Advanced Tool Architectures Supporting Interface-Based Design

Presented by
Edward A. Lee
Chess, UC Berkeley

NSF

UC Berkeley: Chess
Vanderbilt University: ISIS
University of Memphis: MSI

Foundations of Hybrid and Embedded Software Systems

NSF ITR Deliverables

A set of reusable, inter-operating software modules, freely distributed as open-source software. These modules will be toolkits and frameworks that support the design of embedded systems, provide infrastructure for domain-specific tools, and provide model-based code generators.

The starting point is a family of *actor-oriented* modeling tools and associated meta modeling tools.

Tool Architectures

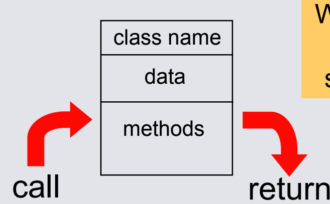
- Objective is to unify:
  - modeling
  - specification
  - design
  - programming

All of these tasks are accomplished by the *system designers.*

- Define modeling & design "languages" with:
  - syntaxes that aid understanding
  - composable abstractions
  - understandable concurrency and time
  - predictable behavior
  - robust behavior

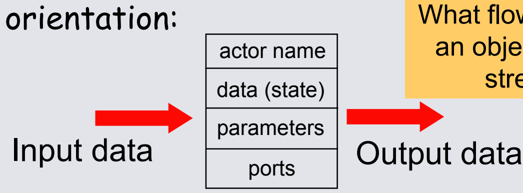
Actor-Oriented Design

- Object orientation:



What flows through an object is sequential control

- Actor orientation:



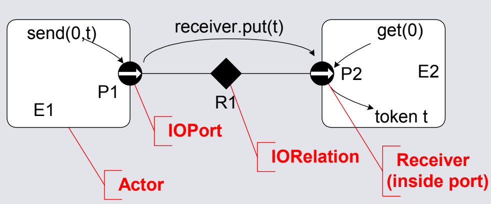
What flows through an object is data streams

Examples of Actor-Oriented Component Frameworks

- Simulink (The MathWorks)
- Labview (National Instruments)
- Modelica (Linkoping)
- OCP, open control platform (Boeing)
- GME, actor-oriented meta-modeling (Vanderbilt)
- SPW, signal processing worksystem (Cadence)
- System studio (Synopsys)
- ROOM, real-time object-oriented modeling (Rational)
- Port-based objects (U of Maryland)
- I/O automata (MIT)
- VHDL, Verilog, SystemC (Various)
- Polis & Metropolis (UC Berkeley)
- Ptolemy & Ptolemy II (UC Berkeley)
- …

Actor View of Producer/Consumer Components

Basic Transport:



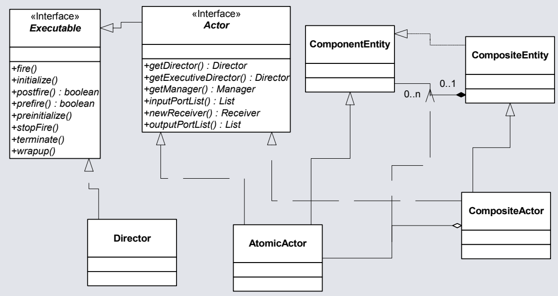
Models of Computation:

- continuous-time
- dataflow
- rendezvous
- discrete events
- synchronous
- time-driven
- publish/subscribe
- …

Key idea: The *model of computation* defines the component interaction patterns and is part of the framework, not part of the components themselves.

Object-Oriented and Actor-Oriented Design

- Object orientation:
  - strong typing
  - inheritance
  - procedural interfaces
- Actor orientation
  - concurrency
  - communication
  - real time
- These are complementary



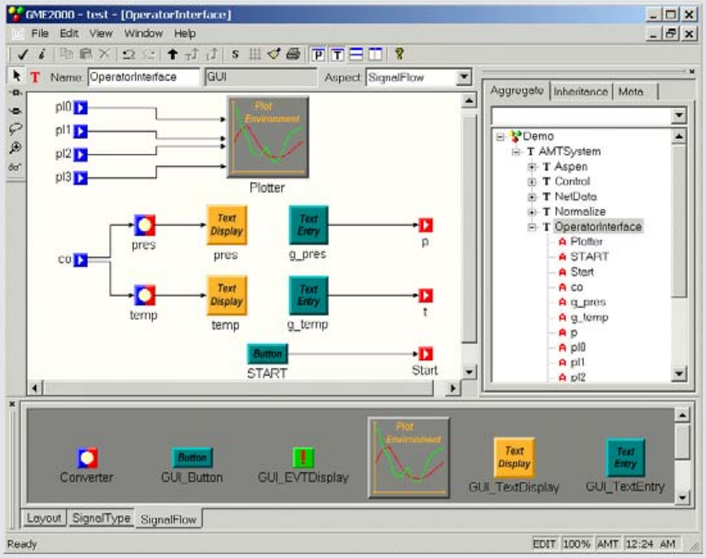
UML object model emphasizes static structure.

Actor orientation offers:
- modeling the continuous environment (and hybrid systems)
- understandable concurrency (vs. RPC, semaphores, and mutexes)
- specifications of temporal behavior (vs. “prioritize and pray”)

Two of Our Tool Starting Points

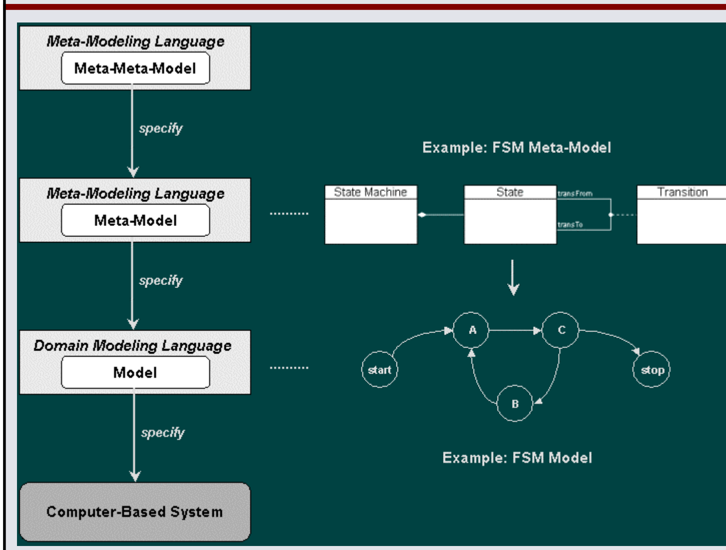
- GME: Generic Modeling Environment
  - Vanderbilt ISIS
  - Meta modeling of actor-oriented modeling
  - Proven for representing “abstract syntax” (called by some “static semantics”)
- Ptolemy II
  - UC Berkeley Chess
  - Framework for exploring actor-oriented semantics
  - Beginnings of meta modeling of actor-oriented “abstract semantics”

Actor-Oriented Modeling in GME



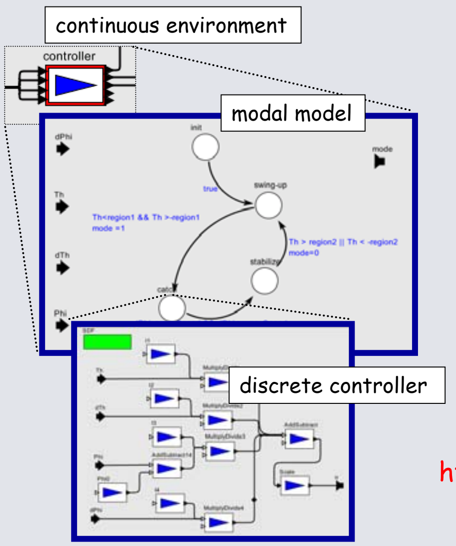
Domain-specific actor-oriented modeling environments are created from meta models, and a sophisticated, domain-specific UI is generated from those models.

Meta Modeling in GME



Meta models consist of UML object models enriched by OCL constraints which capture structural properties shared by a family of models.

Ptolemy II



example Ptolemy model: hybrid control system

A laboratory supporting experimentation with actor-oriented design, concurrent semantics, and visual syntaxes.

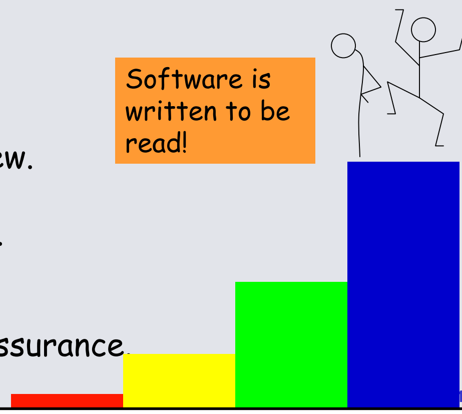
http://ptolemy.eecs.berkeley.edu

Software Practice

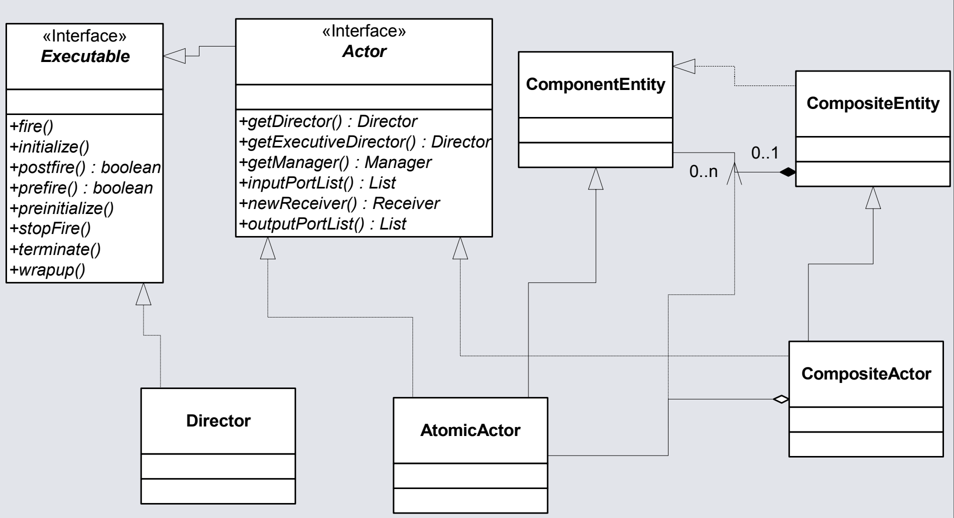
- Ptolemy II and GME are widely recognized to be unusually high quality software from a research group.
- Software practice in the Ptolemy Project:
  - Object models in UML
  - Design patterns
  - Layered software architecture
  - Design and code reviews
  - Design document
  - Nightly build
  - Regression tests
  - Sandbox experimentation
  - Code rating

Code rating

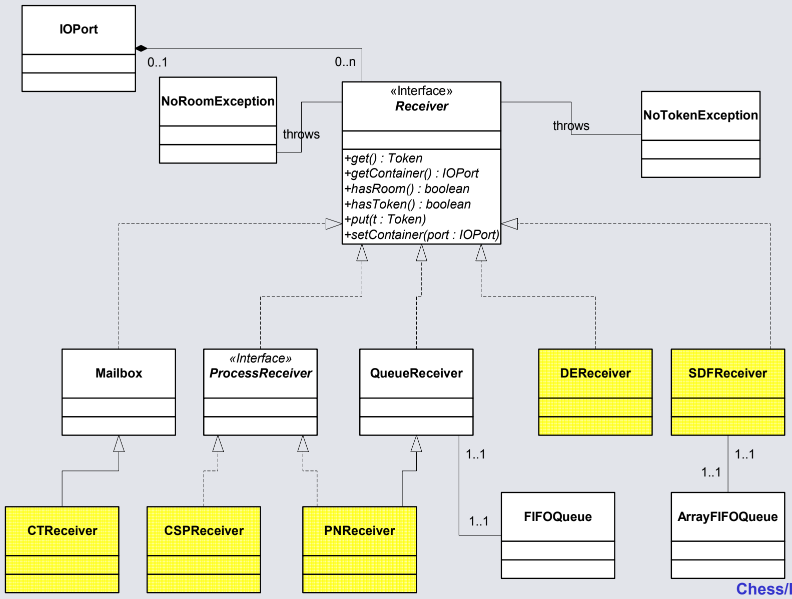
- A simple framework for
  - quality improvement by peer review
  - change control by improved visibility
  - encouraging innovation
- Four confidence levels
  - Red. No confidence at all.
  - Yellow. Passed design review. Soundness of the APIs.
  - Green. Passed code review. Quality of implementation.
  - Blue. Passed final review. Backwards-compatibility assurance.



Modeling Semantics in Ptolemy II – Object Model for Executable Components



Communication Protocols – Object Model for Messaging Framework



Structuring This Space with Interface Theories

- Concept of *Interface Theories* is due to Tom Henzinger and his colleagues.

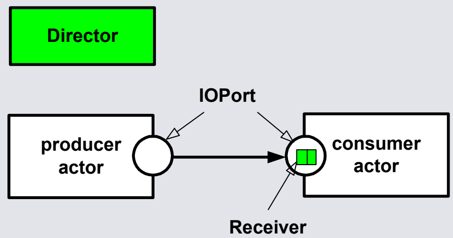
- We are using this concept to figure out what the Ptolemy Group has done with its software prototypes.

Receiver Interface – Software Architecture Perspective

«Interface»
Receiver

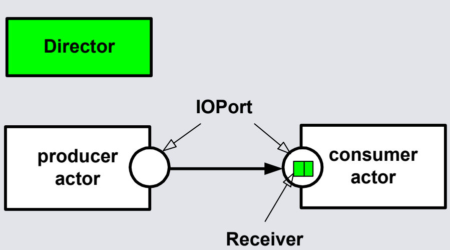
+get() : Token
+getContainer() : IOPort
+hasRoom() : boolean
+hasToken() : boolean
+put(t : Token)
+setContainer(port : IOPort)

These polymorphic methods implement the communication semantics of a *domain* in Ptolemy II. The receiver instance used in communication is supplied by the director, not by the component.



Behavioral Types – Interface Theory Perspective

- Capture the dynamic interaction of components in *types*
- Obtain benefits analogous to data typing.
- Call the result *behavioral types*.

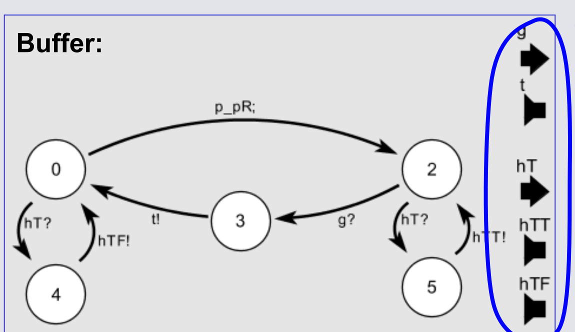


- Communication has
  - data types
  - behavioral types
- Components have
  - data type signatures
  - behavioral type signatures
- Components are
  - data polymorphic
  - domain polymorphic

A Preliminary Behavioral Type System

- Based on *interface automata*
  - Proposed by de Alfaro and Henzinger
  - Concise composition (vs. standard automata)
  - *Alternating simulation* provides contravariant inputs/outputs
- Compatibility checking
  - Done by automata composition
  - Captures the notion “components can work together”
- Alternating simulation (from Q to P)
  - All input steps of P can be simulated by Q, and
  - All output steps of Q can be simulated by P.
  - Provides the ordering we need for subtyping & polymorphism

Simple Example: One Place Buffer Showing Consumer Interface Only



consumer interface

Model of the interaction of a one-place buffer, showing the interface to a consumer actor.

Inputs:

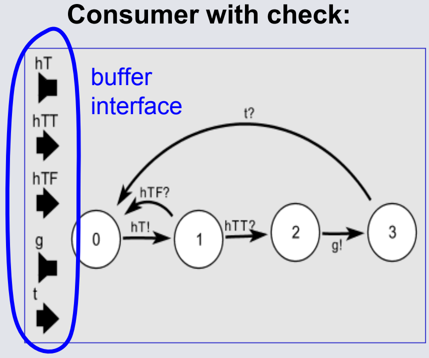
g	get
hT	hasToken

Outputs:

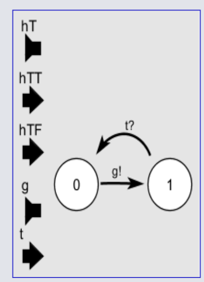
t	Token
hTT	Return True from hasToken
hTF	Return False from hasToken

Two Candidate Consumer Actors

Consumer with check:



Consumer without check:



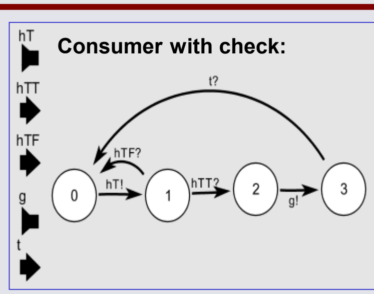
Inputs:

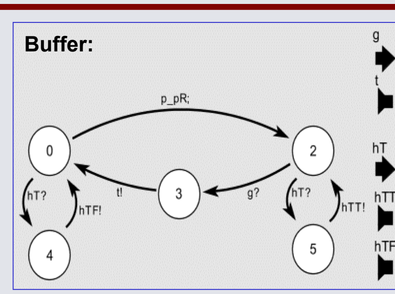
t	Token
hTT	Return True from hasToken
hTF	Return False from hasToken

Outputs:

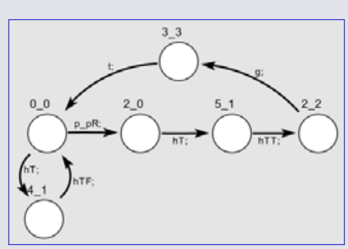
g	get
hT	hasToken

Composition: Behavioral Type Check



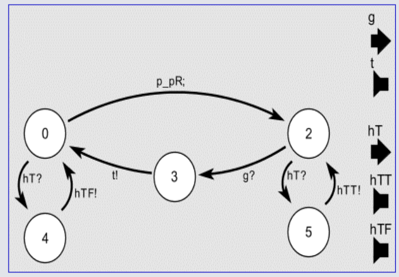


Illegal states are pruned out of the composition. A composite state is illegal if an output produced by one has no corresponding input in the other.

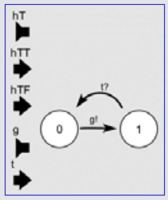


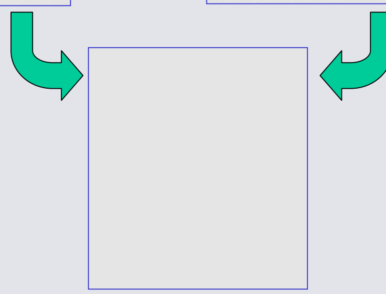
Composition: Behavioral Type Check

Buffer:



Consumer without check:



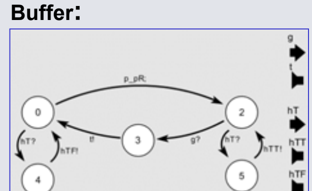


An empty composition means that all composite states are illegal. E.g., here, 0_0 is illegal, which results in pruning all states.

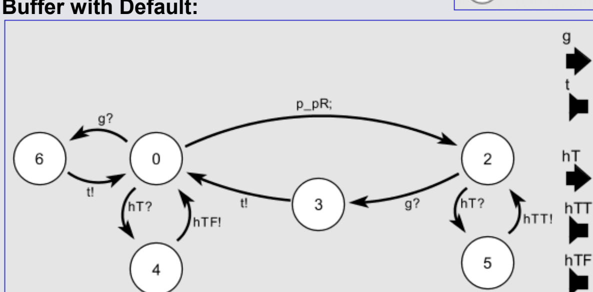
Subclassing and Polymorphism

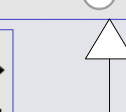
We can construct a type lattice by defining a partial order based on alternating simulation. It properly reflects the desire for contravariant inputs and outputs.

Buffer:



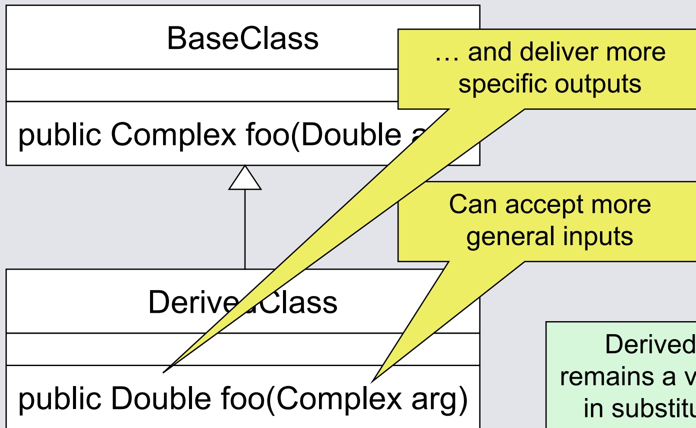
Buffer with Default:





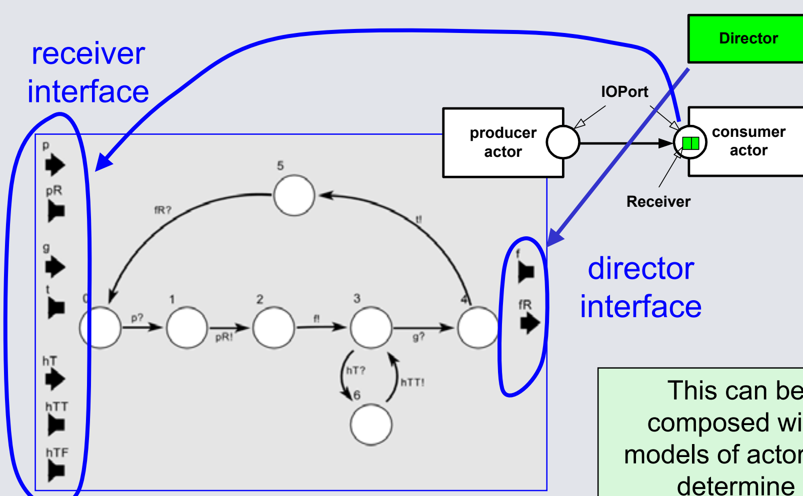
Alternating simulation relation

Contravariance of Inputs and Outputs in a Classical Type System



DerivedClass remains a valid drop-in substitution for BaseClass.

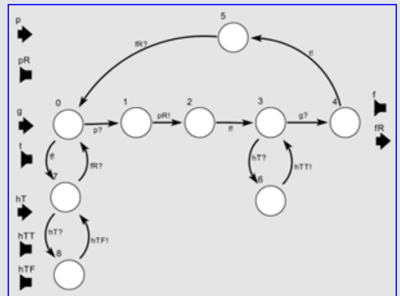
Representing Models of Computation Synchronous Dataflow (SDF) Domain



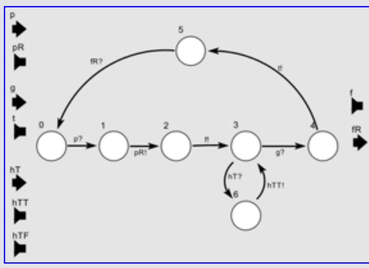
This can be composed with models of actors to determine compatibility.

Subtyping Relation Between Models of Computation: SDF $\leq$ DE

DE Domain



SDF Domain



This enables the design of components that can operate within multiple models of computation ("domain polymorphic components")

Summary of Behavioral Types – Preliminary Results

- We capture patterns of component interaction in a type system framework: *behavioral types*
- We describe interaction types and component behavior using *interface automata*.
- We do type checking through *automata composition* (detect component incompatibilities)
- Subtyping order is given by the alternating simulation relation, supporting *polymorphism*.
- A *behavioral type system* is a set of automata that form a lattice under alternating simulation.

Scalability

- Automata represent behavioral types
  - Not arbitrary program behavior
  - Descriptions are small
  - Compositions are small
  - Scalability is probably not an issue
- Type system design becomes an issue
  - What to express and what to not express
  - Restraint!
    - Will lead to efficient type check and type inference algorithms

Issues and Ideas

- Composition by name-matching
  - awkward, limiting.
  - use ports in hierarchical models?
- Rich subtyping:
  - extra ports interfere with alternating simulation.
  - projection automata?
  - use ports in hierarchical models?
- Synchronous composition:
  - composed automata react synchronously.
  - modeling mutual exclusion is awkward
  - use transient states?
  - hierarchy with transition refinements?

More Speculative

- We can reflect component dynamics in a run-time environment, providing *behavioral reflection.*
  - admission control
  - run-time type checking
  - fault detection, isolation, and recovery (FDIR)
- Timed interface automata may be able to model *real-time* requirements and constraints.
  - checking consistency becomes a type check
  - generalized schedulability analysis
- Need a *language* with a behavioral type system
  - Visual syntax given here is meta modeling
  - Use this to build domain-specific languages

Conclusions

- You can expect from this team:
  - Sophisticated software
  - High quality, open-source software
  - Domain-specific modules
  - Generators for domain-specific modules
- Emphasis on:
  - Meta modeling of abstract syntax
  - Meta modeling of semantics
  - Actor-oriented design methods
  - Interface definitions
  - Composable models