

Developing Dependable Systems Using Aspect-Oriented Modeling Techniques: Promises & Challenges

Robert B. France
Dept. of Computer Science
Colorado State University
france@cs.colostate.edu



Outline of talk

- On the difficulty of developing dependable software
- An Overview of Aspect-Oriented Modeling (AOM)
- AOM Challenges
- Conclusion



Developers of mission-critical open distributed software systems need to balance multiple, interdependent design concerns such as **availability, performance, survivability, fault tolerance, and security**.

- A concern can be a set of related requirements or a set of related design objectives.



Balancing dependability concerns

- Balancing requires making trade-offs and assessing risks associated with design features that address the concerns
 - Organizations seldom have all the resources needed to build software systems that have the desired levels of dependability
 - Need to consider and evaluate alternative features to determine the extent they
 - address concerns
 - cost-effectively mitigate product-related risks.
 - Pervasiveness of dependability features complicates their evaluation and evolution



The Problem-Implementation Gap



- A problem-implementation gap exists when implementation and problem abstraction levels differ
 - when the gap is wide significant effort is required to implement solutions
- Complexity arises as a result of effort needed to bridge wide problem-implementation gaps
 - Bridging the gap using manual techniques introduces accidental complexities in these cases

"I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representations"



Key Software Development Principles

- Separation of concerns
 - Abstraction
 - Separation of views
- Rigour and Formality
 - Supports development of analysis tools
 - Reducing accidental complexities through automation



Why consider modeling techniques?

Software development is a modeling activity!

Programmers build and evolve mental models of problems and solutions as they develop code

Programmers express solutions in terms of abstractions provided by a programming language

How can we better leverage modeling techniques?

Colorado State University
Computer Science Department

Going beyond traditional support for separation of concerns

- The Separation of Concerns principle is considered fundamental in software engineering
- Much attention paid to providing support for modularizing descriptions of problems and solutions (separation of parts)
- Less attention has been paid to providing support for understanding interactions across separated parts

Colorado State University
Computer Science Department

On the importance of understanding interactions across features: An example

The first launch of the space shuttle Columbia was delayed because "(b)ackup flight software failed to synchronize with primary avionics software system"

(<http://science.ksc.nasa.gov/shuttle/missions/sts-1/mission-sts-1.html>)

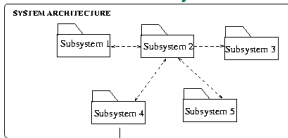
Colorado State University
Computer Science Department

Balancing dependability concerns

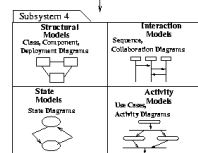
- Balancing requires making trade-offs and assessing risks associated with design features that address the concerns
 - Organizations seldom have all the resources needed to build software systems that have the desired levels of dependability
 - Need to consider and evaluate alternative features to determine the extent they
 - address concerns
 - cost-effectively mitigate product-related risks.
 - Pervasiveness of dependability features complicates their evaluation and evolution

Colorado State University
Computer Science Department

Models as System Views



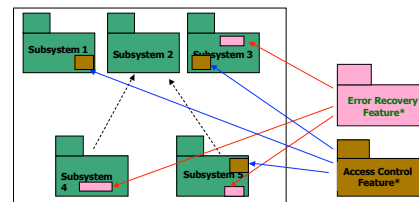
Language-defined views



- UML models present different views of systems
- Evolution of system effected by evolving models (views)
 - Requires well defined relationships between models
 - requires well defined notions of realization/refinement/abstraction (e.g., see Catalysis Method)

Colorado State University
Computer Science Department

Crosscutting Design Views

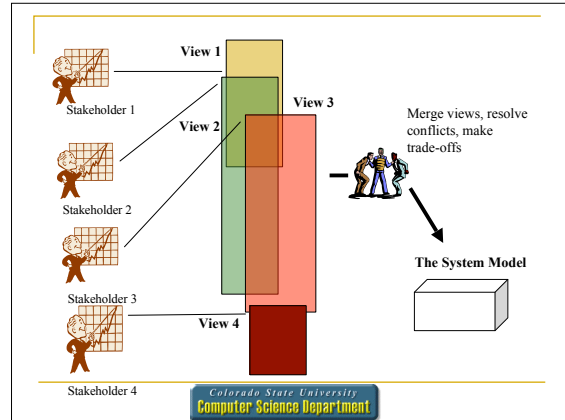


*A feature is a logical unit of behavior

Colorado State University
Computer Science Department

The problem with cross-cutting features

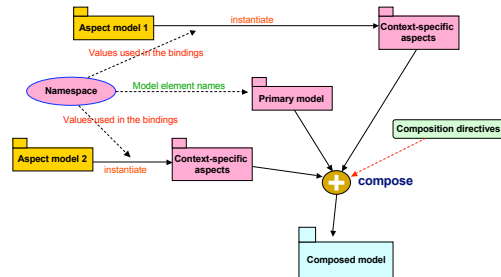
- ... understanding and changing them!
 - Information is distributed
 - Maintaining consistency in the presence of changes is problematic
 - Difficult to consider alternative treatments
- Lack of attention to balancing dependability concerns early in the development cycle can lead to major re-architecting in later stages of development



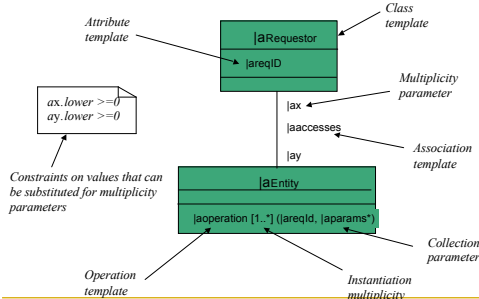
Aspect-Oriented Design Models

- Separation of Concerns
 - Primary Model*: A model of core functionality; determines dominant structure
 - Aspect Model*: Describes a feature that cross-cuts modules in the dominant design structure
- Aspects as Design Patterns
 - Isolate crosscutting features by capturing their structural and behavioral pattern

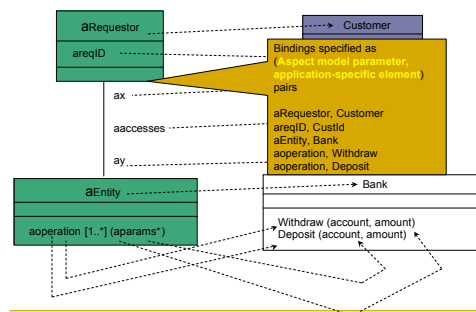
Aspect Oriented Modeling

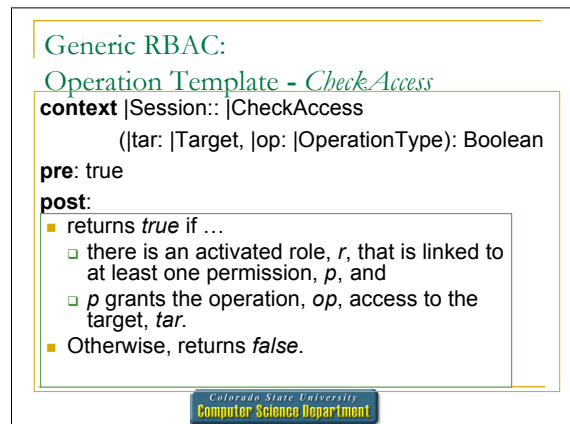
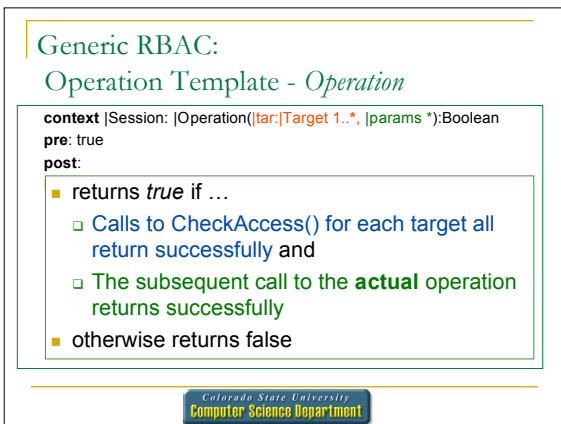
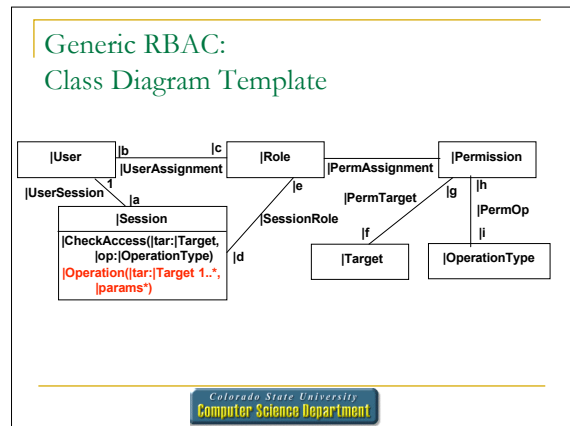
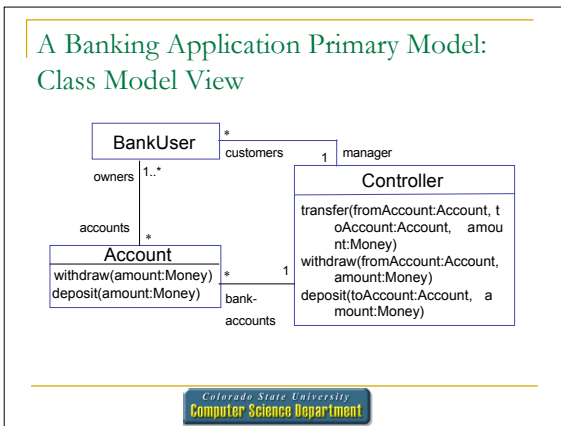
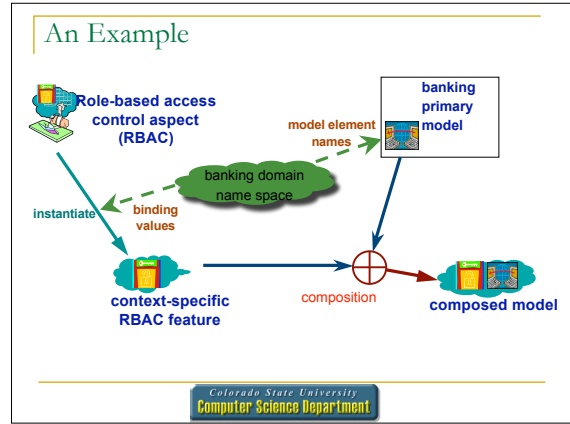
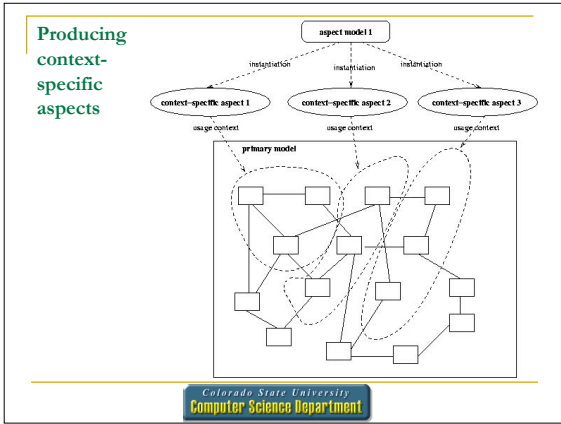


A simple aspect: a client-server pattern

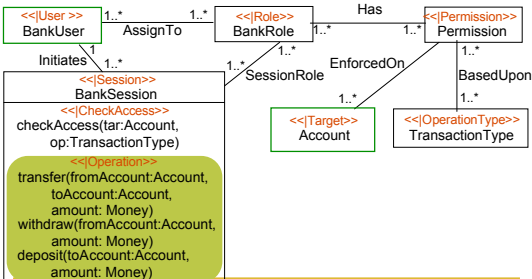


Producing a context-specific aspect

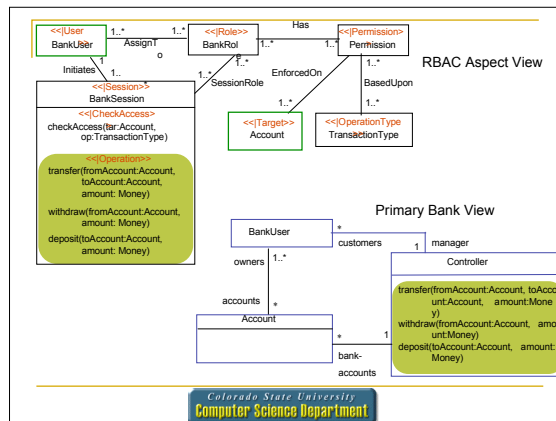




A Context-specific RBAC Class Diagram



Colorado State University
Computer Science Department



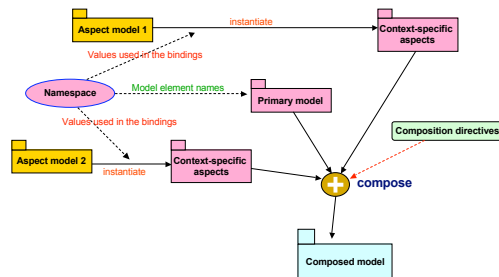
Colorado State University
Computer Science Department

Instantiation Challenges

- How can one automatically identify candidate locations for incorporating aspects?
 - AspectJ uses names with wildcards
 - Need a better way of characterizing locations
- Possible solutions
 - Characteristics of model elements (e.g., criticality, business value, sensitivity) captured in metadata
 - Use presence of behavioral and structural patterns in primary models to identify candidate locations in primary models

Colorado State University
Computer Science Department

Model Composition



Colorado State University
Computer Science Department

Model Composition

- Basic composition procedure
 - Elements with matching syntactic properties are assumed to represent the same semantic concepts
 - An element's **signature** consists of the syntactic properties that determine matches
 - Matched elements are merged according to default merging rules

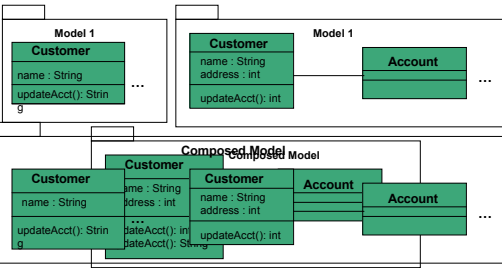
Colorado State University
Computer Science Department

Signature-based Composition

- Signature:** A set of property values
 - Properties:** Properties (e.g., attributes, association ends) are associated with the class of the element in the UML meta-model
 - Signature Type:** A particular set of properties that determine the signature
 - Default signature type:** Used when signature type is not specified explicitly
 - Complete signature type:** Consists of all the elements properties
- Example:**
 Signature type: {operation (name, {parameter (name, type)}) }
 Operation example: update(x:int, y:int):int
 Operation signature: {update, { (x, int) (y, int) } }

Colorado State University
Computer Science Department

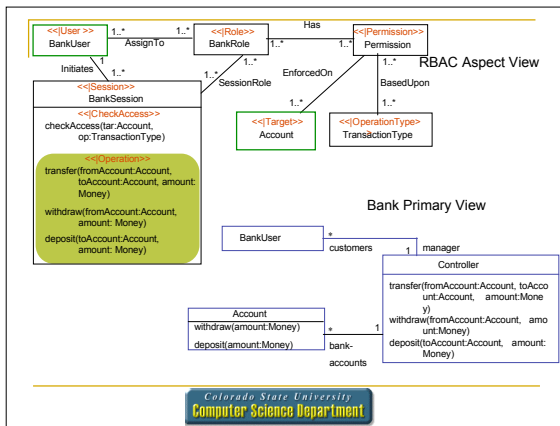
Signature-based composition



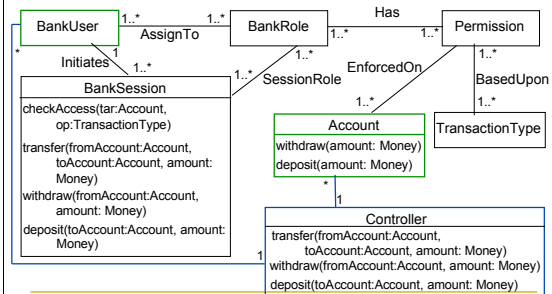
Merging using a signature type consisting of class name, attributes, operations and association ends
 - merging using a class signature consisting of class name and complete signature
 - transfer for attributes, operations and association ends
 - isAbstract and ownedAttributes.

Default composition rules

- If model elements match with respect to the signatures, they are merged
 - Constituent elements are recursively merged using signatures of those elements
 - If a model element property in one matching element is not in the other, then it appears in the composed model.
- If model elements don't match they appear in the composed model
- If the matching model elements have pre and post conditions,
 - Merge of preconditions is a disjunction
 - Merge of postconditions is a conjunction.
- If the matching model elements have constraints (class invariants), the constraint associated with the element in the composed model is the conjunction of the constraints.
- If the matching elements are associations, then the *stronger* (more restrictive) multiplicity at an association end is used.

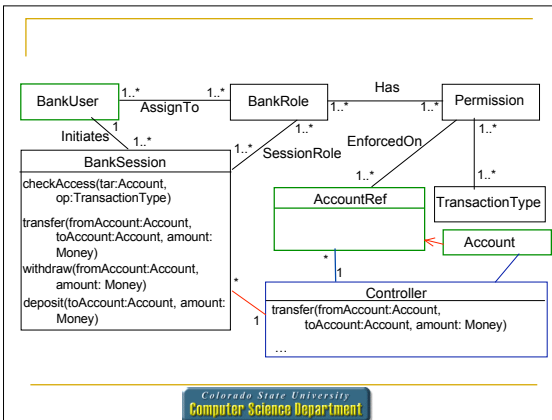


Composed Class Model



Problems with basic composed model

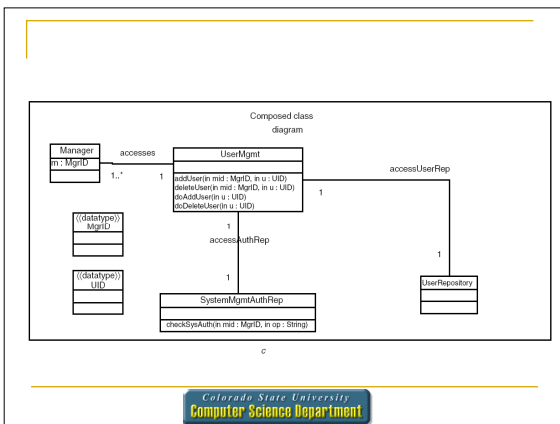
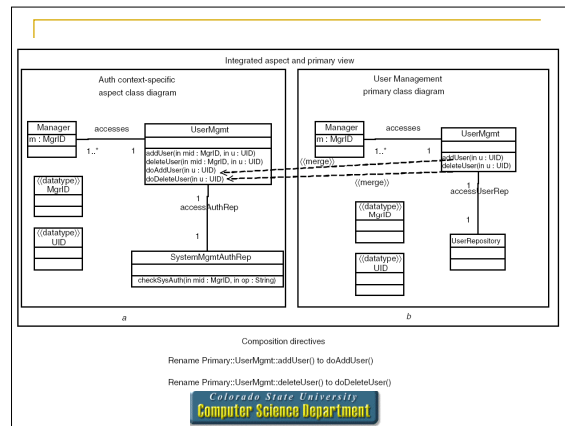
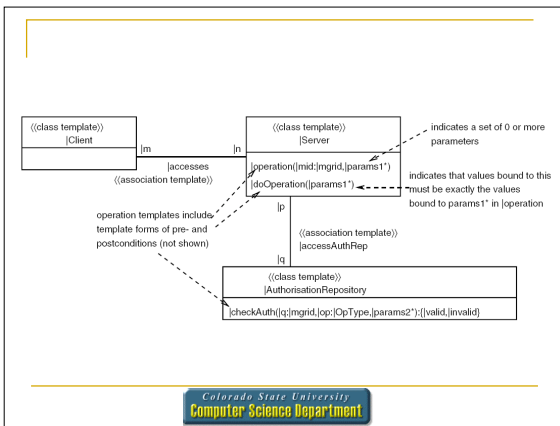
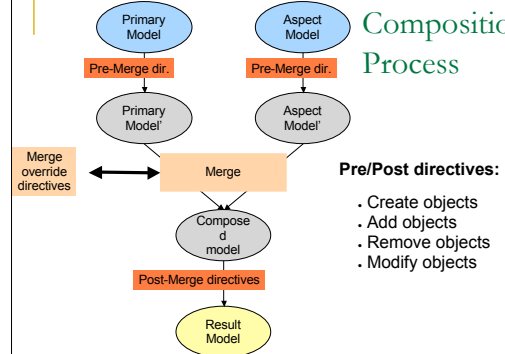
- Bank user can bypass session and call controller operations directly
 - Need to delete association between BankUser and Controller
- Bank session operations need to access operations in controller
 - Need to add association between BankSession and Controller
- Account in permission should really be a reference to an account
- **Composition directives** can be used to fix above problems



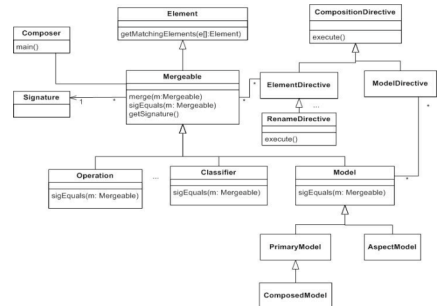
Composition Directives

- Used to extend and modify basic composition procedure
- Two types
 - Model directives:** determine the order in which aspect models are merged
 - Element directives:** affect how model elements are composed

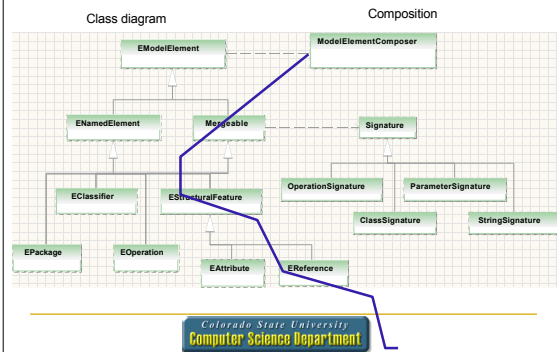
Composition Process



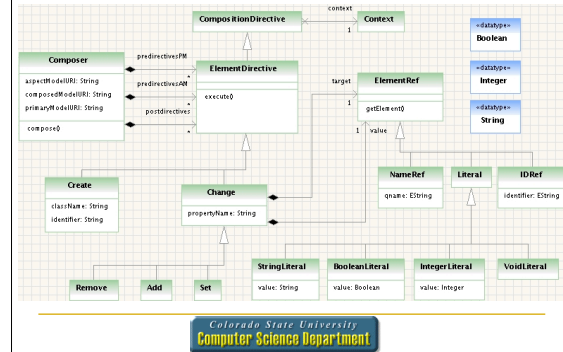
Composition Metamodel



Composition Meta-model (merge)



Composition Meta-model (directives)



Element References

- Name reference
 - `MyPackage::MyClass::MyOperation`
- ID reference
 - `$myObject`
- Literals
 - String: "a string"
 - Integer: 12
 - Boolean: true/false
 - Void: void

Create directive

- Description
 - Creates a new object and gives it an ID
- Examples
 - `create Class as $c`
 - `create Operation as $op`

Change directives

- Description :
 - Changes the value of a property of an object
- Set (property multiplicity must be 1)
 - `ObjectRef.PropertyName = ValueRef`
- Add (property multiplicity > 1)
 - `ObjectRef.PropertyName + ValueRef`
- Remove (property multiplicity > 1)
 - `ObjectRef.PropertyName - ValueRef`

Composition Challenges

- Verifying presence and absence of properties in composed models
- Tackling architectural mismatches
 - Different structures used to represent the same concept
 - Concepts described at different levels of abstraction across aspect and primary models
- Improving matching criteria
 - Reducing misses and false identifications

Conclusion

