

A Declarative Executable Model for Object-Based Systems using Functional Decomposition

Pierre Kelsen

March 2006

The Problem

**How should one model
software systems?**

Modeling Goals

- Modeling = Designing (software applications) before coding
- Why model
 - It is easier to fix a bug early at design time than in the actual code
 - It is easier to comprehend a system when considering it at different levels of abstraction and from different viewpoints

UML

- UML is the de-facto standard in modeling large software systems
- Grounded in OO-style programming – central concepts are classes and objects
- Historically based on earlier modeling languages proposed by James Rumbaugh, Grady Booch and Ivar Jacobson (OMT, Use cases, OOSE)

Drawbacks of UML

- UML is large and complex (very much like the systems it wants to model!)
 - Comprises many different concepts
 - Imprecise semantics
- Synchronizing code with models is difficult
 - Using multiple models/diagrams makes it difficult to keep them consistent with each other and the code
 - Much code has to be added by hand

Quote

"Object-oriented programming is an exceptionally bad idea which could only have originated in California."

(E W Dijkstra)

What we propose

- A simple model that represents both static and dynamic aspects of an application (in UML these correspond to separate models)
- A model that is executable
 - Models based on hybrid approach combining graphical elements with code snippets
 - Code generation produces fully functional code

The DEMOS tool

- Developed by Christian Glodt during the last two years (FACTORS/DASCOM projects)
- Implemented as an Eclipse plug-in
- Supports editing and executing our models using background *rule-based code generation*
- Used to illustrate concepts presented in this talk

Motivation

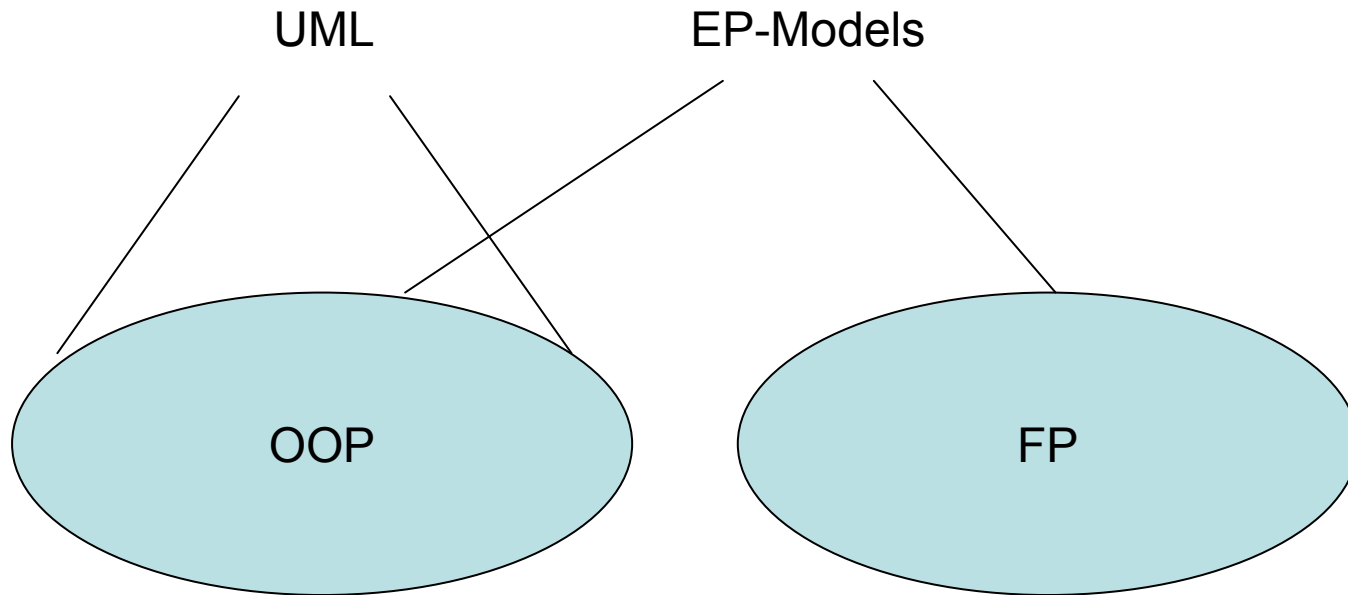
- Declarative model originated in the FACTORS research project
 - Goal: develop mathematical models for studying software complexity
 - Found traditional programming languages “too expressive” to permit a deeper study of this subject
 - need for declarative executable model was identified

EP-Models

- A first look \Rightarrow DEMOS tool

EP-Models versus UML

- One important difference between UML and EP-Models



The Event-Property Model

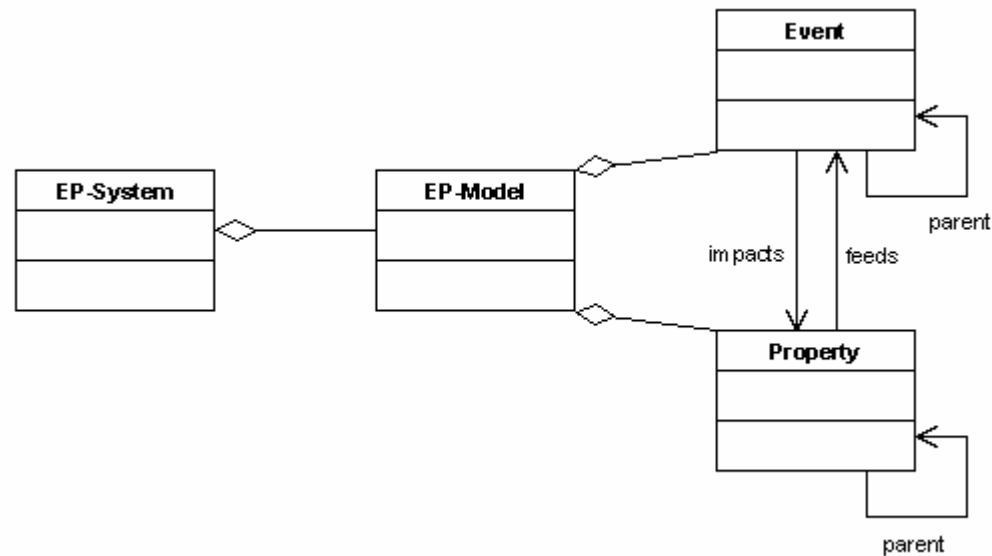
- The next slides will answer the questions
 - What are event property-models?
 - How does one develop applications with these models?

EP-Models: Overview

- An Event-Property system, or **EP-system**, consists of a set of Event-Property-models or **EP-models**.
- EP-models contain two types of entities: **Properties** and **Events**.
- These entities are related using four types of relationships \Rightarrow see metamodel

EP-Models: Metamodel

- A high-level meta-model



- Note: parent relationships not restricted to single model!

EP-Models: Static View

- *Local properties* define the static structure of the state of a model
- Each property has a *type* and a *value*
- Type can be *external* (e.g., Java Type `javax.swing.JButton`) or *internal* (referring to another model)
- Properties can be single-valued or of a collection type
- Example \Rightarrow DEMOS

EP-Models: Static View (2)

- **System state** represents snapshot of EP-system when it executes
- Composed of instances
- Each instance has a value for each of its properties
- **Formally: instance = triplet (M, id, φ) where**
 - M is a model
 - id : unique identifier
 - $\Phi = \text{valuation}$: function that assigns to each local property of M a value of the type of the property

EP-Models: Dynamic View

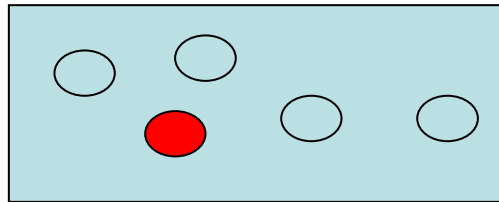
- The system state can change over time by reacting to **events**
- A *local event* represents an external stimulus that triggers change
- A local event has a *type* and a *source*.
- Current version: event types are specific to the Java platform
- Example \Rightarrow DEMOS

The Transformation Mapping

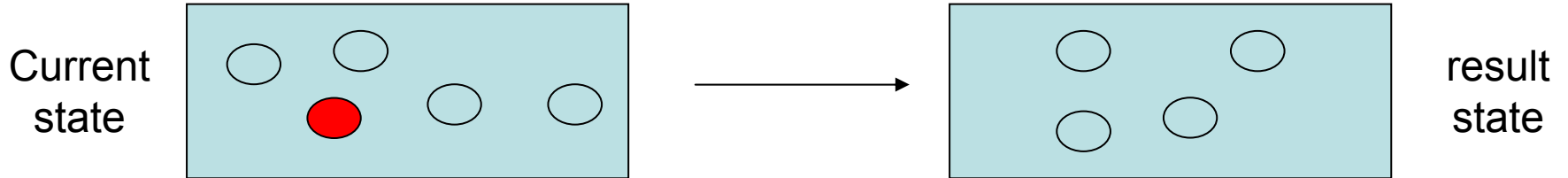
- Mathematically can express the state transformation using a mapping
- Define: instance event = pair (e,x) where e is local event and x is instance
- If e occurs on instance x , then (e,x) is an *active instance event* and x is the *locus* of the event
- Transformation mapping:
(Current state, active instance event)
 \rightarrow *Result State*

Centered States/ Functions

- A **centered state** is a system state together with a distinguished instance, the **center**.



- A **centered function** is a function whose domain is a set of centered states
- Example: transformation mapping for given event



Functional Decomposition

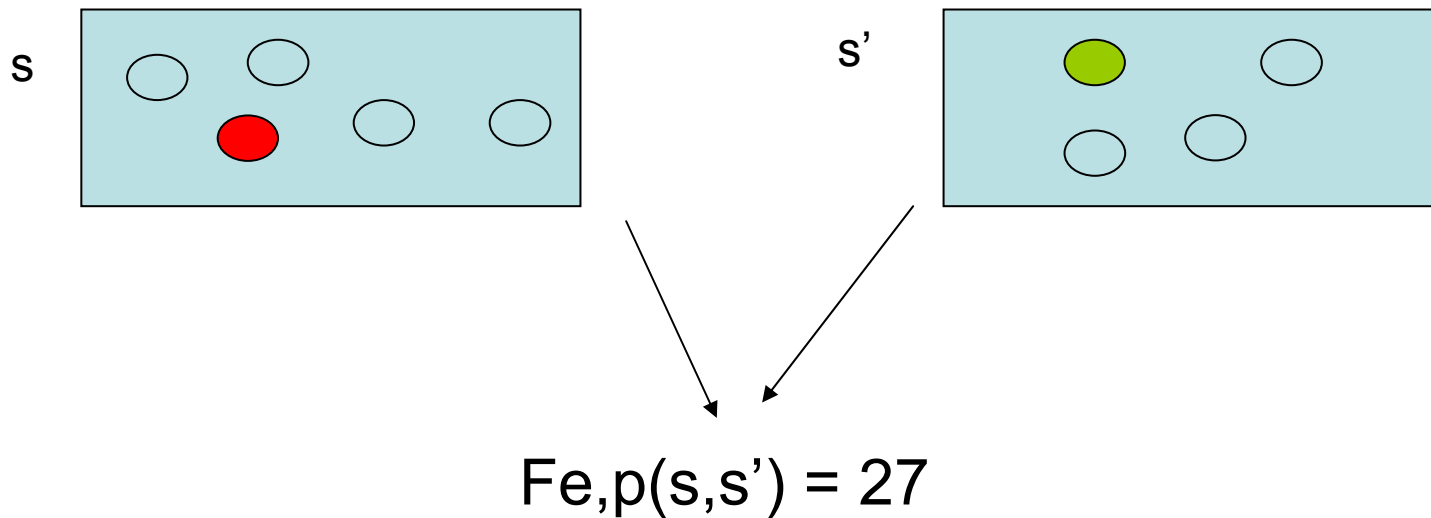
- Main question:
 - How do we represent the transformation mapping?
 - Basic strategy
 - Decompose it into simpler functions
- ⇒ Functional Decomposition

Functional Decomposition (2)

- First step: consider the value of a single property p of some model M
- Let $\mathbf{Fe,p}$ denote the value of local property p after e occurs
- $\mathbf{Fe,p}$ depends on
 - Current state and locus of event
 - Result state and particular instance of M

Bicentered Functions

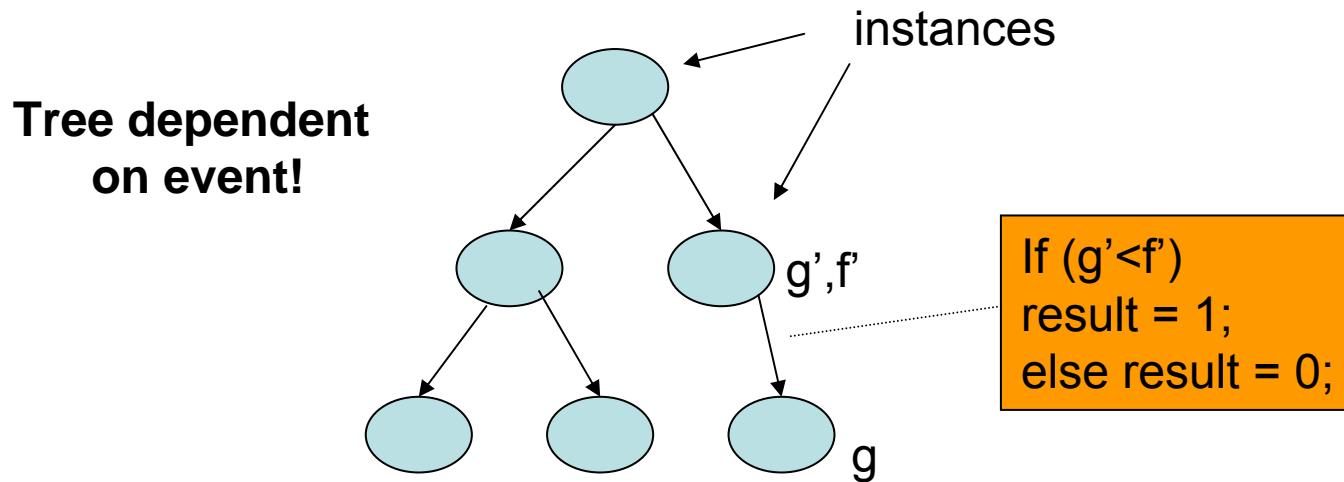
- Can view **Fe,p** as a function of two centered states, a **bicentered function**



Bicentered Functions (2)

- How to decompose a bicentered function
 - Define tree structure *for each event* that spans instances in the result tree
 - Decompose bicentered function at a node into
 - Bicentered functions at parent node
 - Centered functions at parent node

Bicentered Functions (3)



In example g' may be a bicentered function and f' be a centered function.

What is left

- Need to explain three things
 - How to define the tree structure
 - How to represent bicentered functions and their decomposition
 - How to represent centered functions and their decomposition

The Event Graph

- Event graph defined for local event
- Vertices are local events and **remote events**
 - For an event to affect instances of other models add remote events to those models
- Edges are event links, i.e., triplets (e,p,e')
 - Represents event e propagating via a p -link
 - Links can either be *forward* or *inverse*
- Example \Rightarrow DEMOS tool

Representing Centered Functions

- Centered functions are represented by *query properties*
- Just like bicentered functions centered functions are decomposed into simpler functions
- For this define for a property a **property graph**
 - Each node of the property graph is a local or query property
 - Code at each node computes value in terms of children values
- Example \Rightarrow DEMOS tool

Representing Bicentered Functions

- Represent bicentered functions by **parameters** associated with event
- For each parameter and each incoming event link define code that computes value of parameter in terms of centered and bicentered functions at parent event
- Bicentered functions at parent are parameters
- How about centered functions at parent event?

Attaching Centered Functions

- Centered functions are defined in model using query properties
- To access a query property in an event, the query property needs to be *attached* to the event
 - Represented in the model by a *feeds* relation from the property to the event
- Example \Rightarrow DEMOS tool

Function $F_{e,p}$

- Bicentered function that computes the new value of property p when event e occurs
- Not represented by a parameter but by code snippet on event-impacts-property link
- Code can use values of parameters and attached properties of source event
- Example \Rightarrow DEMOS tool

A Sandbox Model for Computation

- Code snippets used at various places in the model
 - On query properties to compute centered function
 - On local properties for initialization (skipped)
 - On event links to compute parameter
 - On event-impacts-property links to compute new value of property

A Sandbox Model for Computation (2)

- Code snippets can only access input values
 - For query properties and local properties children values
 - For parameters the values of parameters and attached properties at source of event link
 - On impact links values of parameters and attached properties at source event
- This sandbox model is fully implemented in the DEMOS tool

A Sandbox Model for Computation (3)

- Advantage of sandbox model
 - Greatly reduces coupling compared to traditional OO-systems
- No support in UML for such a facility

The Curse of Sequentiality

- Another significant advantage of our model is that sequentiality is greatly reduced
 - Order of event execution immaterial given some simple restrictions on event graph
- Traditional OO systems owe much of their complexity to the overspecification of sequentiality
 - Encouraged by UML sequence and collaboration diagrams

Other topics

- A simple component model for EP-models
 - Adapter classes and adapter models to integrate existing Java classes into EP-systems
- ⇒ DEMOS tool

Conclusions

- EP-models present a **declarative model** of software systems
- Use OO-ideas for representing **state** but functional principles for representing **behavior**
- Offer significant advantages of traditional UML-based models
 - They are executable!
 - Reduced coupling via sandbox models
 - Reduced complexity by minimizing sequentiality
 - Uses small number of intuitive concepts

Questions

- Can we use EP-models effectively for modeling larger software system?
(i.e., “Are they ready for Prime Time???)
- Can similar declarative models be used to model other aspects of applications ?
→ DASCOM project
- Do EP-systems provide a suitable setting for studying software complexity?

DEMOS-Tool: Beta Testing

- Wanted:
Beta testers

