Java Code Generation

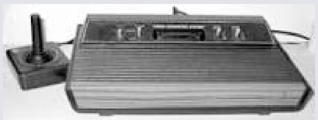
Steve Neuendorffer
UC Berkeley

5th Biennial Ptolemy Miniconference
Berkeley, CA, May 9, 2003

Outline

- Motivation
- Code generation architecture
- Component Specialization
  - Parameter
  - Type
  - Connection
  - Domain
- Token Unboxing and Obfuscation

Design Pressures



Market Customization

Increasing Complexity

Safety Requirements

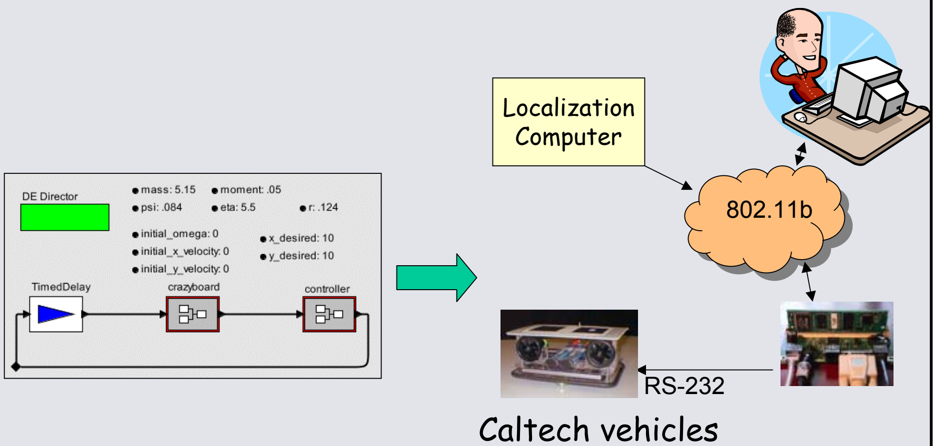
Design Reuse is Key!

Motivation

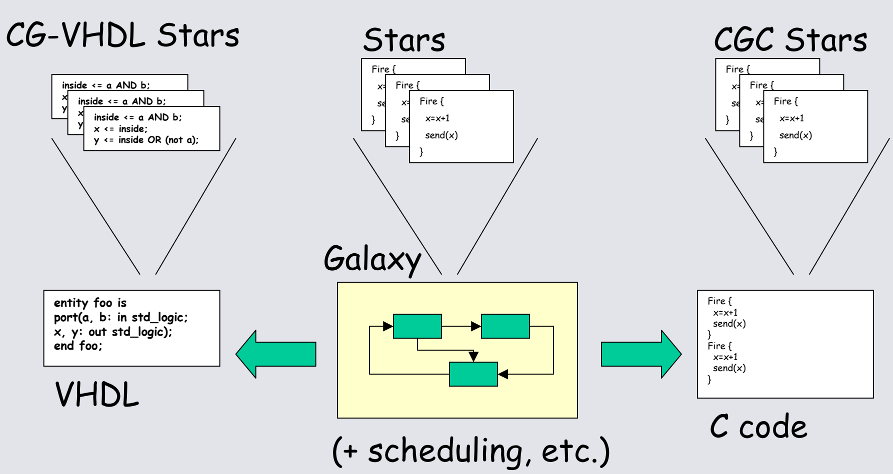
- System modeling using high-level components enables rapid prototyping
- System implementation becomes the bottleneck

Motivation

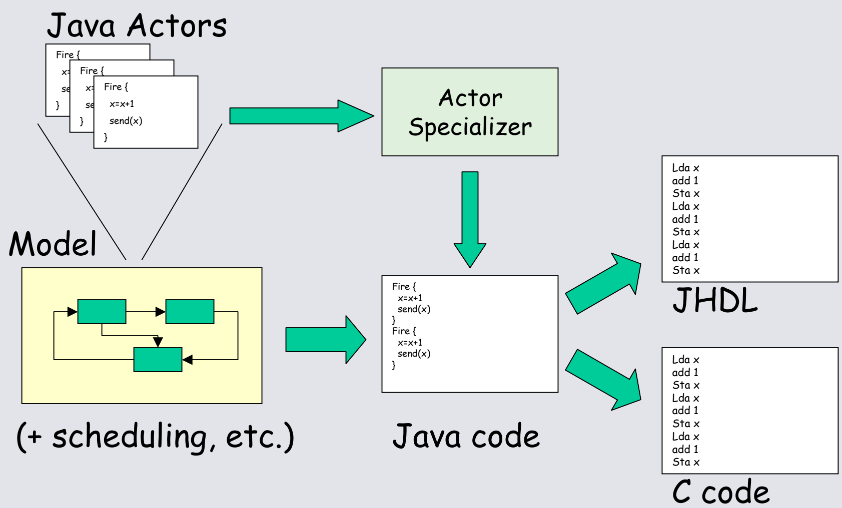
- Generation of hardware and software architectures for embedded systems



Ptolemy Classic

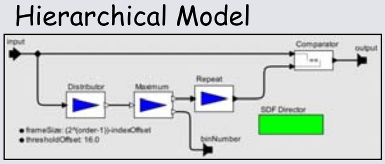


Ptolemy II



Component Specification

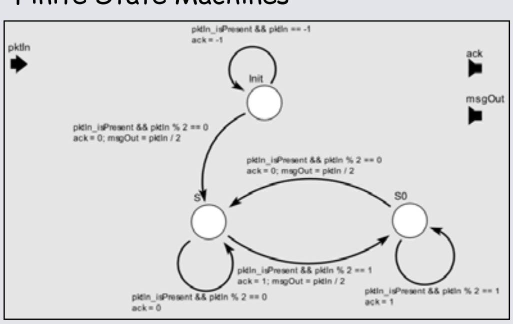
Hierarchical Model



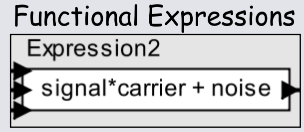
Java Code

```
public interface Executable {
 public boolean prefire() throws IllegalActionException;
 public void initialize() throws IllegalActionException;
 public void fire() throws IllegalActionException;
 …
}
```

Finite State Machines



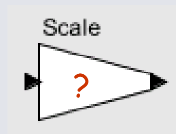
Functional Expressions



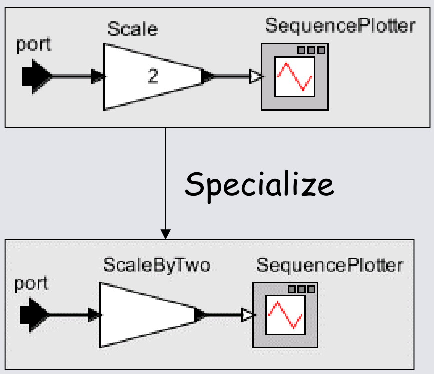
Special Purpose Languages

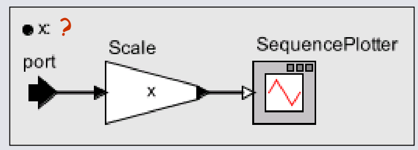
```
actor Switch [T] ()

  Integer Select, multi T Input ==> T Output :

  action Select: [i], Input: [a] at i ==> [a] end
end
```

Parameter Specialization



(Specified in Java code)

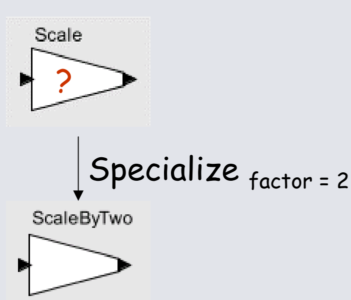


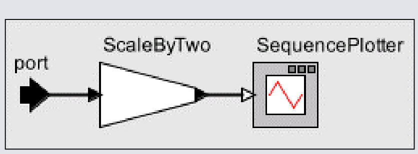


Here the scale factor has not been determined yet, because it depends on the parameter x.

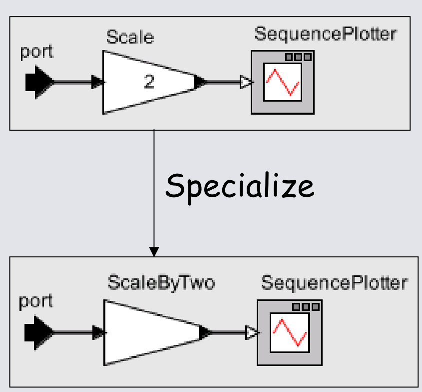
Implicit vs. Explicit information

Explicit Specialization





Implicit Specialization



Parameter Specialization

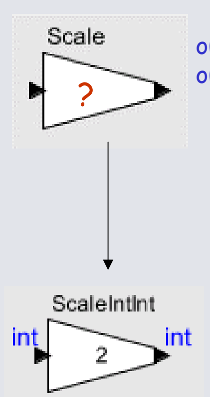
Implicit Parameter Specialization relies on model analysis to determine parameter values that set and cannot change.

Dynamic parameters:

- Parameters accessible through a user interface.
- Parameters that can be set in the FSM transitions.
- Parameters with values depending on unbound variables

All other parameters can be specialized using implicit context.

Type Specialization

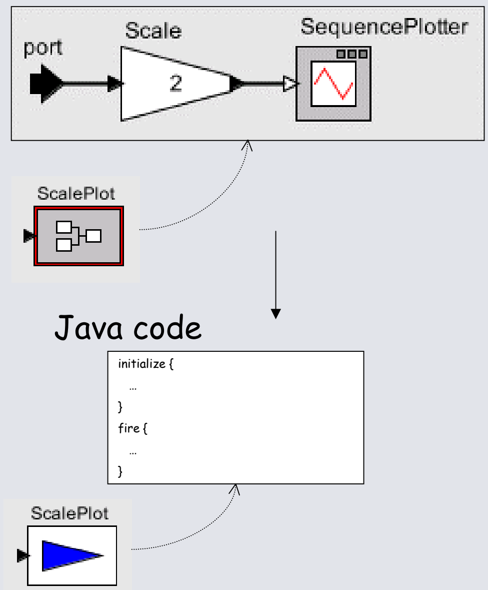


output > input
output > factor

Implicit analysis simply uses the standard type inference mechanism.

Currently assume that even when parameter *values* change, *types* do not.

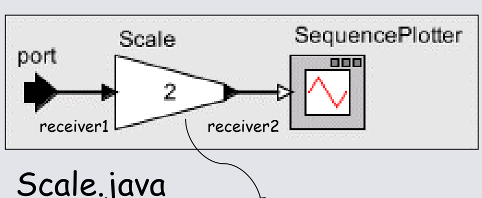
Aggregation



Parameter and Type specialization can be performed on individual actors.

Domain and Connection specialization occur as part of aggregation.

Connection Specialization



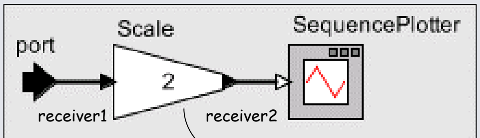
Scale.java

```
public void fire() {
  if (input.hasToken(0)) {
    Token in = input.get(0);
    Token factorToken =
           factor.getToken();
    Token result =
           in.multiply(factorToken);
    output.send(0, result);
  }
}
```

Connection specialization ties actors directly to the channels they are connected to.

Connections are assumed not to change.

Connection Specialization

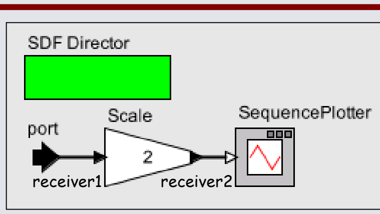



```
public void fire() {
  if (receiver1.hasToken()) {
    Token in = receiver1.get();
    Token factorToken =
           factor.getToken();
    Token result =
           in.multiply(factorToken);
     receiver2.put(result);
  }
}
```

Connection specialization ties actors directly to the channels they are connected to.

Connections are assumed not to change.

Domain Specialization

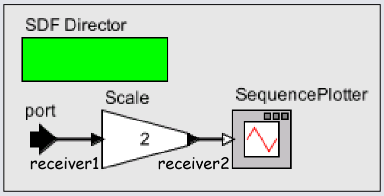



```
public void fire() {
  if (receiver1.hasToken()) {
    Token in = receiver1.get();
    Token factorToken =
           factor.getToken();
    Token result =
           in.multiply(factorToken);
     receiver2.put(result);
  }
}
```

Connection specialization ties actors directly to the channels they are connected to.

Domains are assumed not to change.

Domain Specialization




```
public void fire() {
  if (true) {
    Token in = receiver1._array[index1++];
     index1 = index1 %1;
    Token factorToken = factor.getToken();
    Token result =
            in.multiply(factorToken);
     receiver2._array[index2++] = result;
     index2 = index2 % 1;
  }
}
```

Connection specialization ties actors directly to the channels they are connected to.

Domains are assumed not to change.

Token Unboxing

```
public void fire() {
    int in = receiver1._array[index1];
    boolean inIsNull =
          receiver1._arrayIsNull[index1];
    index1 = index1++ % 1;
    int factorToken = factor;
    boolean factorTokenIsNull = false;
    int result = in*factorToken;
    boolean resultIsNull =
          inIsNull && factorTokenIsNull;
    receiver2._array[index2++] = result;
    receiver2._arrayIsNull[index2++] =
          resultIsNull;
    index2 = index2++ % 1;
}
```

- After specialization, memory use is a significant performance bottleneck.
- Token Unboxing removes allocation of token objects by replacing each token with its constituent fields.

Obfuscation

Java `.class` files contain a large number of strings

- String literals
- Class names
- Method signatures
- Field signatures
- Exception messages

Obfuscation renames these strings to shorter ones, where possible.

Reduces bytecode side.

Why does this all work?

- Ptolemy actor specifications are highly polymorphic and reusable.
- However, we commonly use them only in monomorphic contexts.
  - Constant, exactly analyzable types.
  - Connections, domains don't change.
  - Parameter values change only in known patterns.

Why does this all work?

- We've eliminated a large amount of synchronization overhead.
  - Workspace.getReadAccess()
  - Workspace.doneReading()
- We've eliminated object allocation, which reduces load on the garbage collector.
- Generated code is entirely self contained. Functionality is important, interfaces are not.

Capabilities

- Applications
  - Control algorithm for Caltech vehicles.
  - Rijndael encryption algorithm.
  - HTVQ Video compression.
- Supported
  - Expression actor
  - FSM actor
  - Modal models
  - SDF and Giotto domains
- Not supported
  - Record types
  - Transparent hierarchy

How to use

Command-line interface

```
>> copernicus model.xml
```

Code is generated in:

```
$PTII/ptolemy/copernicus/java/cg/model/
```

Vergil User interface

view -> Code Generator

Allows easier changing of parameters.

Conclusion

Java code generation is at the point where it might be useful for speeding up the simulation of some models.

Current work:

- Embedded Java platform
- Integration with hardware synthesis
- Guided refinement