System-Level Design Languages: Orthogonalizing the Issues

The GSRC Semantics Project

Tom Henzinger

Luciano Lavagno

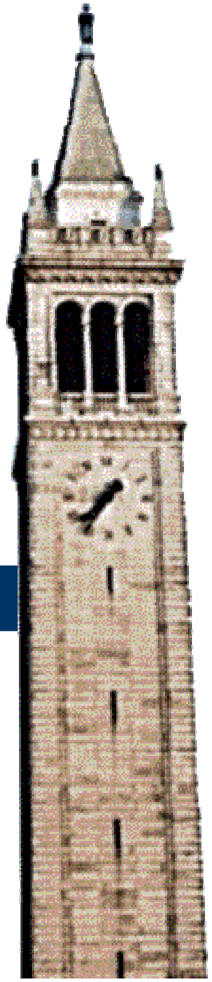
Edward Lee

Alberto Sangiovanni-Vincentelli

Kees Vissers

Edward A. Lee

UC Berkeley



What is GSRC?

The MARCO/DARPA Gigascale Silicon Research Center

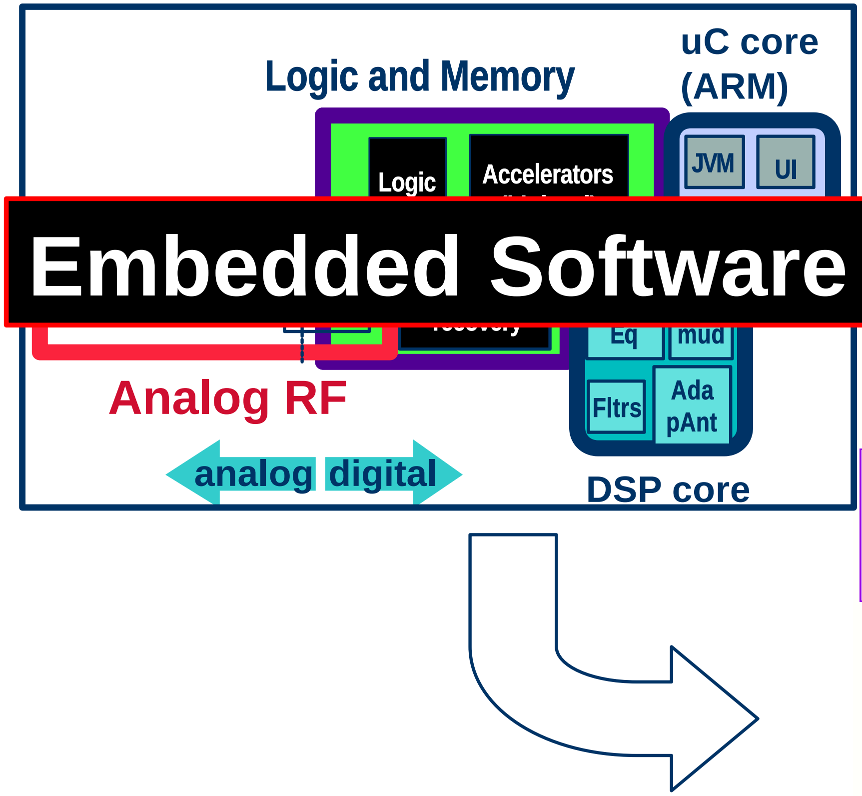
- keep the fabs full
- close the productivity gap
- rebuild the RTL foundation
- enable scaleable, heterogeneous, component-based design

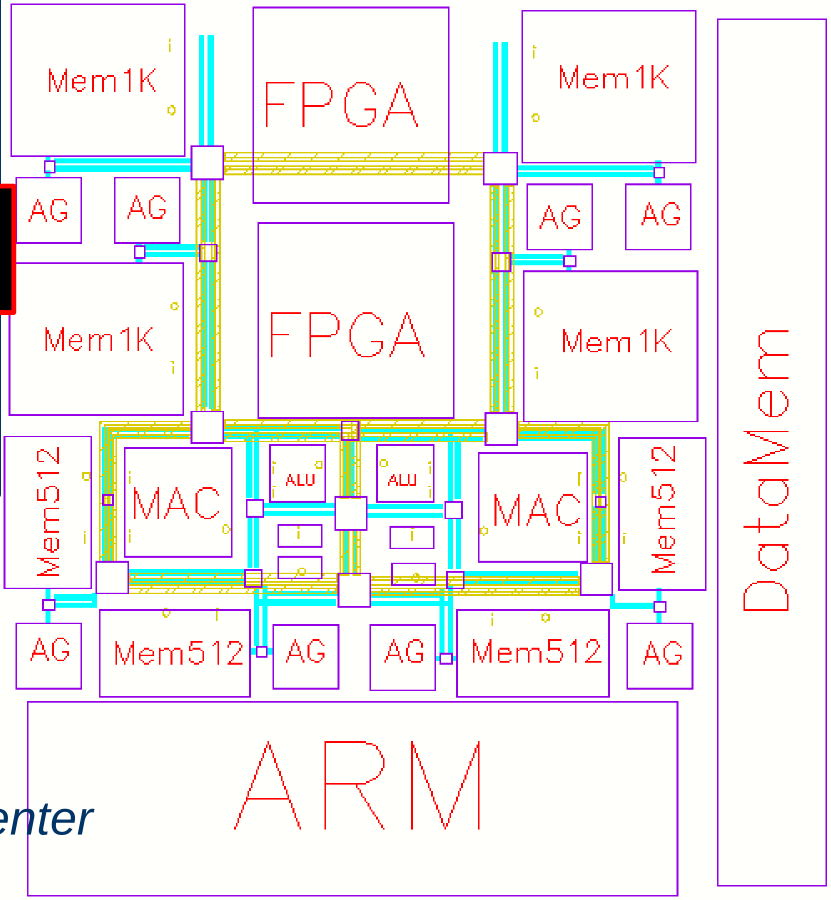
http://www.gigascale.org

Participants:

- UC Berkeley
- CMU
- Stanford
- Princeton
- UCLA
- UC Santa Barbara
- UC San Diego
- Purdue
- Michigan
- UC Santa Cruz

What is System Level?





Source: Berkeley Wireless Research Center

The Future of System-Level Architecture?

Poor common infrastructure.
Weak specialization.
Poor resource management.
Poor planning.

Elegant Federation

Moving away from obsessive uniformity towards elegant federation of heterogeneous models.

Two Rodeo Drive, Kaplan, McLaughlin, Diaz

Source: *Kaplan McLaughlin Diaz*, R. Rappaport, Rockport, 1998

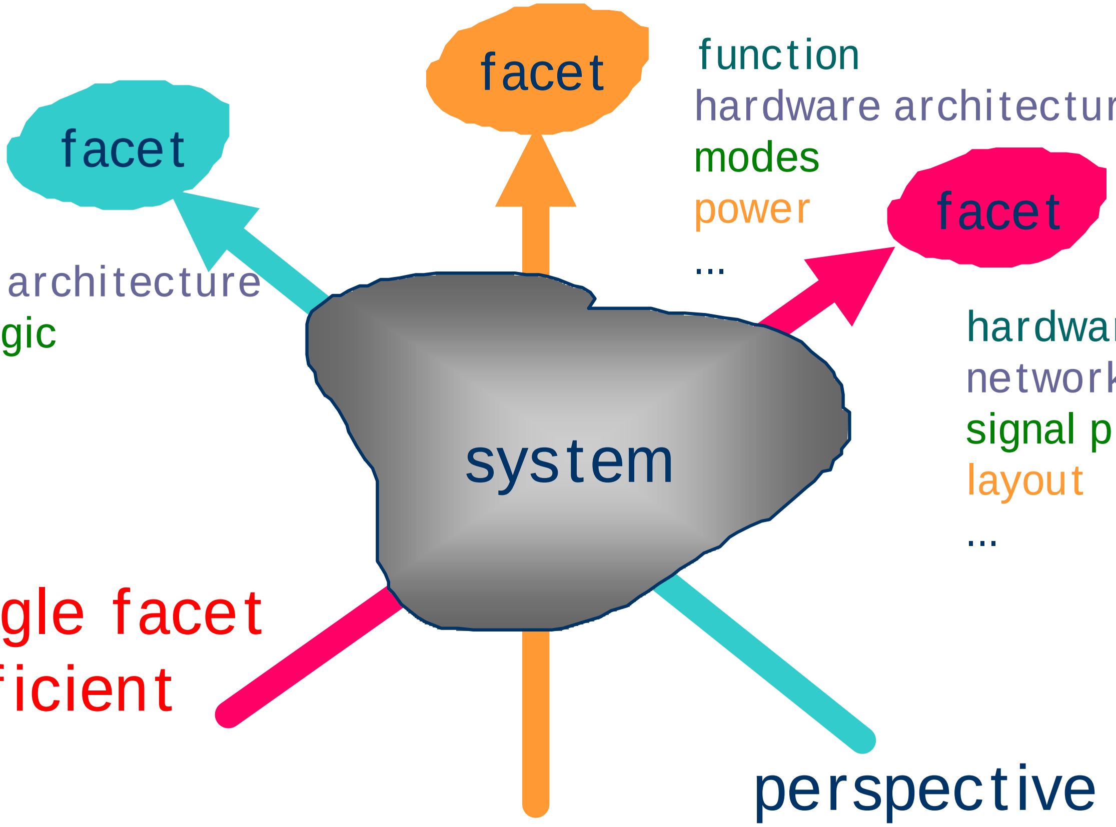
Focus on Capabilities, not Languages

- Modeling
- Simulation
- Visualization
- Synthesis
- Verification
- Modularization

The problem we are here to address is *interoperability* and *design productivity*. Not standardization.

Perspectives

Designers, users, maintainers interact with facets



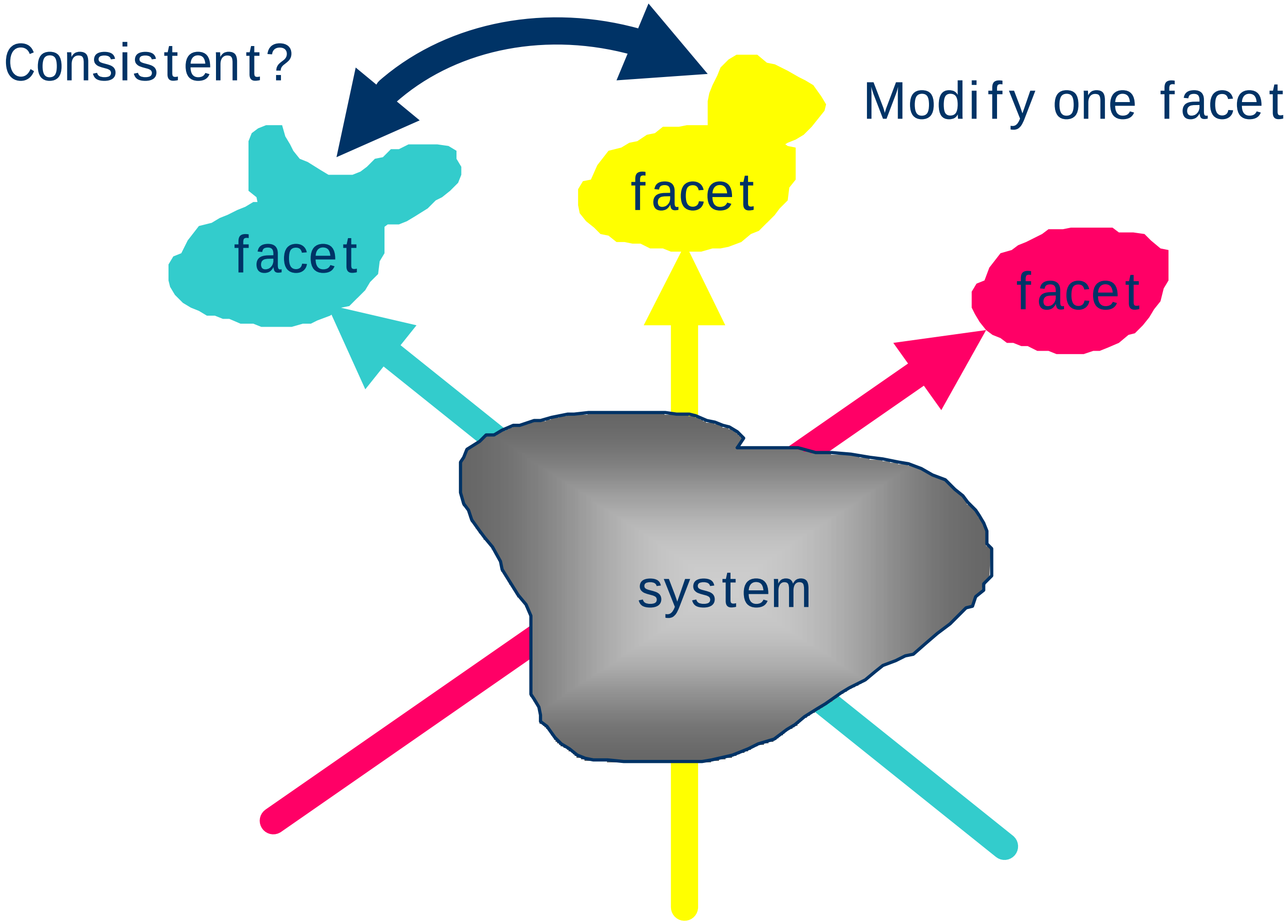
timing
software architecture
control logic
logic
...

function
hardware architecture
modes
power
...

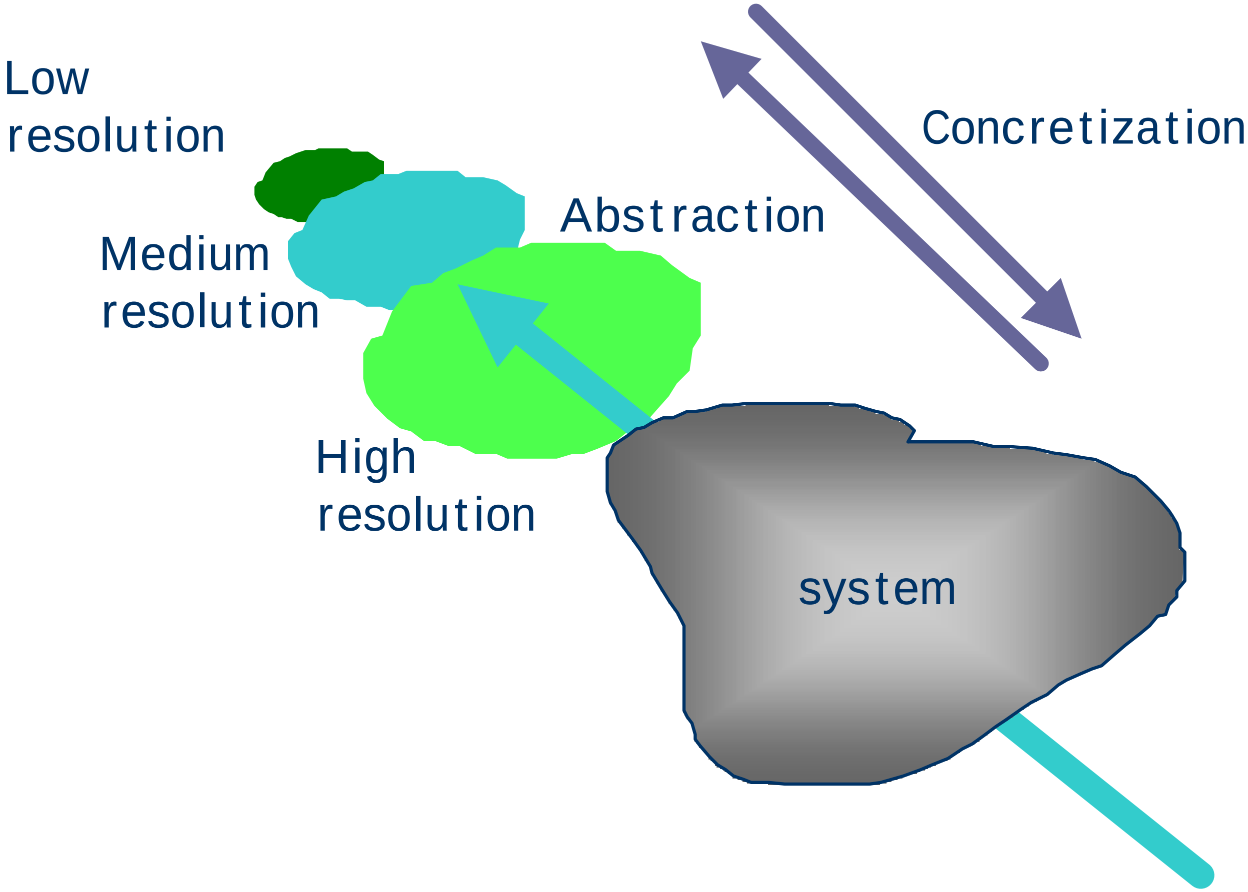
hardware architecture
networking
signal processing
layout
...

No single facet is sufficient

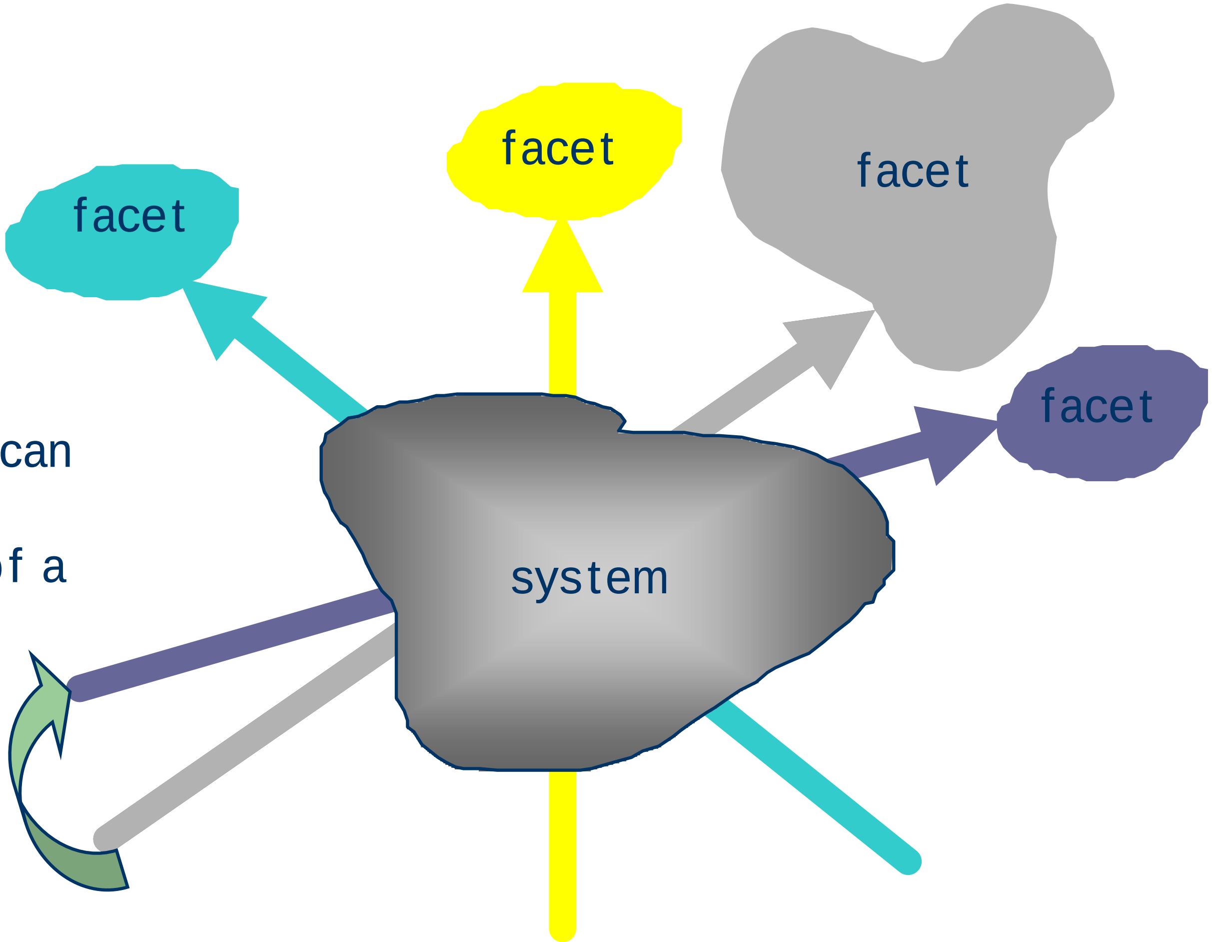
Interactions



Abstraction



Choosing Perspectives



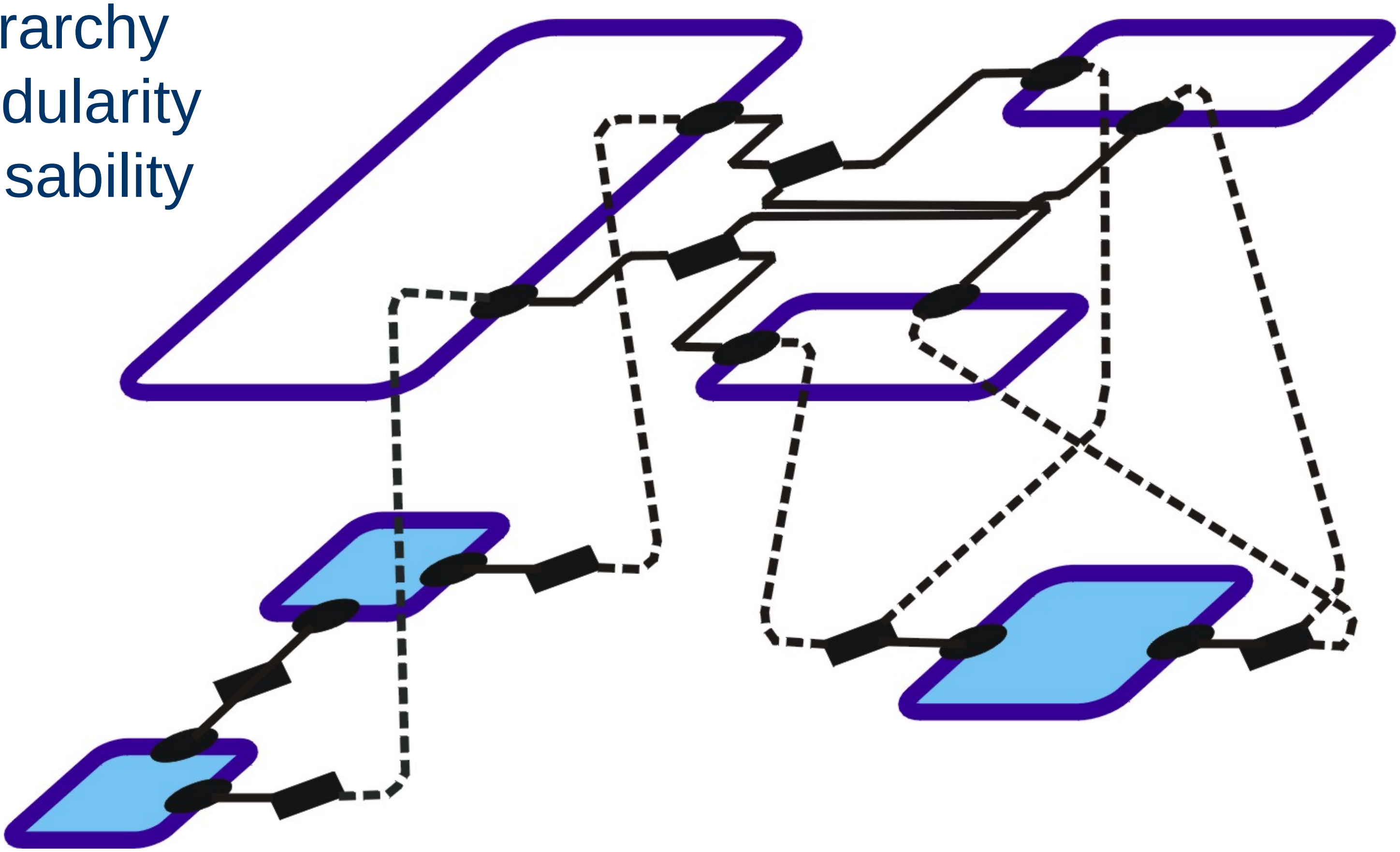
Shift in perspective can reduce the complexity of a facet

Interoperability Levels

- Code can be written to translate the data from one tool to be used by another.
- Tools can open each other's files and extract useful information (not necessarily *all* useful information).
- Tools can interoperate dynamically, exchanging information at run time.

Component-Based Design

hierarchy
modularity
reusability



Must Be Able to Specify

- Netlists
- Block diagrams
- Hierarchical state machines
- Object models
- Dataflow graphs
- Process networks

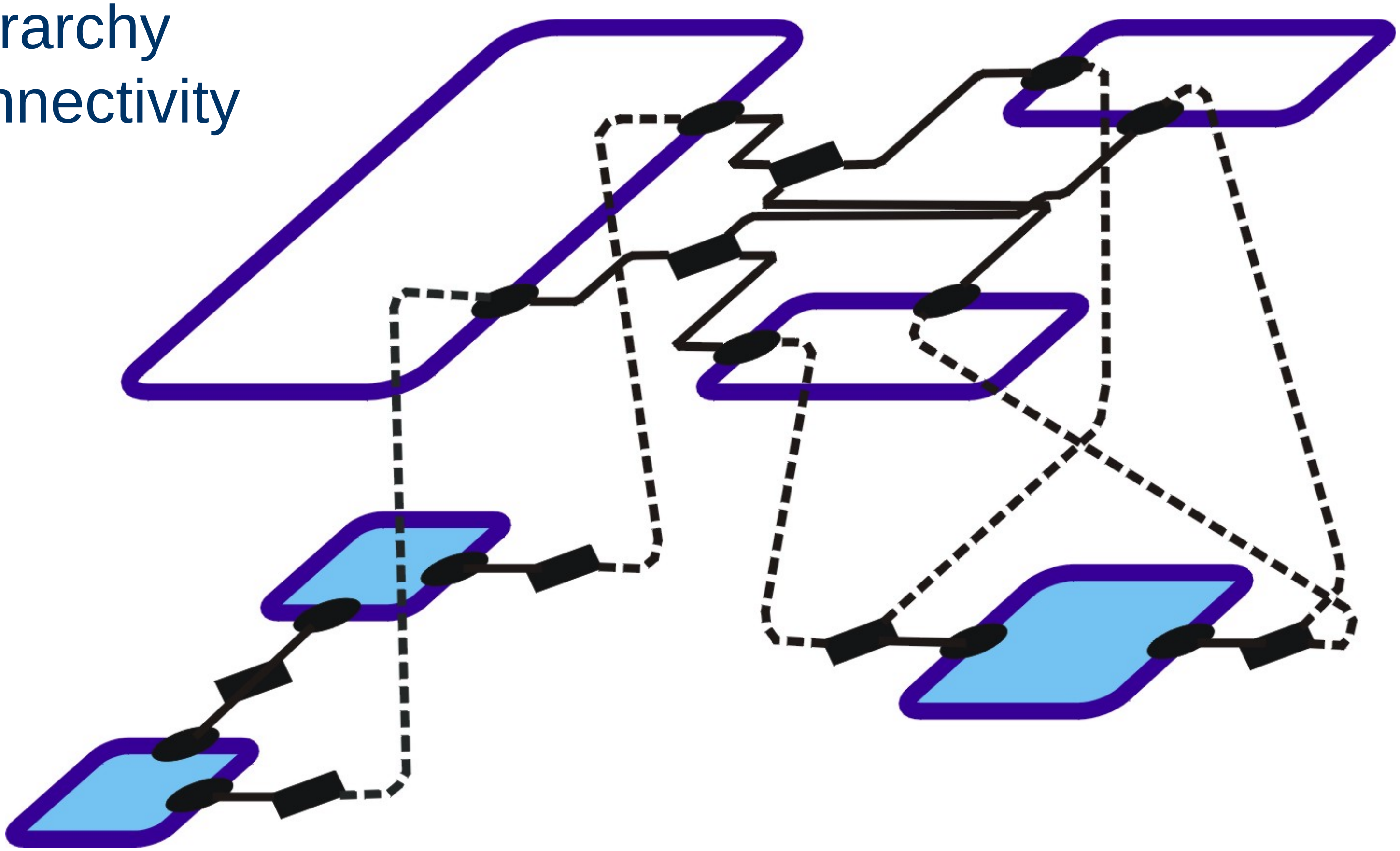
Principle: Orthogonalize Concerns in SLDLs

- Abstract Syntax
- Concrete Syntax
- Syntactic Transformations
- Type System
- Component Semantics
- Interaction Semantics

Do this first, since without it, we won’t get anywhere

Abstract Syntax

hierarchy
connectivity



Not Abstract Syntax

- Semantics of component interactions
- Type system
- File format (a concrete syntax)
- API (another concrete syntax)

An abstract syntax is the logical structure of a design. What are the pieces, and how are they related?

Definitions

A **frame** f

$Ports_f$, a set;

$Relations_f$, a set;

$Links_f \subseteq Ports_f \times Relations_f$.

A **model** m

$Frame_m$, a frame;

$Hierarchy_m$, a hierarchy on $Frame_m$.

Hierarchy

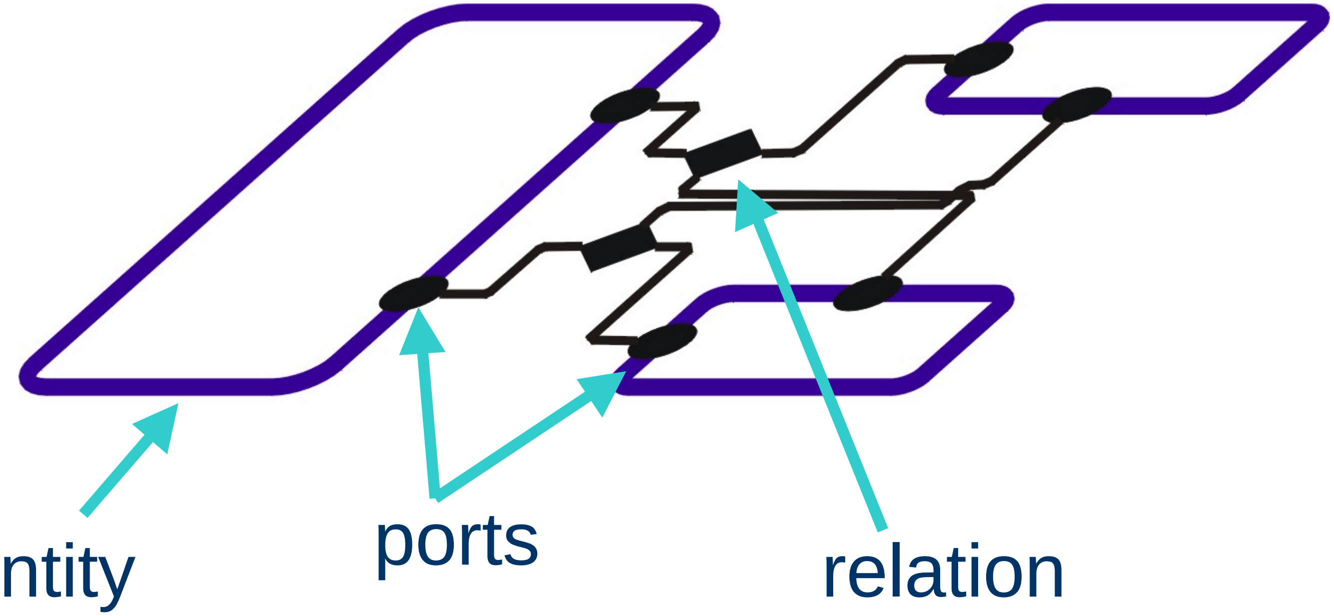
A **hierarchy** h on the frame f

$Entities_h$, a set;
$ContainedPorts_h : Entities_h \rightarrow \mathcal{P}(Ports_f)$;
$ContainedRelations_h : Entities_h \rightarrow \mathcal{P}(Relations_f)$;
$ContainedEntities_h : Entities_h \rightarrow \mathcal{P}(Entities_f)$;

Constraints:

- A port cannot be contained by more than entity.
- A link cannot cross levels of the hierarchy

Connected Components

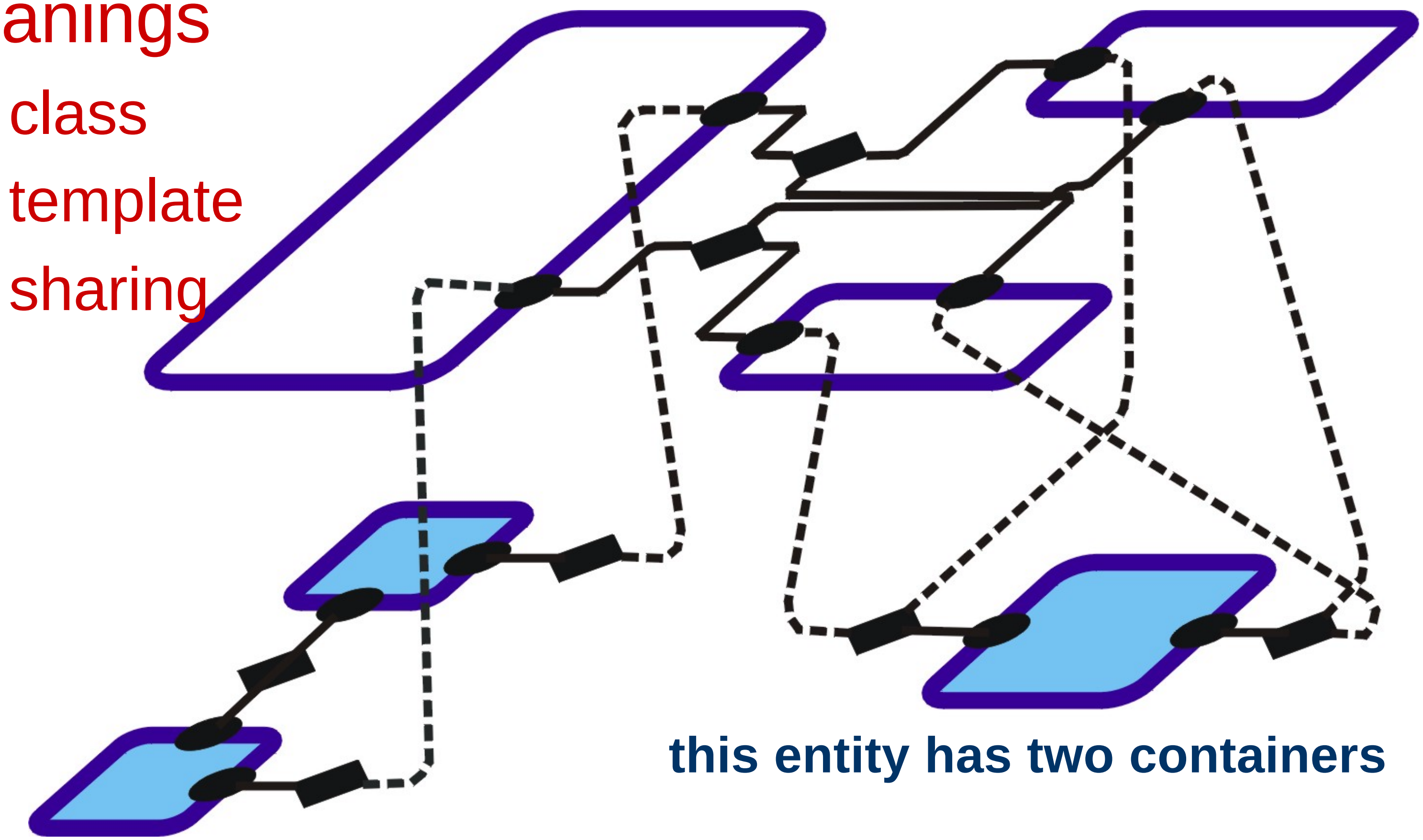


- Frame in black
- Hierarchy in blue

Hierarchy and Sharing

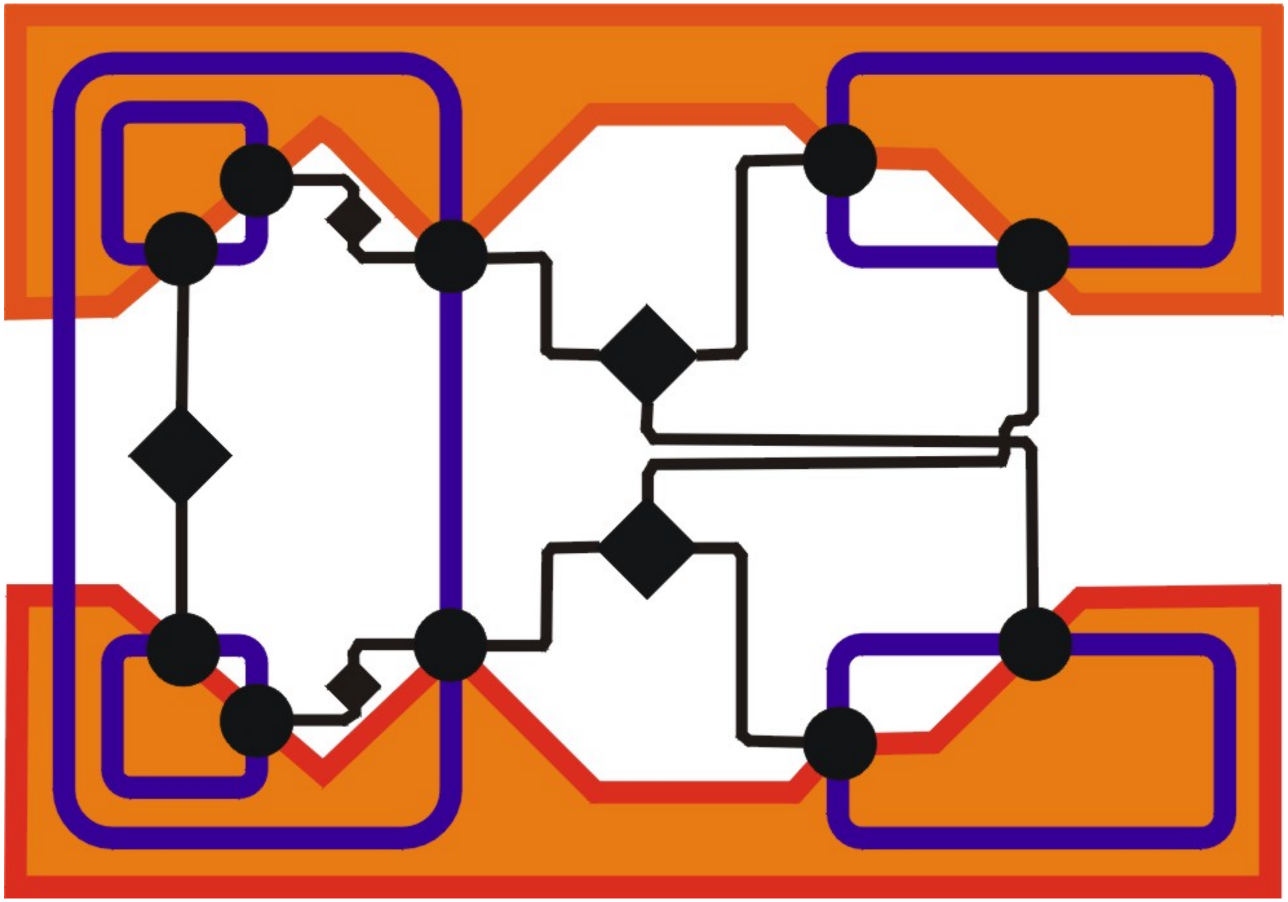
Meanings

- class
- template
- sharing



this entity has two containers

Heterarchy



One hierarchy in blue, another in orange.

The GSRC Abstract Syntax

- Models hierarchical connected components
  - block diagrams, object models, state machines, …
  - abstraction and refinement
- Supports classes and instances
  - object models
  - inheritance
  - static and instance variables
- Supports multiple simultaneous hierarchies
  - structure and function
  - objects and concurrency

Concrete Syntaxes

- Persistent file formats
- Close to the abstract syntax
- Make it extensible to capture other aspects
- Enable design data exchange
  - without customization of the tools

Most language discussions focus on concrete syntaxes, which are arguably the least important part of the design

MoML – An XML Concrete Syntax

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model PUBLIC "…" "http://…">
<model name="top" class="path name">
  <entity name="source" class="path name">
    <port name="output"/>
  </entity>
  <entity name="sink" class="path name">
    <port name="input"/>
  </entity>
  <relation name="r1" class="path name"/>
  <link port="source.output" relation="r1"/>
  <link port="sink.input" relation="r1"/>
</model>
```

MoML DTD

Modeling Markup Language

```
<!ELEMENT link EMPTY>
<!ATTLIST link port CDATA #REQUIRED
            relation CDATA #REQUIRED
            vertex CDATA #IMPLIED>
```

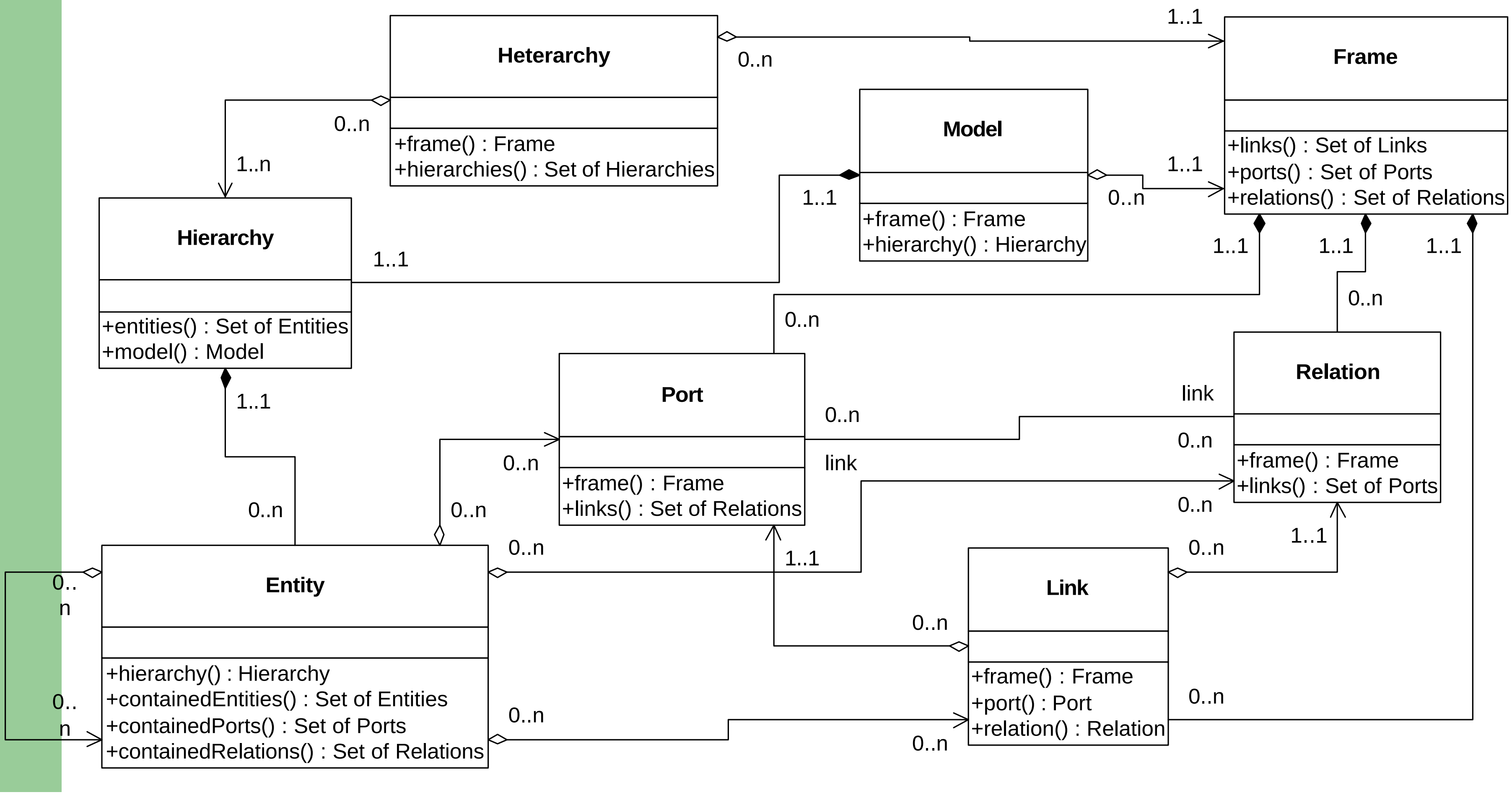
Since this document type definition captures only the abstract syntax, it is very small and simple. Other information is embedded using distinct XML DTDs.

```
<!ELEMENT model (attribute | class | configure | doc | director| entity | import | link | relation)*>
<!ATTLIST model name CDATA #REQUIRED
        class CDATA #REQUIRED>

<!ELEMENT attribute (doc | configure)*>
<!ATTLIST attribute class CDATA #IMPLIED
          name CDATA #REQUIRED
          value CDATA #IMPLIED>

<!ELEMENT class (attribute | configure | director | doc | entity| link)*>
<!ATTLIST class name CDATA #REQUIRED
        extends CDATA #REQUIRED>

<!ELEMENT configure (#PCDATA)>
<!ATTLIST configure source CDATA #IMPLIED>

<!ELEMENT director (attribute | configure)*>
<!ATTLIST director name CDATA "director"
          class CDATA #REQUIRED>

<!ELEMENT doc (#PCDATA)>

<!ELEMENT entity (attribute | class | configure | doc | director  | entity | rendition | relation)*>
<!ATTLIST entity name CDATA #REQUIRED
         class CDATA #REQUIRED>

<!ELEMENT import EMPTY>
<!ATTLIST import source CDATA #REQUIRED>

<!ELEMENT link EMPTY>
<!ATTLIST link port CDATA #REQUIRED
        relation CDATA #REQUIRED
        vertex CDATA #IMPLIED>

<!ELEMENT location EMPTY>
<!ATTLIST location x CDATA #REQUIRED
           y CDATA #IMPLIED
           z CDATA #IMPLIED>

<!ELEMENT port (doc | configure)*>
<!ATTLIST port name CDATA #REQUIRED
        class CDATA #REQUIRED
        direction (input | output | both) "both">

<!ELEMENT relation (vertex*)>
<!ATTLIST relation name CDATA #REQUIRED
          class CDATA #REQUIRED>

<!ELEMENT rendition (configure | location)*>
<!ATTLIST rendition class CDATA #REQUIRED>

<!ELEMENT vertex (location?)>
<!ATTLIST vertex name CDATA #REQUIRED
         pathToCDATA #IMPLIED>
```

Syntactic Transformations

- A set of operations on models
  - creation of ports, relations, links, and entities
  - mutation
- Applications
  - visual editors
  - higher-order functions
  - instantiation
  - unrolling recursion

API: Concrete Syntax Supporting Syntactic Transformations



Heterarchy

+frame() : Frame
+hierarchies() : Set of Hierarchies

Frame

+links() : Set of Links
+ports() : Set of Ports
+relations() : Set of Relations

Model

+frame() : Frame
+hierarchy() : Hierarchy

Hierarchy

+entities() : Set of Entities
+model() : Model

Port

+frame() : Frame
+links() : Set of Relations

Relation

+frame() : Frame
+links() : Set of Ports

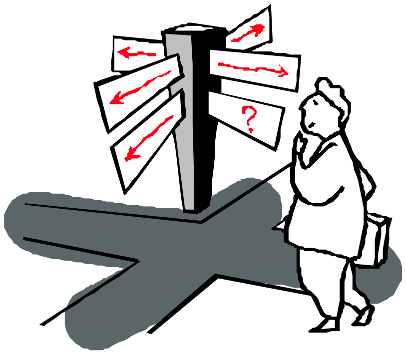
Entity

+hierarchy() : Hierarchy
+containedEntities() : Set of Entities
+containedPorts() : Set of Ports
+containedRelations() : Set of Relations

Link

+frame() : Frame
+port() : Port
+relation() : Relation

Where We Are…



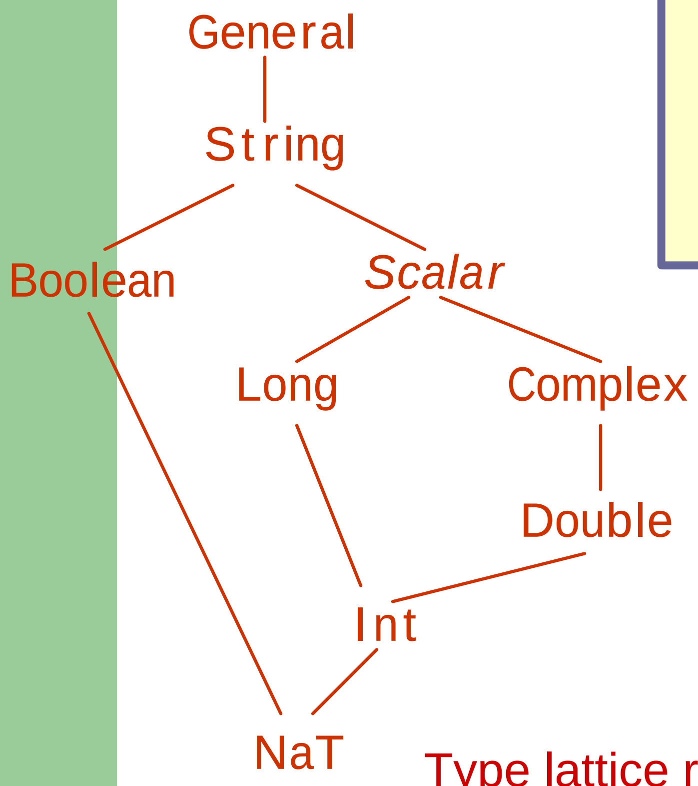
- Abstract Syntax ✓
- Concrete Syntax ✓
- Syntactic Transformations ✓

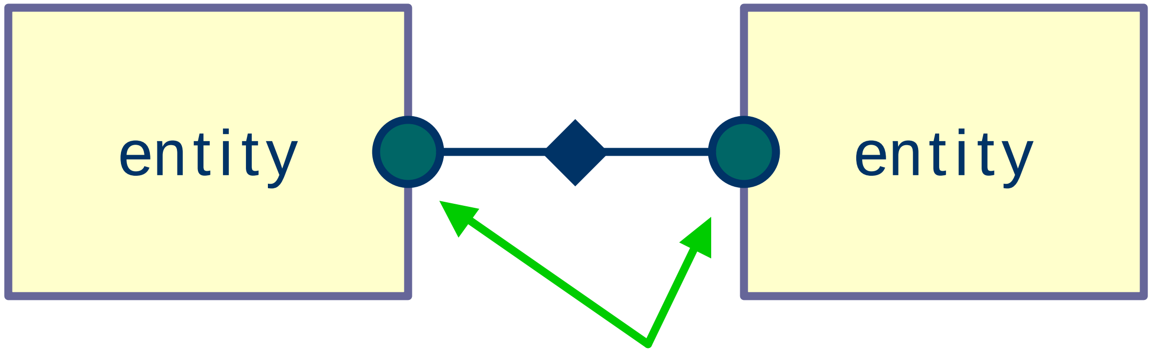
logical structure

- Type System
- Component Semantics
- Interaction Semantics

meaning

Type Systems





need compatible data types

Type lattice represents subclassing & ad-hoc convertibility.

Desirable Properties in a Type System

- Strong typing
- Polymorphism
- Propagation of type constraints
- Composite types (arrays, records)
- User-defined types
- Reflection
- Higher-order types
- Type inference
- Dependent types

We can have compatible type systems without compatible languages (witness CORBA)

Component Semantics

Entities are:

- States?
- Processes?
- Threads?
- Differential equations?
- Constraints?
- Objects?

Are Software Component Models Enough?

Largely missing:

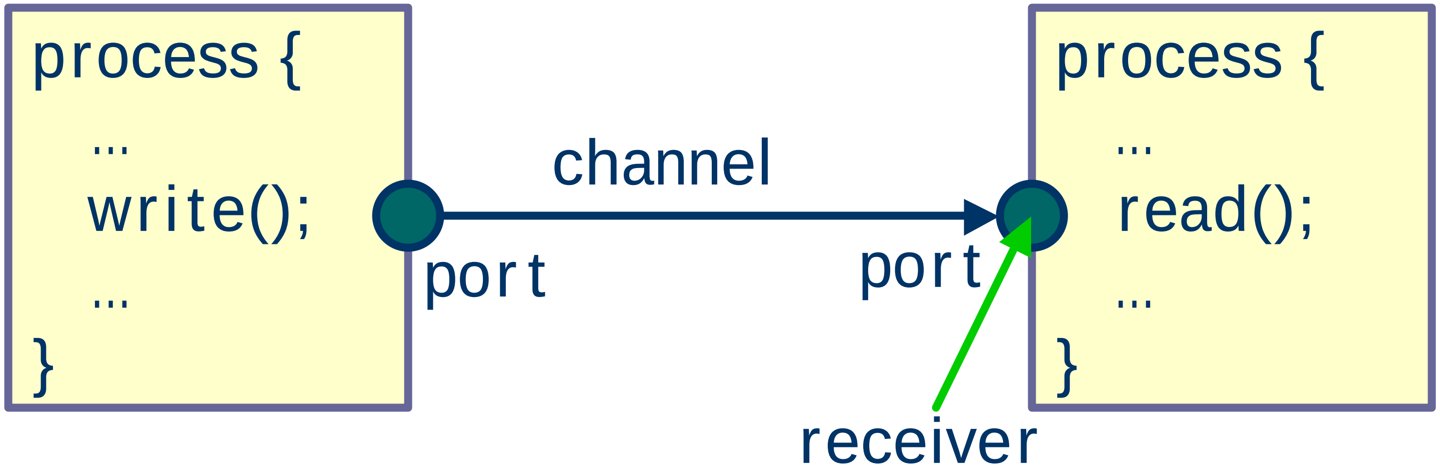
- Time
- Concurrency
- Safety
- Liveness

Are Hardware Component Models Enough?

Largely missing:

- Abstraction (esp time)
- Inheritance
- Type systems
- Polymorphism
- Portability

One Class of Semantic Models: Producer / Consumer



process {
…
write();
…
}

channel

port

port

receiver

process {
…
read();
…
}

- Are actors active? passive? reactive?
- Are communications timed? synchronized? buffered?

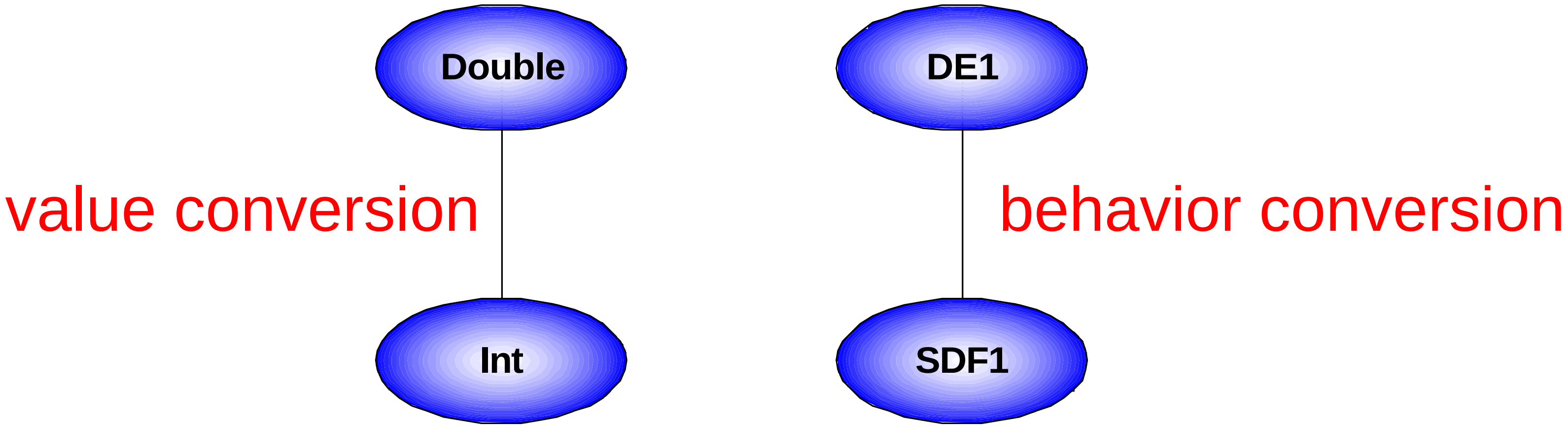
Domains

- CSP – concurrent threads with rendezvous
- CT – continuous-time modeling
- DE – discrete-event systems
- DT – discrete time (cycle driven)
- PN – process networks
- SDF – synchronous dataflow
- SR – synchronous/reactive

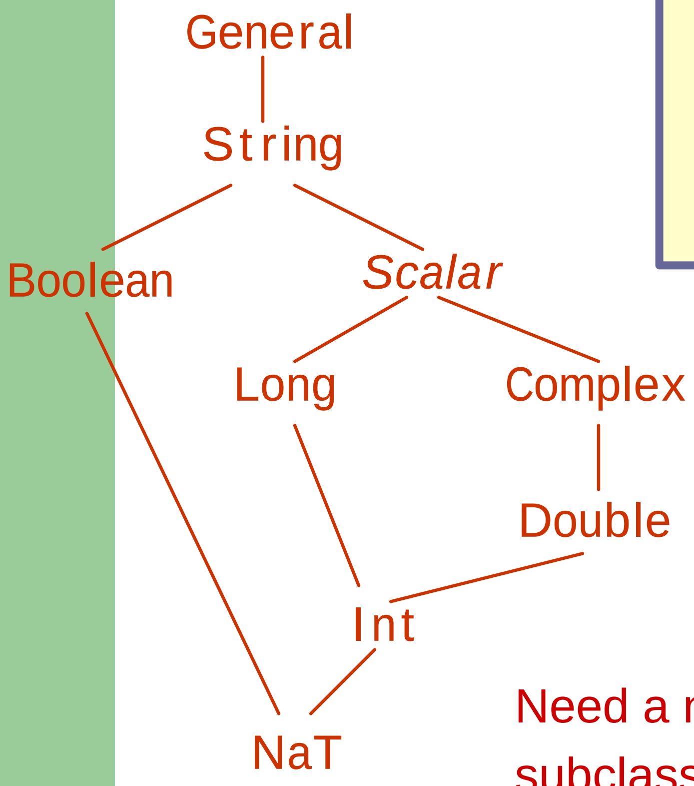
Each of these defines a component ontology and an interaction semantics between components. There are many more possibilities!

Interfaces

- Represent not just data types, but interaction types as well.



Current Approach – System-Level Types



General

String

Boolean

Scalar

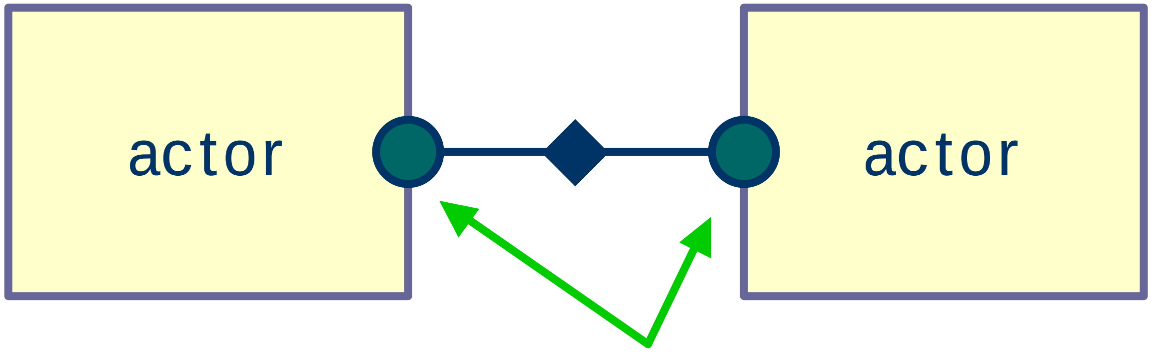
Long

Complex

Double

Int

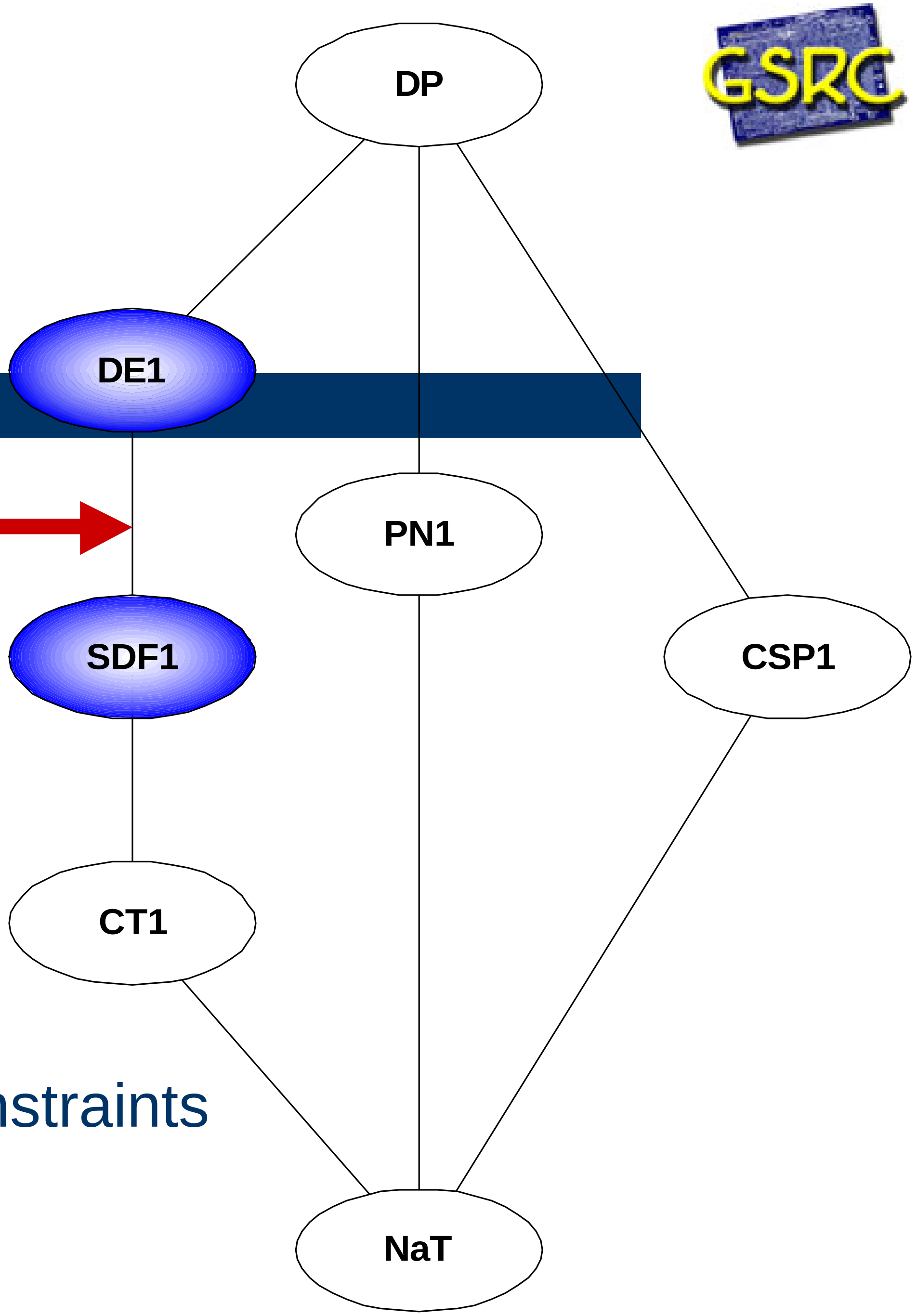
NaT



represent interaction semantics as types on these ports.

Need a new type lattice representing subclassing & ad-hoc convertibility.

Type Lattice



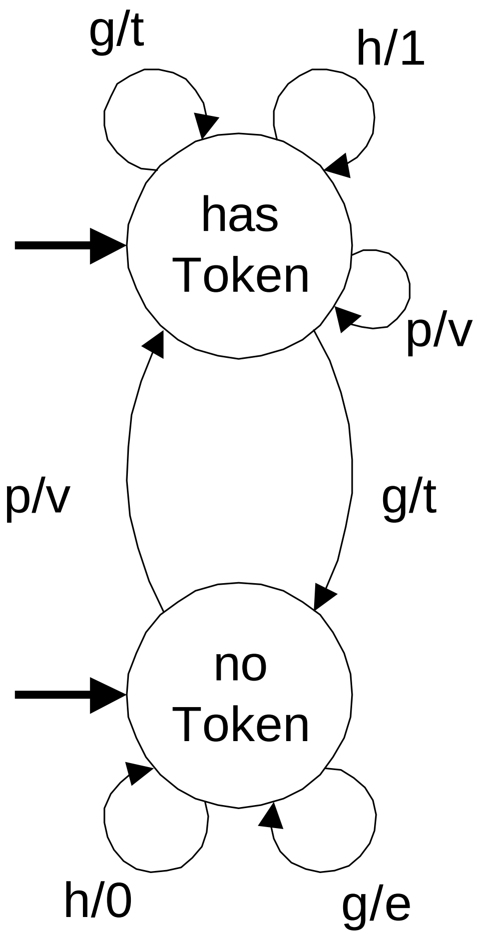
Simulation relation

Achievable properties:

- Strong typing
- Polymorphism
- Propagation of type constraints
- User-defined types
- Reflection

SDF Receiver Type Signature

SDF1



Input alphabet:

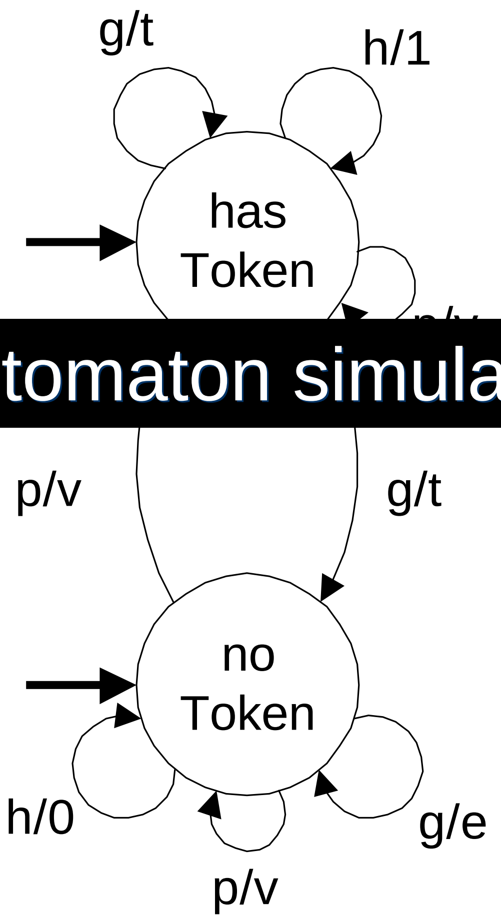
g: get
p: put
h: hasToken

Output alphabet:

0: false
1: true
t: token
v: void
e: exception

DE Receiver Type Signature

DE1



This automaton simulates the previous one

Put does not necessarily result in immediate availability of the data.

Input alphabet:

g: get
p: put
h: hasToken

Output alphabet:

0: false
1: true
t: token
v: void
e: exception

System-Level Types

- Declare dynamic properties of component interfaces
- Declare timing properties of component interfaces

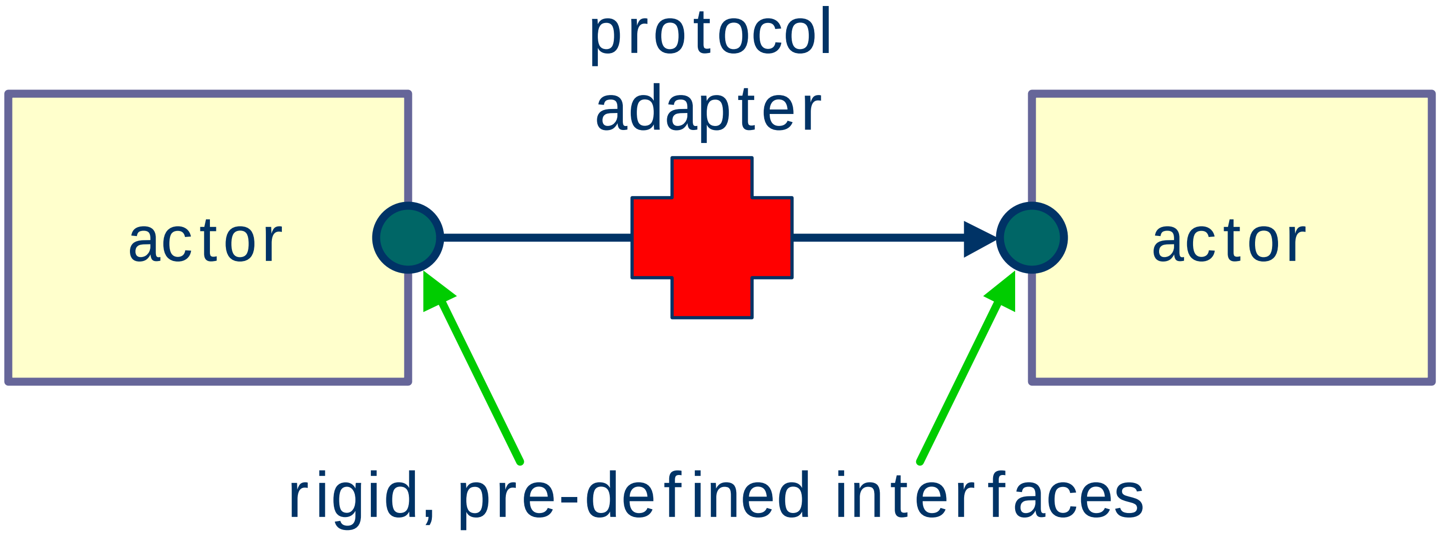
Benefits:

- Ensure component compatibility
- Clarify interfaces
- Provide the vocabulary for design patterns
- Detect errors sooner
- Promote modularity
- Promote polymorphic component design

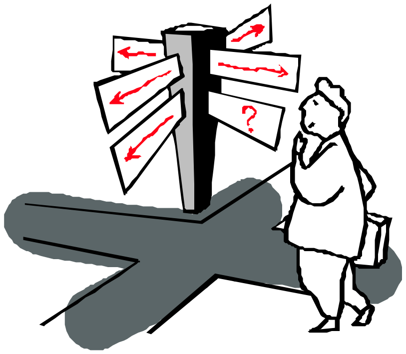
Our Hope – Polymorphic Interfaces

polymorphic interfaces

Alternative Approach – Interface Synthesis



Where We Are…



- Abstract Syntax ✓
- Concrete Syntax ✓
- Syntactic Transformations ✓
- Type System ✓
- Component Semantics ✓
- Interaction Semantics ✓

Benefits of Orthogonalization

- Modularity in language design
  - e.g. can build on existing abstract syntax
- Different levels of tool interoperability
  - e.g. visualization tool needs only the abstract syntax
- Terminology independent of concrete syntax
  - e.g. design patterns
- Focus on frameworks instead of languages
  - dealing with heterogeneity
- Issue-oriented not ASCII-oriented

Ptolemy Project – Sanity Check

Ptolemy II –

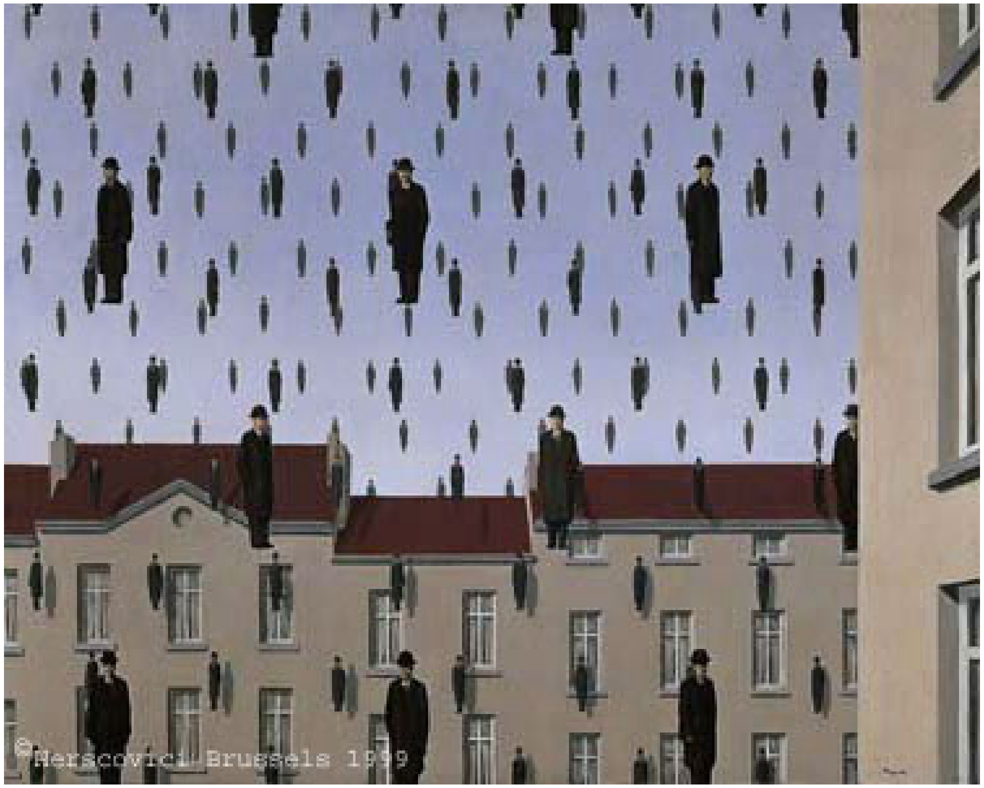
- A reference implementation
- Testbed for abstract syntax
- Block diagram MoML editor
- Mutable models
- Extensible type system
- Testbed for system-level types

http://ptolemy.eecs.berkeley.edu

Design in an Abstract Universe

When choosing syntax and semantics, we can invent the “laws of physics” that govern the interaction of components.

As with any such laws, their utility depends on our ability to understand models governed by the laws.



Magritte, Gelconde

http://www.gigascale.org/semantics