Concurrent Component Patterns, Models of Computation, and Types

Edward A. Lee
Yuhong Xiong

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley

Presented at Fourth Annual Workshop on New Directions in Software Technology (NDIST'01), St. John, US Virgin Islands, December 2001.

Pattern-Related Questions

- Are components active? If passive, how are they activated?
  - event driven
  - dataflow
  - time driven
  - synchronous

- How are multiple sources of stimulus merged?
  - nondeterministic merge
  - round robin
  - priorities
  - time stamps

- Are communications synchronous?
  - synchronous method calls
  - thread rendezvous
  - asynchronous with futures
  - asynchronous with feedback
**View of Concurrent Components:**

*Actors with Ports and Attributes*

**Key idea:** The model of computation is part of the framework within which components are embedded not part of the components themselves. It enforces patterns.

Models of Computation:
- Messaging schema
- Flow of control
- Concurrency

**Actor View of Producer/Consumer Components**

**Basic Transport:**

Models of Computation:
- continuous-time
- dataflow
- rendezvous
- discrete events
- synchronous
- time-driven
- publish/subscribe
  • …
Examples of Actor-Oriented Component Frameworks

- Simulink (The MathWorks)
- Labview (National Instruments)
- OCP, open control platform (Boeing)
- 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)
- ...

Contrast with Object Orientation

- Call/return imperative semantics
- Concurrency is realized by ad-hoc calling conventions
- Patterns are supported by futures, proxies, monitors

Object orientation emphasizes inheritance and procedural interfaces.
Actor orientation emphasizes concurrency and communication abstractions.
Actor Orientation with a Visual Syntax

Ptolemy II is an experimental framework supporting exploration of concurrent component models of computation.

Realization of a Model of Computation is a “Domain” in Ptolemy II

- The “laws of physics” of component interaction
  - communication semantics
  - flow of control constraints

  In astrophysics: a “domain” is a region of the universe where a certain set of “laws of physics” applies.

- Multiple domains may be combined hierarchically
  - depends on the concept of “domain polymorphism”
Ptolemy II Domains

- Define the flow(s) of control
  - “execution model”
  - Realized by a Director class
- Define communication between components
  - “interaction model”
  - Realized by a Receiver class

Example Domains

- Communicating Sequential Processes (CSP): rendezvous-style communication
- Process Networks (PN): asynchronous communication, determinism
- Synchronous Data Flow (SDF): stream-based communication, statically scheduled
- Discrete Event (DE): event-based communication
- Synchronous/Reactive (SR): synchronous, fixed point semantics
- Time Driven (Giotto): synchronous, time-driven multitasking
- Timed Multitasking (TM): priority-driven multitasking, deterministic communication
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 – Codification of Domain Semantics

- 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
  - domain type signatures

- Components are
  - data polymorphic
  - domain polymorphic

Second Version of a Behavioral Type System

- Based on Interface automata
  - Proposed by de Alfaro and Henzinger
  - Concise composition (vs. standard automata)
  - *Alternating simulation* provides contravariance

- 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

- Key theorem about compatibility and alternating simulation
Example: Synchronous Dataflow (SDF) Consumer Actor Type Definition

Such actors are passive, and assume that input is available when they fire.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>f fire</td>
<td>fR Return from fire</td>
</tr>
<tr>
<td>t Token</td>
<td>g get</td>
</tr>
<tr>
<td>hTT Return True from hasToken</td>
<td>hT hasToken</td>
</tr>
<tr>
<td>hTF Return False from hasToken</td>
<td></td>
</tr>
</tbody>
</table>

Type Definition – Synchronous Dataflow (SDF) Domain
Type Checking – Compose
SDF Consumer Actor with SDF Domain

Type Definition –
SDF Consumer Actor in SDF Domain

interface to producer actor

1. receives token from producer
2. accept token
3. internal action: fire consumer
4. internal action: call get()
5. internal action: get token
6. internal action: return from fire
Type Definition – Discrete Event (DE) Domain

- This domain may fire actors without first providing inputs

Recall Component Behavior
SDF Consumer Actor

1. is fired
2. calls get()
3. gets a token
4. returns
Type Checking – Compose
SDF Consumer Actor with DE Domain

- Empty automaton indicates incompatibility
- Composition type has no behaviors

Subtyping Relation
Alternating Simulation: SDF ≤ DE
System-Level Type Lattice – Defined by Alternating Simulation

- Consumer actor types
- Subtyping relation
- Shown here for a few Ptolemy II domains

If an actor is compatible with a certain type, it is also compatible with the subtypes

Type Definition – Domain Polymorphic Consumer Actor

This actor checks for token availability before attempting to get the token.
Domain Polymorphic Actor Composes with the DE Domain

Domain Polymorphic Actor Also Composes with the SDF Domain
Conclusion

- 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.

- Subtyping order is given by the alternating simulation relation, supporting polymorphism.

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 become a type check
  - generalized schedulability analysis