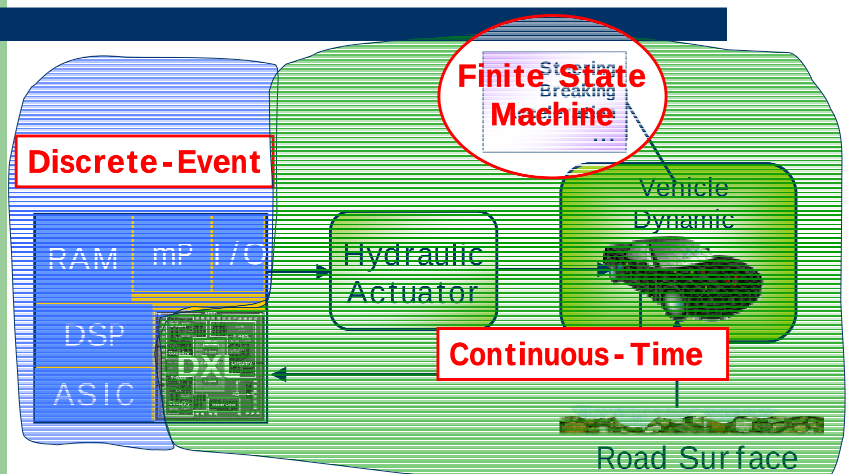
UC Berkeley
Mobies Technology Project

Process-Based Software
Components for Networked
Embedded Systems

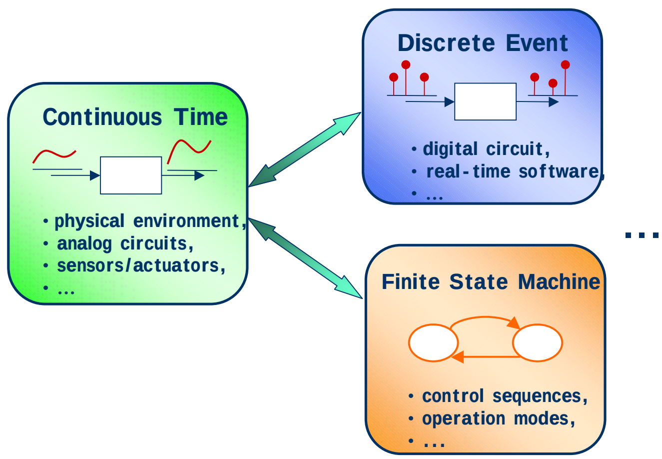
PI: Edward Lee
CoPI: Tom Henzinger

Heterogeneous Modeling

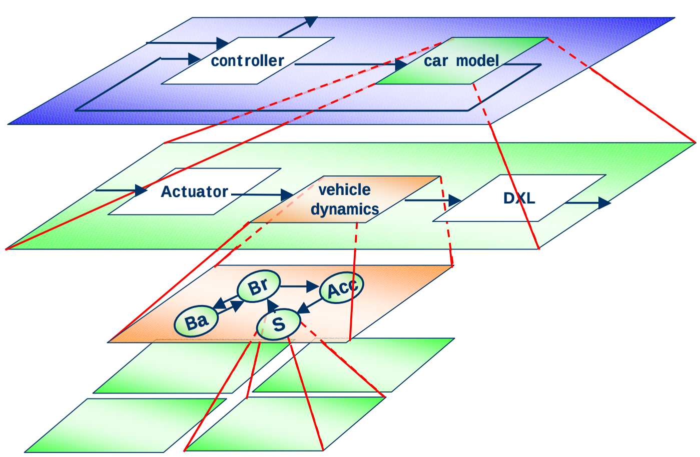


Example:An Automotive Active-Suspension System

Continuous & Discrete Dynamics

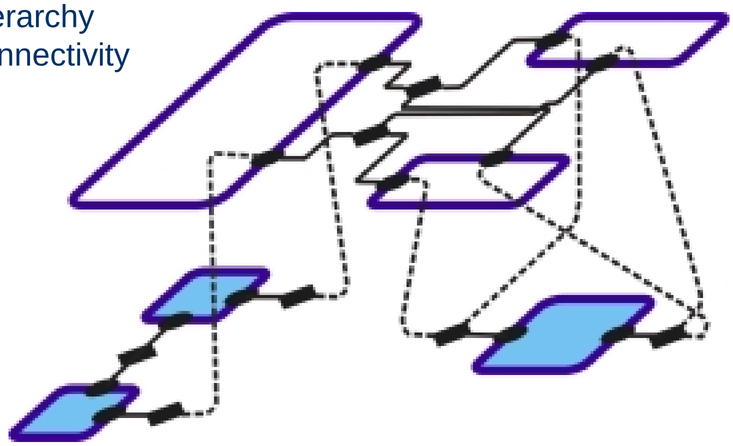


Components and Composition



Abstract Syntax for Component-Based Design

hierarchy
connectivity



Not Abstract Syntax

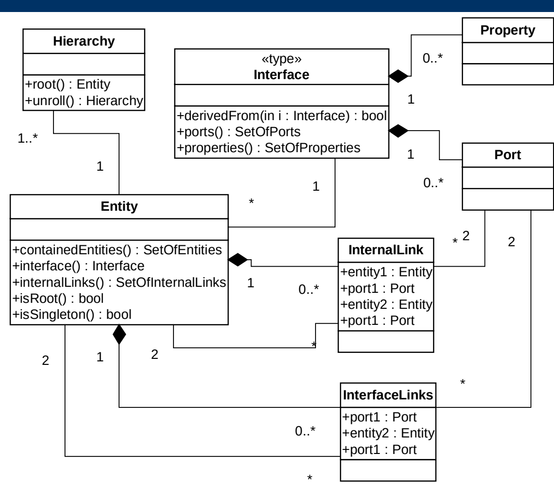
- Semantics of component interactions
- Type system
- File format (a concrete syntax)
- API (another concrete syntax)

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

The GSRC Abstract Syntax

- Models hierarchical connected components
  - block diagrams, object models, state machines, …
  - abstraction and refinement
  - recursive constructs
- Supports classes and instances
  - object models
  - inheritance
  - static and instance variables
- Specified by concrete syntaxes
  - sets and functions
  - UML object model
  - XML file format

Abstract Syntax Object Model



Constraints (in OCL?):

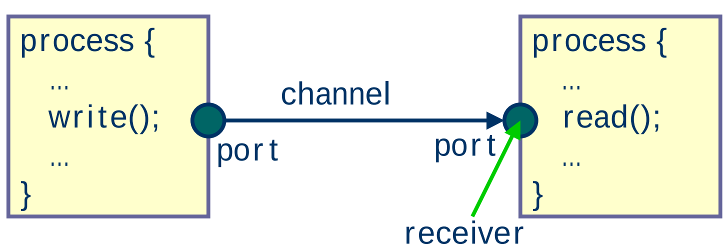
- Links do not cross levels of hierarchy
- If interface *i* is derived from *j* then it inherits its ports and properties
- An *instance hierarchy* has only singleton entities.
- A *role hierarchy* has some non-singleton entities.

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?

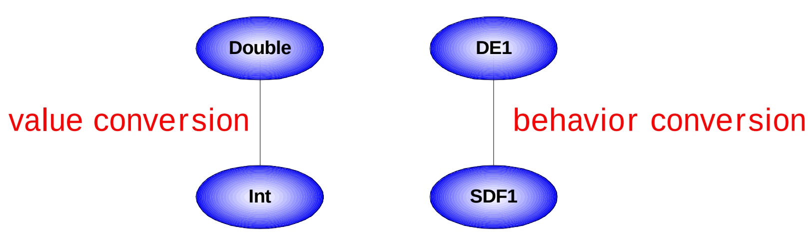
Domains – Provide semantic models for component interactions

- 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
- PS – publish-and-subscribe

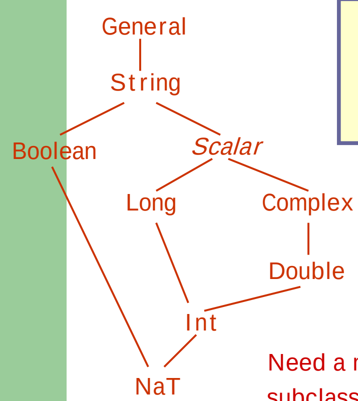
Each of these defines a component ontology and an interaction semantics between components. There are many more possibilities!

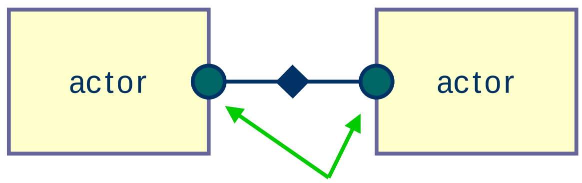
Component Interfaces

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



Approach – System-Level Types



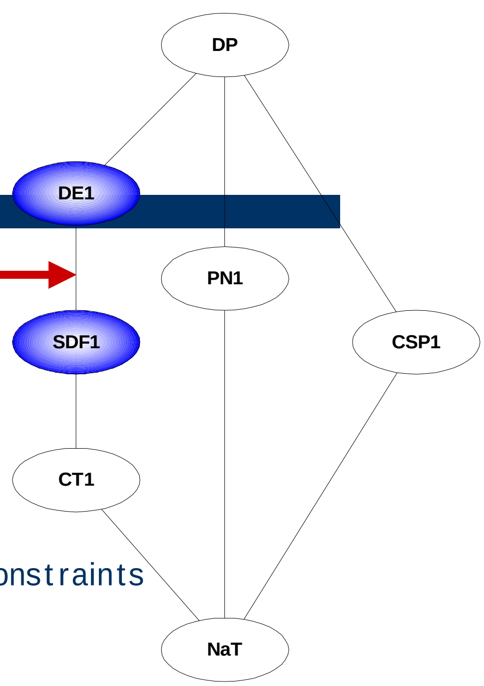


represent interaction semantics as types on these ports.

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

Type Lattice

Simulation relation

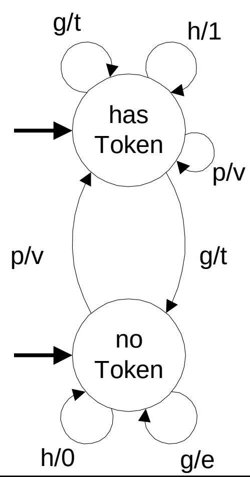


Achievable properties:

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

SDF Receiver Type Signature

SDF1



Input alphabet:

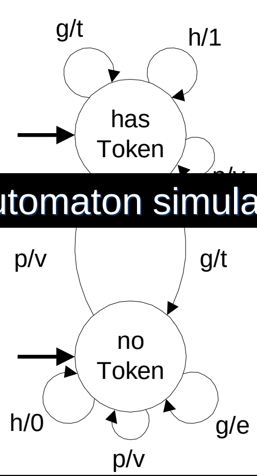
g: get
p: put
h: hasToken

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

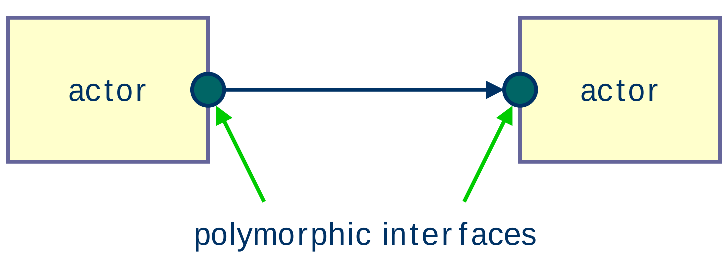
System-Level Types

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

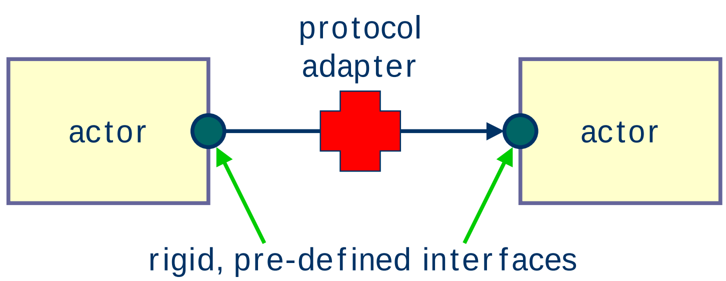
Benefits:

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

Our Hope – Polymorphic Interfaces



More Common Approach – Interface Synthesis



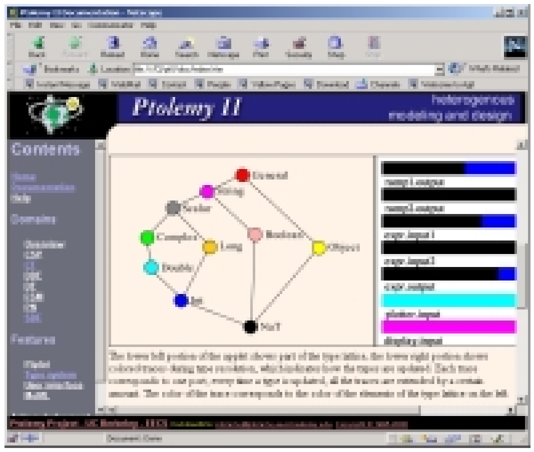
Ptolemy II – A Starting Point?

Ptolemy II –

- Java based, network integrated
- Many domains implemented
- Multi-domain modeling
- XML syntax for persistent data
- Block-diagram GUI
- Extensible type system
- Code generator on the way

http://ptolemy.eecs.berkeley.edu

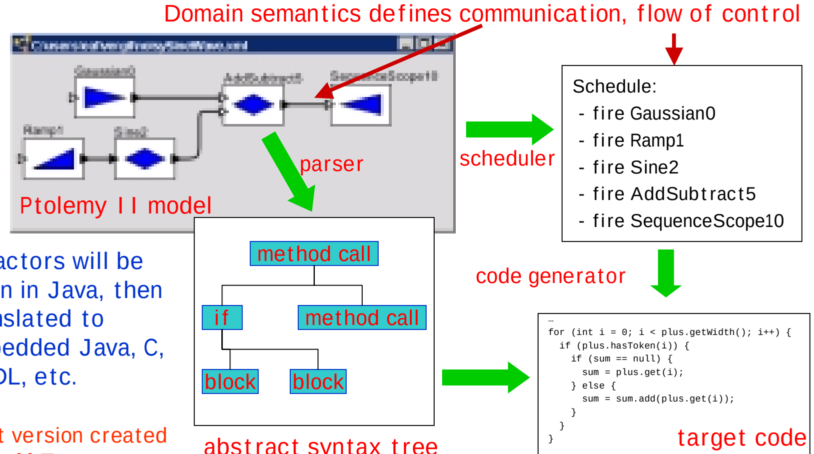
Type System Infrastructure



Ptolemy II has an extensible type system infrastructure with a plug-in interface for specifying a type lattice. At the left, an applet illustrates type resolution over a (simplified) type lattice representing data types exchanged between actors.

Nascent Generator Infrastructure

Domain semantics defines communication, flow of control



All actors will be given in Java, then translated to embedded Java, C, VHDL, etc.

First version created by Jeff Tsay.

Generator Approach

- Actor libraries are built and maintained in Java
  - more maintainable, easier to write
  - polymorphic libraries are rich and small
- Java + Domain translates to target language
  - concurrent and imperative semantics
- Efficiency gotten through code transformations
  - specialization of polymorphic types
  - code substitution using domain semantics
  - removal of excess exception handling

Code transformations (on AST)

```
// Original actor source
Token t1 = in.get(0);
Token t2 = in.get(1);
out.send(0, t1.multiply(t2));
```

specialization of Token declarations

```
// With specialized types
IntMatrixToken t1 = in.get(0);
IntMatrixToken t2 = in.get(1);
out.send(0, t1.multiply(t2));
```

The Ptolemy II type system supports polymorphic actors with propagating type constraints and static type resolution. The resolved types can be used in optimized generated code.

See Jeff Tsay, *A Code Generation Framework for Ptolemy II*

Code transformations (on AST)

```
// With specialized types
IntMatrixToken t1 = in.get(0);
IntMatrixToken t2 = in.get(1);
out.send(0, t1.multiply(t2));
```

Domain-polymorphic code is replaced with specialized code. Extended Java (from Titanium project) treats arrays as primitive types.

↓ transformation using domain semantics

```
// Extended Java with specialized communication
int[][] t1 = _inbuf[0][_inOffset = (_inOffset+1)%5];
int[][] t2 = _inbuf[1][_inOffset = (_inOffset+1)%5];
_outbuf[_outOffset = (_outOffset+1)%8] = t1 + t2;
```

See Jeff Tsay, *A Code Generation Framework for Ptolemy II*

Code transformations (on AST)

```
// Extended Java with specialized communication
int[][] t1 = _inbuf[0][_inOffset = (_inOffset+1)%5];
int[][] t2 = _inbuf[1][_inOffset = (_inOffset+1)%5];
_outbuf[_outOffset = (_outOffset+1)%8] = t1 + t2;
```

↓ convert extended Java to ordinary Java

```
// Specialized, ordinary Java
int[][] t1 = _inbuf[0][_inOffset = (_inOffset+1)%5];
int[][] t2 = _inbuf[1][_inOffset = (_inOffset+1)%5];
_outbuf[_outOffset = (_outOffset+1)%8] =
    IntegerMatrixMath.multiply(t1, t2);
```

See Jeff Tsay, *A Code Generation Framework for Ptolemy II*

Near-Term Goals

- Interface definitions for relevant domains
  - Those with potential for real-time execution
- Abstraction of real-time properties
  - requirements and performance
- Evolution of generator infrastructure
  - Demonstrate synthesis of embedded Java
- Explore real-time Java
  - Better safety, network integration

Process

- Website shared with Phase II Berkeley project
  - mailing list with archiving
  - discussion forums
  - CVS archive
- Quasi-weekly meetings
  - Comparing software architectures
  - Comparing approaches
- Software
  - nightly builds
  - automated test suite
  - design and code reviews
  - UML modeling
- Embedded systems lab
  - Construction starts in July