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

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

http://www.gigascale.org
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.

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

No single facet is sufficient
Interactions

- Consistent?
- Modify one facet

system

facet

facet

facet
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 $Frame_m$. 
A **hierarchy** $h$ on the frame $f$:

- $\text{Entities}_h$, a set;
- $\text{ContainedPorts}_h : \text{Entities}_h \rightarrow \mathcal{P}(\text{Ports}_f)$;
- $\text{ContainedRelations}_h : \text{Entities}_h \rightarrow \mathcal{P}(\text{Relations}_f)$;
- $\text{ContainedEntities}_h : \text{Entities}_h \rightarrow \mathcal{P}(\text{Entities}_f)$;

**Constraints:**
- A port cannot be contained by more than one 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.
<?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>
Modeling Markup Language

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

[Diagram of class relationships and attributes]

Hierarchy:
- entities(): Set of Entities
- model(): Model

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

Port:
- frame(): Frame
- links(): Set of Ports

Relation:
- frame(): Frame
- links(): Set of Ports

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

Link:
- frame(): Frame
- port(): Port
- relation(): Relation
Where We Are...

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

---

logical structure

meaning
Type Systems

Type lattice represents subclassing & ad-hoc convertibility.

need compatible data types
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

- 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.

```
value conversion

Double

Int

behavior conversion

DE1

SDF1
```
Current Approach – System-Level Types

- General
  - String
- Boolean
  - Scalar
    - Long
    - Complex
      - Double
  - Int
- NaT

Actors 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

Input alphabet:
g: get
p: put
h: hasToken

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

This automaton simulates the previous one

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

actor

polymorphic interfaces

actor
Alternative Approach – Interface Synthesis

actor \hspace{1cm} \text{protocol adapter} \hspace{1cm} \text{rigid, pre-defined interfaces} \hspace{1cm} \text{actor}
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.

http://www.gigascale.org/semantics