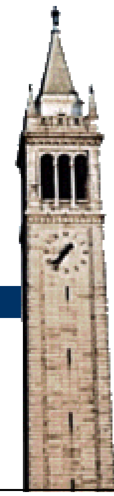
# The Gigascale Silicon Research Center

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

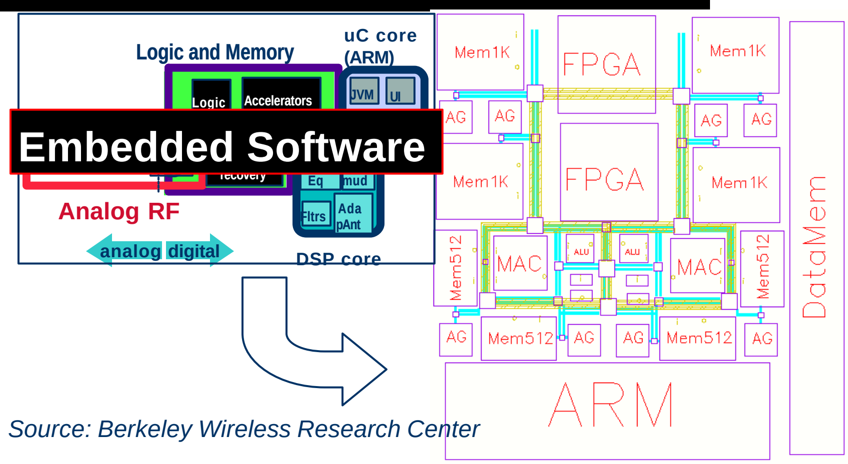
- keep the fabs full
- close the productivity gap
- rebuild the RTL foundation
- enable scaleable, heterogeneous, component-based design

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*



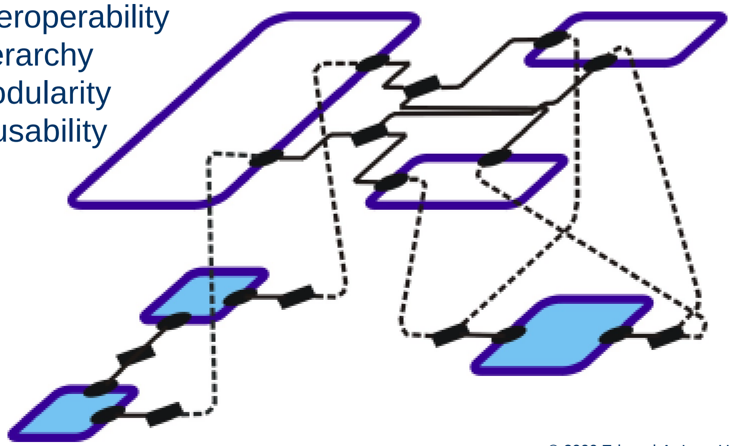
# Focus on Capabilities, not Languages

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

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

## Component-Based Design

interoperability
hierarchy
modularity
reusability





## 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.

## Principle: Orthogonalize Concerns in SLDLs

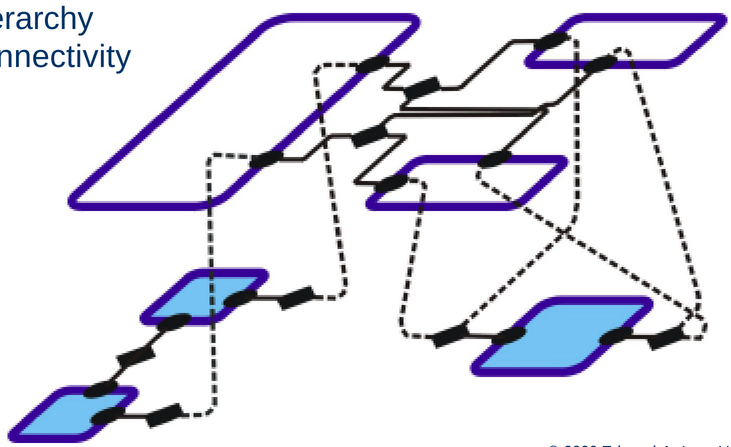
- Abstract Syntax
- Concrete Syntax
- Syntactic Transformations
- Type System
- Component Semantics
- Interaction Semantics

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?



## Must Be Able to Specify

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

## Interfaces and Ports

- A **partially ordered set** *Interfaces*
- A **set** *Ports*
- A **function** $ports: Interfaces \to \wp(Ports)$ s.t.
  - if $i < j$ then $ports(i) \subseteq ports(j)$



## Properties

- A **set** *Properties*
- A **function**
  $properties: Interfaces \to \wp(Properties)$ s.t.
  - if $i < j$ then $properties(i) \subseteq properties(j)$

## Inheritance Hierarchy

- *Interfaces*
- a partial ordering relation "<"
- *Ports*
- $ports: Interfaces \rightarrow \wp(Ports)$
- *Properties*
- $properties: Interfaces \rightarrow \wp(Properties)$

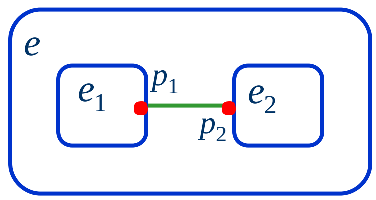


## Entities and Containment Hierarchy

- A **set** *Entities*
- A **member** $root \in Entities$
- A **function** $interface: Entities \rightarrow Interfaces$
- A **function** $containedEntities: Entities \rightarrow \wp(Entities)$

## Internal Links

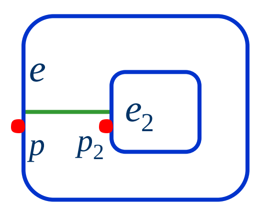
- A **function** *internalLinks*: $Entities \rightarrow \wp(Entities \times Ports \times Entities \times Ports)$
  - $(e_1, p_1, e_2, p_2) \in internalLinks(e) \Rightarrow$
    - $p_1 \in ports(interface(e_1))$
    - $p_2 \in ports(interface(e_2))$
    - $e_1 \in containedEntities(e)$
    - $e_2 \in containedEntities(e)$






## Interface Links

- A **function** *interfaceLinks*: $Entities \rightarrow \wp(Ports \times Entities \times Ports)$
  - $(p, e_2, p_2) \in interfaceLinks(e) \Rightarrow$
    - $p \in ports(interface(e))$
    - $p_2 \in ports(interface(e_2))$
    - $e_2 \in containedEntities(e)$

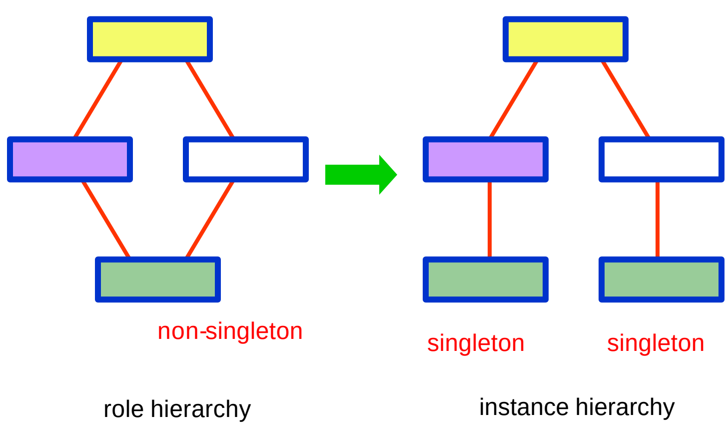


## Instance and Role Hierarchies

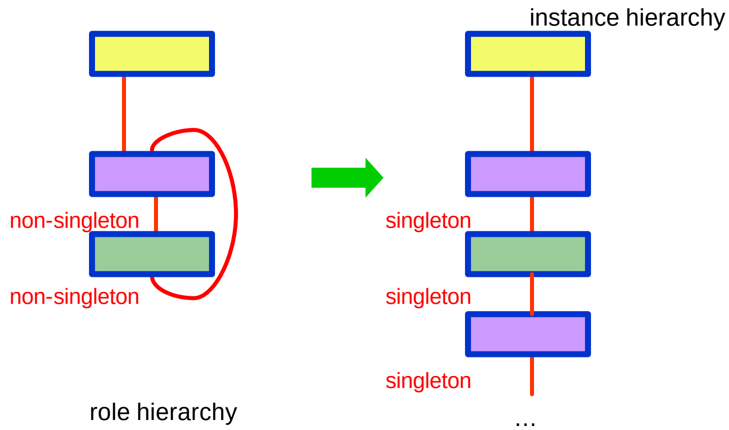
- A **function** *isSingleton*: *Entities* $\rightarrow$ *Boolean*
  - *isSingleton* (*root*) = *true*
- An **instance hierarchy** is a containment hierarchy where
  - $\forall\, e \in$ *Entities*, *isSingleton* (*e*) = *true*
- A **role hierarchy** is any other containment hierarchy
  - Every role hierarchy can be unrolled to a unique instance hierarchy.



## Unrolling



## Recursive Containment






## 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.

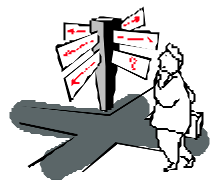
```
<!ELEMENT model (attribute | class | configure | doc | director| entity | import | link | relation)*>
<!ATTLIST model name CDATA #REQUIRED
          class CDATA #REQUIRED>

<!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
          pathTo CDATA #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

## Where We Are…



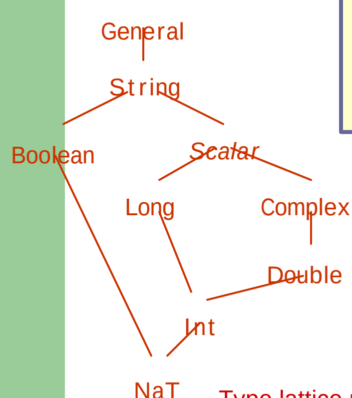
- Abstract Syntax ✔
- Concrete Syntax ✔
- Syntactic Transformations ✔
- Type System
- Component Semantics
- Interaction Semantics

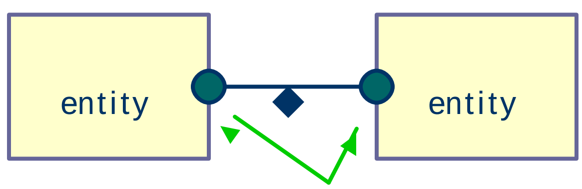
logical structure (Abstract Syntax, Concrete Syntax, Syntactic Transformations)

meaning (Type System, Component Semantics, Interaction Semantics)



## 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)

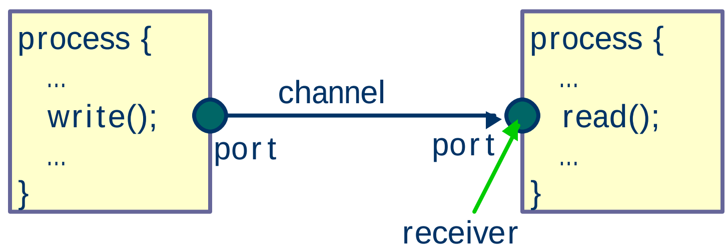


## Component Semantics

Entities are:

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

## One Class of Semantic Models: Producer / Consumer




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



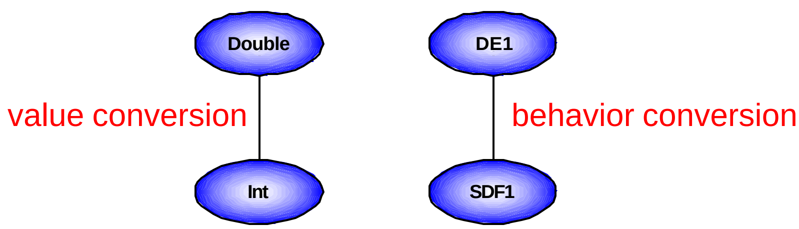
## Particular Consumer/Producer Frameworks (*Domains*)

- 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!

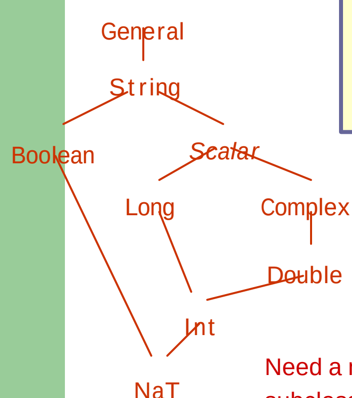
## Interfaces

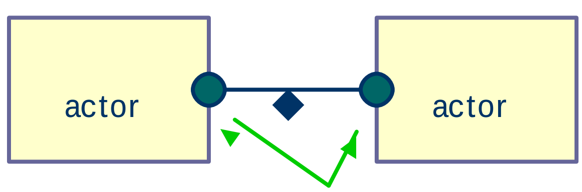
- Represent not just data types, but interaction types as well.






## GSRC Current Approach – System-Level Types

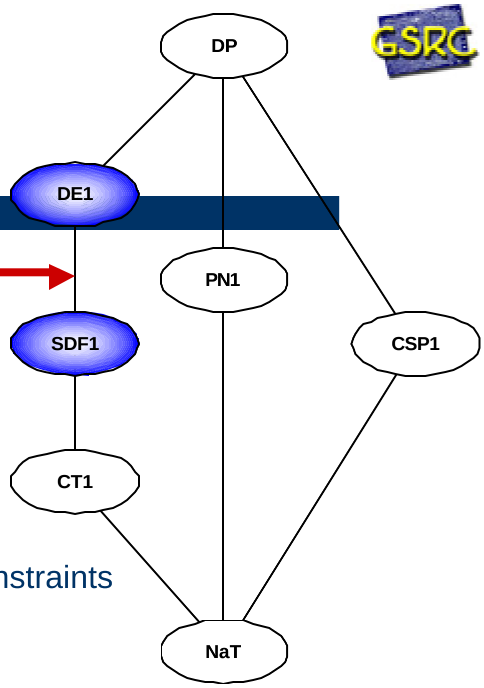



represent interaction semantics as types on these ports.

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



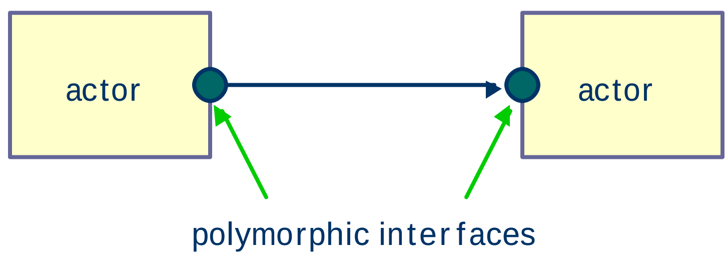
## System-Level Types

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

Benefits:

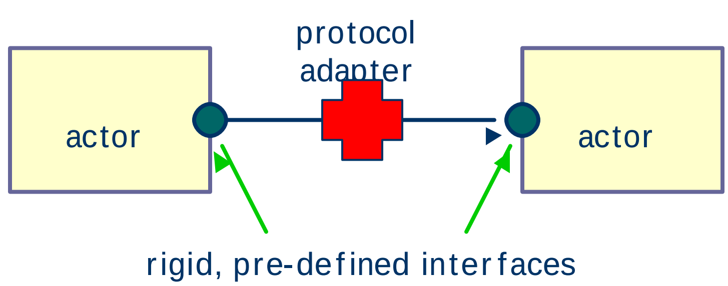
- Ensure component compatibility
- Clarify interfaces
- Provide the vocabulary for design patterns
- Detect errors sooner
- Promote modularity
- Promote polymorphic component design

# Our Hope –
# Polymorphic Interfaces

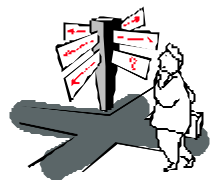





# Approach Used by Others –
# Interface Synthesis



## 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