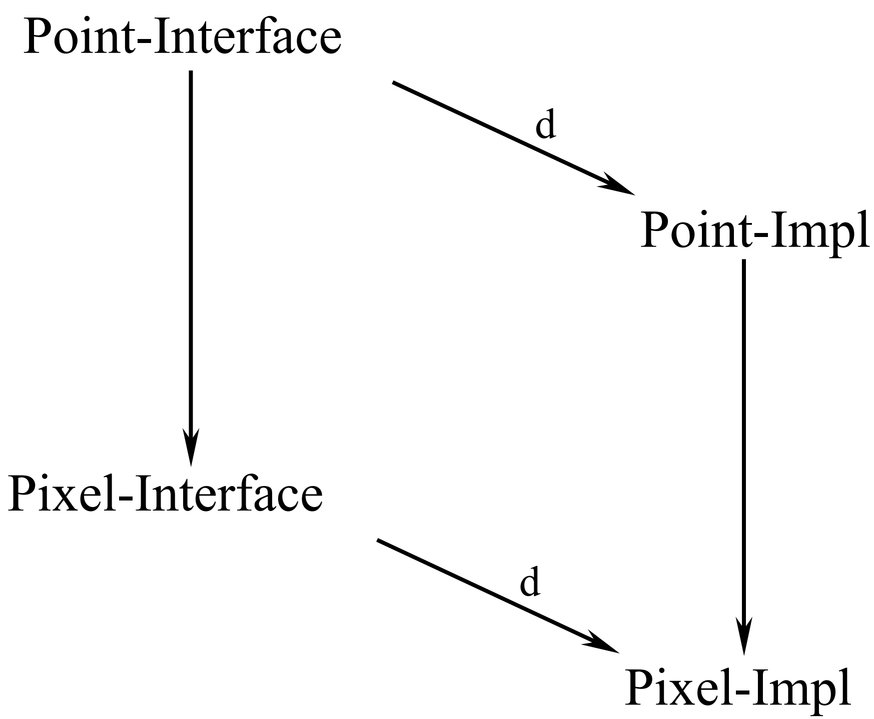
Classes, Inheritance, Implementations

Overriding is not semantically acceptable in Specware



Class Refinement

Overriding is not semantically acceptable in Specware



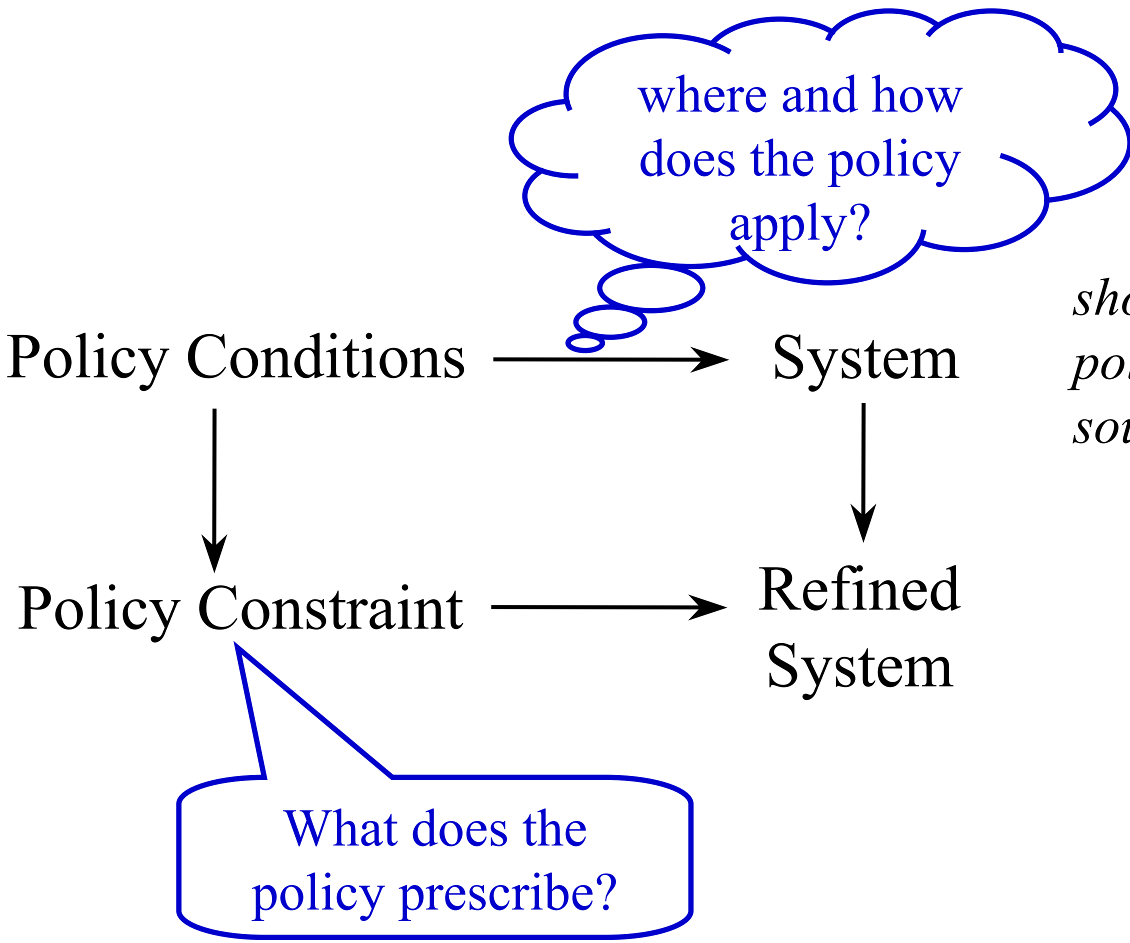
Issue: How to Handle Nonfunctional and Cross-Cutting Concerns wrt Composition and Refinement?

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

Examples

- Log all errors that arise during system execution
- Enforce a system-wide error-handling policy
- Disallow unauthorized data accesses
- Enforce timing and resource constraints on a system design

Policy Enforcement Approach



showing where the policy applies requires sound static analysis

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.”

R. Blakely and C. Heath, Security Design Patterns, The Open Group, 2004 (http://www.opengroup.org/security/gsp.htm).

Design Pattern for Protected Systems

aka Reference Monitor

Client, Resource,
policy axiom

→

Local Network
with Users,
Personnel Database

↓

Client, Guard,
Policy, Resource

→

Local Network
with Access Control
on Personnel Database