Model-Based Code Generation is not a Replacement for Programming

Edward A. Lee

Robert S. Pepper Distinguished Professor
UC Berkeley

With thanks to Man-Kit Leung, Bert Rodier, Gang Zhou

Invited Talk

Workshop on Software Synthesis (part of ES Week)
Grenoble, France, October 16, 2009
It is programming!

- The language is different.
- The language has features well suited to express some things.
- The language is not well suited to express some other things.
- Recognizing the difference appears to be difficult.
The Problem

Students, professors, engineers, and even grownups will use whatever modeling tool they are familiar with for every task at hand, whether it is suitable or not.
Properties of Languages

- Modeling languages like Simulink
  - Concurrent
  - Timed
  - Express dynamics well
- Imperative languages like C
  - Sequential
  - Untimed
  - Express algorithms well
- Hybrid languages that mix imperative with threads like Java
  - Sequential unstructured nondeterminate concurrency
  - Untimed
  - Express few things well, unless you limit yourself to the sequential subset.

*Maybe the features of the first two should be mixed in a different way*
Respect for Imperative Reasoning!

- Imperative reasoning (algorithms, proofs, recipes, etc.) is unnatural to express in actor-oriented languages (as it is also in functional languages).
- Banning imperative reasoning does not seem like a good idea.
- Since people seem to insist on a homogeneous solution, the result is that only a tiny fraction of programmers use actor-oriented languages.

This can be fixed!
Our Proposal: Modeling Languages as Component Architectures rather than Languages

Established component architectures: Object-oriented:

- **class name**
- **data**
- **methods**

What flows through an object is sequential control

- Things happen to objects

Proposed component architectures: Actor oriented:

- **actor name**
- **data (state)**
- **parameters**
- **ports**

What flows through an object is evolving data

- Actors make things happen

Input data  Output data
Ptolemy II: Our Open-Source Laboratory for Experiments with Actor-Oriented Design

http://ptolemy.org

Concurrency management supporting dynamic model structure.

Director from a library defines component interaction semantics.

Large, behaviorally-polymorphic component library.

Type system for transported data.

Visual editor supporting an abstract syntax.
Approach: Concurrent Composition of Software Components, which are themselves designed with Conventional Languages (Java, C, C++ MATLAB, Python)
The challenge is to synthesize good implementations from the blend!

*This would be a good problem for bored compiler people!*

Our attempts:
- Ptolemy Classic (Buck, Pino, Ha, ... 1990-1997)
- Copernicus (Neuendorffer, 2002-2006)
- Ptolemy II codegen version 1 (2004-2008)
- Ptolemy II codegen version 2 (2009- ??)
Ptolemy Classic Leveraged SDF to Generate Parallel Code

SDF model, parallel schedule, and synthesized parallel code (1990)

It is an interesting (and rich) research problem to minimize interlocks and communication overhead in complex multirate applications.

Lee, Berkeley 11
Ptolemy Classic Provided Cosimulation of Hardware and Generated Software

An SDF model, a “Thor” model of a 2-DSP architecture, a “logic analyzer” trace of the execution of the architecture, and two DSP code debugger windows, one for each processor (1990).

Four DSP 96000 floating point processors interconnected using the "ordered memory architecture," which greatly reduced shared memory synchronization costs [Sriram, 1993]
Second Attempt (Copernicus)

- Steve Neuendorffer created in Ptolemy II a code generator base on the idea of object specialization.

- Java objects would be translated at the byte code level to more specialized Java objects based on their usage in a particular context.

- A tour-de-force, but unmaintainable in our context...
Third attempt: resurrect Ptolemy Classic codegen, but with partial evaluation concepts

model analysis

model (actor-oriented program)

execution context:
data types, buffer sizes, schedules, parameters, model structure, etc.

partial evaluator (code generator)

highly optimized target code blocks

input

monolithic and efficient executable

output

Lee, Berkeley 15

Gang Zhou
Man-Kit Leung
The Code Generation Process

- **Definition**

  \[
  \text{ITERATION} := \text{prefire} \cdot \text{fire}^* \cdot \text{postfire} \\
  \text{EXECUTION} := \text{initialize} \cdot \text{ITERATION}^* \cdot \text{wrapup}
  \]

<table>
<thead>
<tr>
<th>Executable:</th>
<th>Code Generator:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize()</td>
<td>GenerateInitialCode()</td>
</tr>
<tr>
<td>Prefire()</td>
<td>GeneratePrefireCode()</td>
</tr>
<tr>
<td>Fire()</td>
<td>GenerateFireCode()</td>
</tr>
<tr>
<td>Postfire()</td>
<td>GeneratePostfireCode()</td>
</tr>
<tr>
<td>Wrapup()</td>
<td>GenerateWrapupCode()</td>
</tr>
</tbody>
</table>
The Code Generation Process

CompositeActor:

```java
fire() {
    D.fire();
}
```

Director:

```java
fire() {
    order = getSchedule();
    for each A ∈ order
        A.fire();
}
```
Fourth attempt: Build on this, but create a Software Architecture for Experimentation

![Diagram of software architecture]

- **CodeGenGenerator**:
  - `+ generateBodyCode()`
  - `+ generateInitializeCode()`
  - `+ generatePreinitializeCode()`
  - `+ generateSharedCode()`
  - `+ generateWrapupCode()`

- **CodeGenGeneratorAdaptor**: Extends
  - `+ generateFireCode()`
  - `+ generateInitializeCode()`
  - `+ generatePreinitializeCode()`
  - `+ generateSharedCode()`
  - `+ generateWrapupCode()`

- **Director**
  - **AtomicActor**
    - **TypedCompositeActor**
      - **Ptolemnizer**
      - **FFT**

- **ProgramGenerator**
  - **CCCodeGenGenerator**
  - **DEDirector**
  - **Ptolemnizer**
  - **FFT**

Lee, Berkeley 18
Target Hierarchy

$PTII/pтолemy/cg/kernel/generic/

$PTII/pтолemy/cg/kernel/generic/program/

generic

program

procedural

C

Java

VHDL

HTML

...
Sections of the Generated Content:

- IncludingFiles
- SharedCode
- VariableDeclaration
- PreinitizeCode
- InitializeCode
- BodyCode
- WrapupCode
Non-trivial Components

*If we need to generate *complex* code for an atomic component (e.g. FFT) that is highly parameterizable*…
Meta Programming

Our (rather primitive) meta-programming mechanism uses templates:

Var $v1;
Var $v2;
Var $v3;

Function $foo () {
  loop i = 1 to $bound:
    bar(i);
  end loop
}
This mechanism enables integration of C code into actor-oriented models.

The EmbeddedCActor in Ptolemy II wraps low-level functionality (written in C) to define an actor. This approach makes it easy to build actor-oriented models and to generate efficient, platform-specific C implementations.
Example target showing that very low overhead code generation and integration of legacy C code is possible.

The iRobot Create (the platform for the Roomba vacuum cleaner) with a pluggable Command Module has an Atmel 8-bit microcontroller with a very small amount of memory.
This design of a hill-climbing control algorithm wraps code provided by iRobot as demo code into actors in Ptolemy II for accessing sensors and actuators.

Adapter classes provide code generators for generic actors and FSM controllers.

---

```cpp
/** initBlock**/
// Set the sensor data to be all zero.
// This initializes the buffer that gets filled by the interrupt service routine that reads from the serial port.
for(int i = 0; i < Sen6Size; i++) {
    sensors[i] = 0x0;
}
/***/

/** ireBlock**/
if (ref(trigger)) {
    // Request Sensors Packet 2
    byteTx(CmdSensors);
    // Request packet 0, which has 25 bytes of information.
    byteTx(0);

    for(int i = 0; i < SenOSize; i++) {
        sensors[i] = byteRx();
    }
```
Conclusion

- Heterogeneous models
  - Actor models
  - Imperative code

- Code generation
  - Synthesis of practical realization

- A component technology
  - Chunks of imperative logic encapsulated in a concurrent MoC