Project Goals and Problem Description

Our focus is on component-based design using principled models of computation and their runtime environments for embedded systems. The emphasis of this project is on the dynamics of the components, including the communication protocols that they use to interface with other components, the modeling of their state, and their flow of control. The purpose of the mechanisms we develop is to improve robustness and safety while promoting component-based design.

Where We Are

- Major Accomplishments
  - Actor-oriented design
  - Behavioral types
  - Component specialization (vs. code generation)
  - Hierarchical heterogeneous models
  - Hierarchical modal models
  - Hybrid systems operational semantics
  - Hybrid system modeling and simulation with HSIF import
  - Giotto and timed-multitasking models of computation
  - Network integrated models (P&S, push-pull, discovery)
  - Actor definition language principles
Where We Are Going

- Current Efforts
  - Actor-oriented classes, inheritance, interfaces and aspects
  - Security with distributed and mobile models
  - Higher-order components
  - Model-based lifecycle management
  - Behavioral and resource Interfaces for practical verification
  - Modeling of Wireless Sensor nets
  - Construction of Scientific Workflows

Platforms

A platform is a set of designs.

Relations between platforms represent design processes.
Progress

Many useful technical developments amounted to creation of new platforms.

- microarchitectures
- operating systems
- virtual machines
- processor cores
- configurable ISAs

Desirable Properties

From above:
- modeling
- expressiveness

From below:
- correctness
- efficiency
Model-Based Design

Model-based design is specification of designs in platforms with “useful modeling properties.”

Recent Action

Giving the red platforms useful modeling properties (e.g. verification, SystemC, UML, MDA)

Getting from red platforms to blue platforms (e.g. correctness, efficiency, synthesis of tools)
Better Platforms

Platforms with modeling properties that reflect requirements of the application, not accidental properties of the implementation.

How to View This Design

From above: Signal flow graph with linear, time-invariant components.

Figure C.12: A block diagram generating a plucked string sound with a fundamental and three harmonics.

From below: Synchronous concurrent composition of components
Embedded Platforms

The modeling properties of these platforms are about the application problem, not about the implementation technology.
Actor-Oriented Platforms

*Actor oriented* models compose concurrent components according to a model of computation.

Time and concurrency become key parts of the programming model.

Actor-Oriented Design

**Object orientation:**

- What flows through an object is sequential control

**Actor orientation:**

- What flows through an object is streams of data
Actor Orientation vs. Object Orientation

Identified problems with object orientation:
- Says little or nothing about concurrency and time
- Concurrency typically expressed with threads, monitors, semaphores
- Components tend to implement low-level communication protocols
- Re-use potential is disappointing

Actor orientation offers more potential for useful modeling properties, and hence for model-based design.

But… New Design Methods Need to Offer Best-Of-Class Methods

- Abstraction
  - procedures/methods
  - classes
- Modularity
  - subclasses
  - inheritance
  - interfaces
  - polymorphism
  - aspects
- Correctness
  - type systems
Example of an Actor-Oriented Framework: Simulink

Less Conventional Actor-Oriented Framework: VisualSense

- Based on Ptolemy II
- Connectivity is wireless
- Customized visualization
- Location-aware models
- Channel models include:
  - packets losses
  - power attenuation
  - distance limitations
  - collisions
- Component models include:
  - Antenna gains
  - Terrain models
  - Jamming
Also Uses Hierarchy for Abstraction

These 49 sensor nodes are actors that are instances of the same class, defined as:

Observation

By itself, hierarchy is a very weak abstraction mechanism.
Tree Structured Hierarchy

- Does not represent common class definitions. Only instances.
- Multiple instances of the same hierarchical component are copies.

leaf components: instances of an OO class

Using Copies for Instances is Awkward

Models become unmaintainable. Changes have be performed on all 49 copies of this:
Alternative Hierarchy: Roles and Instances

- Role hierarchy ("design-time" view)
- Instance hierarchy ("run time" view)

Role Hierarchy

- Multiple instances of the same hierarchical component are represented by *classes* with multiple containers.
- This makes hierarchical components more similar to leaf components.
Example

- Ptolemy II now supports a role hierarchy.
- The definition below is a class and objects at the left are instances, not copies.

Making these objects instances of a class rather than copies reduced the XML representation of the model from 1.1 Mbytes to 87 kBytes, and offered a number of other advantages.

Subclasses, Inheritance?

Interfaces, Subtypes? Aspects?

- Now that we have classes, can we bring in more of the modern programming world?
  - subclasses?
  - inheritance?
  - interfaces?
  - subtypes?
  - aspects?
Actor Interfaces: Ports and Parameters

Example:

Yes We Can!

Actor-oriented design:
- subclasses and inheritance
  - hierarchical models that inherit structure from a base class
- interfaces and subtypes
  - ports and parameters of actors form their interface
- aspects
  - heterarchical models interweave multiple hierarchies, providing true multi-view modeling.

- All of these operate at the abstract syntax level, and are independent of the model of computation, and therefore can be used with any model of computation! Thus, they become available in domain-specific actor-oriented languages.

These are a part of what the Berkeley Center for Hybrid and Embedded Software Systems (Chess) is doing.
Example
A Simple Resource Interface

*EnergyConsumer* interface has a single output port that produces a Double representing the energy consumed by a firing.

*Filter* interface for a stream transformer component.

```
in: Event
energy: Double
out: Double
```

```
in: Double
energy: Double
subtype relation
power: Double
out: Double
```

EnergyConsumingFilter composed interface.

---

Ethereal Sting OEP: Lessons on Dataflow Design Patterns

Input data sequence, at `samplingFrequency` Hz

```
Solution to E0 Challenge Problem

E1 Challenge Problem Components
```

Output data sequence, at detected baud rate. (not known *apriori*)

Lee, UC Berkeley 31

Lee, UC Berkeley 32
SDF is More Flexible Than We Realized

- The Synchronous Dataflow (SDF) model of computation can be easily augmented to make it much more expressive without sacrificing static analyzability.

- Model-based lifecycle management can provide systematic ways to construct supervisory structures (resource management, task management).

Dataflow Taxonomy

- Synchronous dataflow (SDF)
  - Balance equation formalism
  - Statically schedulable
  - Decidable resource requirements

- Heterochronous Dataflow (HDF)
  - Combines state machines with SDF graphs
  - Very expressive, yet decidable
  - Scheduling complexity can be high

- Boolean & Integer Dataflow (BDF, IDF)
  - Balance equations are solved symbolically
  - Turing-complete expressiveness
  - Undecidable, yet often statically schedulable

- Process Networks (PN)
  - Turing-complete expressiveness
  - Undecidable (schedule and resource requirements)
  - Thread scheduling with interlocks provably executes with bounded resources when that is possible.
SDF Principles

- Fixed production/consumption rates
- Balance equations (one for each channel):
  \[ f_A N = f_B M \]
- Schedulable statically
- Decidable:
  - buffer memory requirements
  - deadlock

Undecidability: What SDF Avoids (Buck ’93)

- Sufficient set of actors for undecidability:
  - boolean functions on boolean tokens
  - switch and select
  - initial tokens on arcs

- Undecidable:
  - deadlock
  - bounded buffer memory
  - existence of an annotated schedule
Resampling Design Pattern Using Token Routing

- This pattern requires the use of a semantically richer dataflow model than SDF because the BooleanSwitch is not an SDF actor.
- This has a performance cost and reduces the static analyzability of the model.

Resampling Design Pattern using Modal Models

- Uses transition refinements
- All SDF + FSM
- Can be captured in a *higher-order component* that makes the pattern easy to use.
Scalability of Visual Syntaxes
Iteration by Replication

- naïve approach:
  - 8 tones
  - 8 signal paths
- hard to build
- hardwired scale
- distributor:
  - converts an array of dimension 8 to a sequence of 8 tokens.

Scalability of Visual Syntaxes
Iteration by Dataflow

- Although sometimes useful, this design pattern has limitations:
  - array size must be statically fixed
  - actor to iterate must be stateless, or
  - desired semantics must be to carry state across array elements.
Structured Programming in SDF

- A library of actors that encapsulate common design patterns:
  - **IterateOverArray**: Serialize an array input and provide it sequentially to the contained actor.
  - **MapOverArray**: Provide elements of an array input to distinct instances of the contained actor.
  - Zip, Scan, Case, …
- Like the higher-order functions of functional languages, but unlike functions, actors can have state.
- The implementation leverages the *abstract semantics* of Ptolemy II.

Abstract Semantics – The Key To Hierarchical Heterogeneity

- Flow of control
  - **Initialization**
  - Execution
  - Finalization
- Communication
  - Structure of signals
  - Send/receive protocols
  - `preinitialize()`
    - declare static information, like type constraints, scheduling properties, temporal properties, structural elaboration
  - `initialize()`
    - initialize variables
Abstract Semantics – The Key To Hierarchical Heterogeneity

flow of control
- Initialization
- Execution
- Finalization

communication
- Structure of signals
- Send/receive protocols

iterate()

Lee, UC Berkeley 43

Abstract Semantics – The Key To Hierarchical Heterogeneity

flow of control
- Initialization
- Execution
- Finalization

communication
- Structure of signals
- Send/receive protocols

iterate()
- prefire()
- fire()
- postfire()
- stopFire()

Lee, UC Berkeley 44
Lifecycle Management

- It is possible to hierarchically compose the Ptolemy II abstract semantics.

- Actors providing common patterns:
  - `RunCompositeActor` is a composite actor that, instead of firing the contained model, executes a complete lifecycle of the contained model.
  - `ModelReference` is an atomic actor whose function is provided by a complete execution of a referenced model in another file or URL.

- Provides systematic approach to building systems of systems.

Hierarchical Composition of the Ptolemy II Abstract Semantics

- Flow of control:
  - Initialization
  - Execution
  - Finalization

- Communication:
  - Structure of signals
  - Send/receive protocols

`iterate()`, `prefire()`, `fire()`, `postfire()`, `stopFire()`
Conclusion

- We aren’t done yet…
- Stay tuned…