Transforming and Validating Models: Experiences & Challenges

Robert B. France
Colorado State University

2006 Summer School on Model-Driven Engineering

Why MDE?: To Cope with Complexity

- Software development techniques have not kept pace with the growing complexity of software systems
- Untrained persons can hack something together that apparently works

Building software pyramids

Building complex software with current tools is akin to building pyramids in ancient Egypt
Or cathedrals in medieval times…

Outline

- Part 1: Model Transformation
  - Model Transformation & Validation Introduction
  - Query, View Transformation Overview
  - Specifying & Implementing Transformations: 2 Approaches
- Part 2: Testing Models using UMLAnT (UML Animation and Testing Tool)

Using the UML

"UML as a sketching language" vs. "UML as a basis for MDE"

- The "UML as a basis for MDE" challenges
  - If UML models are to be used as the basis for generating implementations then techniques for transforming, validating and managing models must be available.
  - UML-based MDD techniques must cope with multiple, overlapping views of a problem or solution

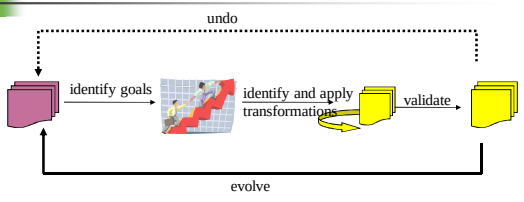
5

Are transformations the heart of MDE?

- MDE = models + transformations
- Transformation-driven development?
  - Specify models
  - Evolve models
    - Horizontal transformations (e.g., refactor to change features or enhance design quality, transform descriptive models to analysis models)
    - Vertical transformations (e.g., realize design in code, refine design features)

6

Process Structure Model



7

Actions & Possible Cycle Goals

- **Refinement/Realization**: e.g., create design models from analysis models.
- **Abstraction**: e.g., gain understanding of an existing artifact or model by abstracting out irrelevant details.
- **Inferencing**: e.g., transforming a model to make implied properties explicit.
- **Analyzing**: e.g., transforming a model to make it amenable to particular forms of analysis.
- **Refactoring**: e.g., transforming a model to improve its evolvability.

8

Vertical Transformations

- Refinement/Realization
  - Realization example: transforming detailed design models to code
  - Refinement example: transforming abstract designs to less abstract designs
- Abstraction
  - Example: reverse-engineering of code to design

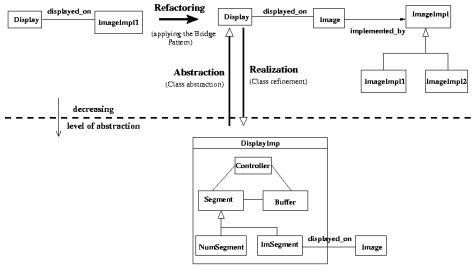
9

Horizontal Transformations

- Refactoring: Improving model quality
  - E.g., applying patterns to designs
    - Requires precise representation of patterns
- Inferencing: Inferring properties from models
- Analyzing: Extracting information needed for analysis

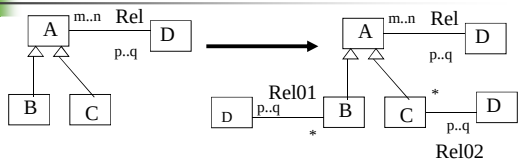
10

Refactoring and Realization Example



11

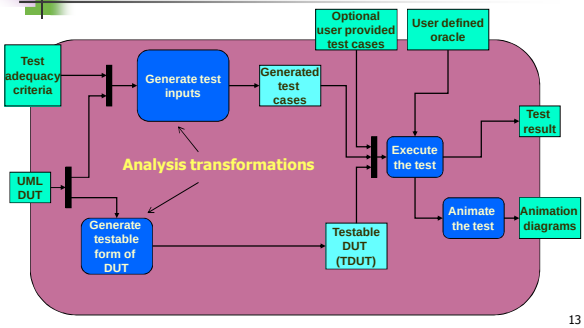
Simple Model Inferencing Example (work of A. Evans et al.)



$Rel01 \cup Rel02 \subseteq Rel$

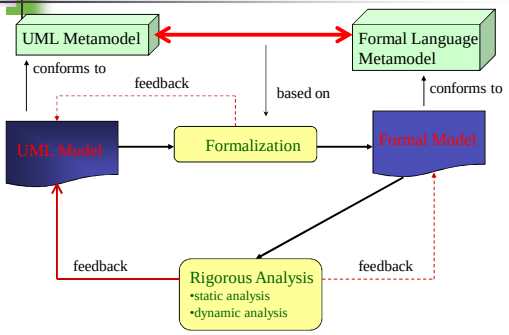
12

Transforming to an analysis model: An Example



13

Integrated Methods



14

Query, View, Transformation Standard

… only because someone will ask the question "What about the QVT?"

Query View Transformation Standard

- Standard language-based concepts for specifying and implementing transformations
- Based on best available experience (!)
  - and other concerns …
- Supports hybrid declarative, imperative styles

16

QVT Structure

- Declarative section
  - transformations specified as relationships among modeling structures (MOF models)
  - Provides support for specifying patterns (as object template expressions) and for pattern matching
- Imperative section
  - Provides a standard language for describing implementations of relations as Mapping Operations
    - A transformation described entirely by Mapping Operations is called an operational transformation
  - Provides support for using non-standard implementation operators for realizing relations

QVT Overview

From the QVT "final" standard

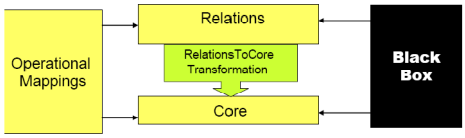


Figure 6.1 - Relationships between QVT metamodels

Specifying Relations

```
relation PackageToSchema /* map each package to a schema */
{
    checkonly domain uml p:Package {name=pn}
    enforce domain rdbms s:Schema {name=pn}
}

relation ClassToTable /* map each persistent class to a table */
{
    domain uml c:Class {
            namespace = p:Package {},
            kind='Persistent',
            name=cn }
    domain rdbms t:Table {
            schema = s:Schema {},
            name=cn,
            column = cl:Column { name=cn+'_tid', type='NUMBER'},
            primaryKey = k:PrimaryKey { name=cn+'_pk', column=cl} }

    when { PackageToSchema(p, s); }

    where { AttributeToColumn(c, t); } }
```


Graphical Representation

```
relation UML2Rel
{
        checkonly domain uml1 c:Class {
                name = n, attribute = a:Attribute{name = an}}
        checkonly domain r1 t:Table {
                name = n, column = col:Column{name = an}}
}
```

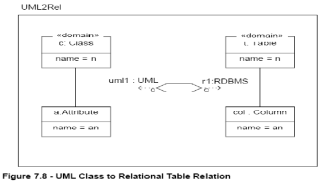
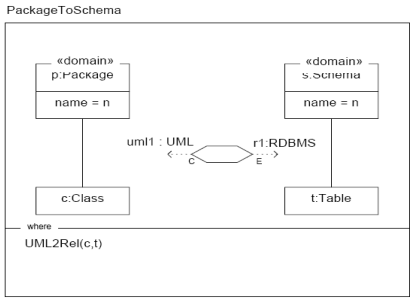


Figure 7.8 - UML Class to Relational Table Relation

Graphical Representation



More Simple Examples (from the standard)

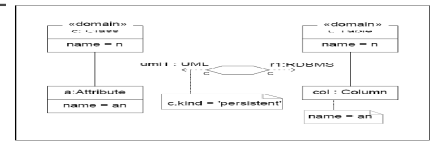


Figure 7.11 - UML2REL with constraints

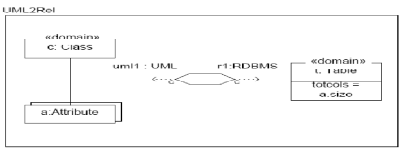
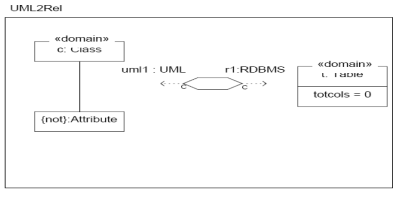


Figure 7.12 - Example using a Set

The {not} construct

```
relation UML2Rel
{
        checkonly domain uml1 c:Class
          { attribute = Set(Attribute){}}{attribute->size() = 0}
        checkonly domain r1 t:Table {totcols = 0 }
}
```



gure 7.13 - Example using {not}

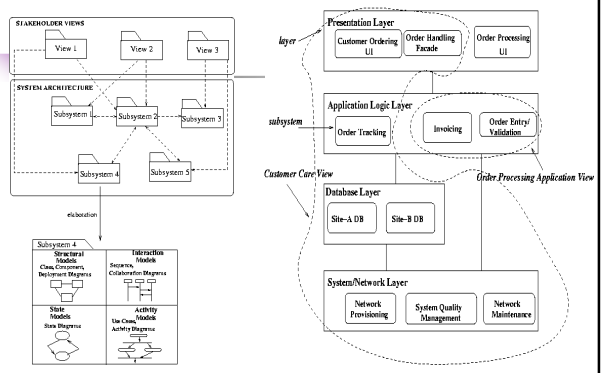
Transformation Challenges

Transforming System 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 refinement/abstraction

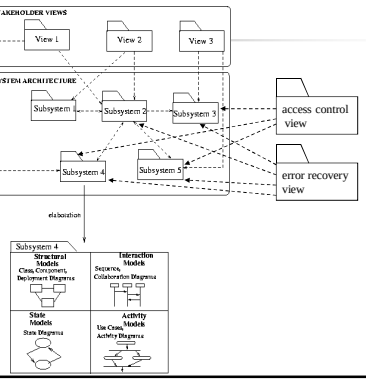
25

System Views



26

Crosscutting Views



27

The problem with multiple views

- … it complicates model transformation …
- Is this complexity essential or accidental?

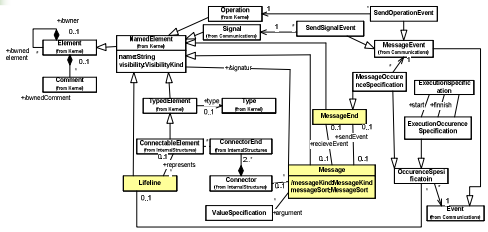
28

Specifying Transformations

- Transformations typically specified in terms of relationships among metamodel elements
  - Transformation actions that are specified: navigate, filter, modify
- Only a subset of a language's metamodel is involved in a particular transformation
- Large, complex metamodels (e.g., the UML metamodel) pose special challenges

29

Navigating the "MetaMuddle"



Can you derive the relationship between a message end and a lifeline?

30

Testing Transformations

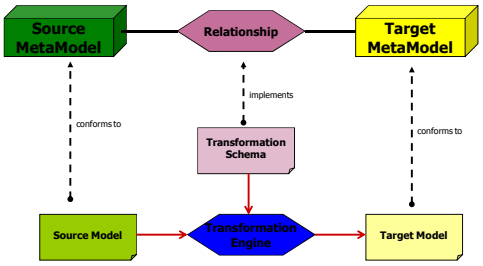
- How does one validate or test a transformation?
  - What are the test criteria?
  - How does one categorize the input space to support coverage analysis?
  - How can the oracle be defined?

31

Specifying and Implementing Transformations

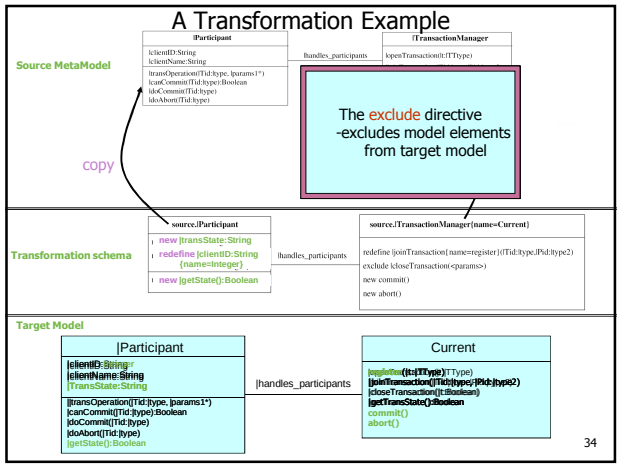
Experiences

An Imperative Approach to Model Transformation



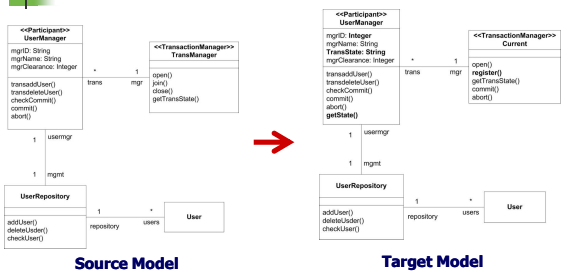
33

A Transformation Example



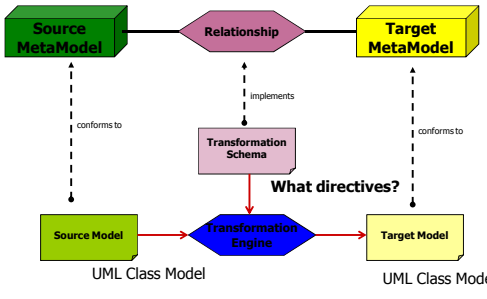
34

Transaction Processing Transformation



35

An Imperative Approach to Model Transformation



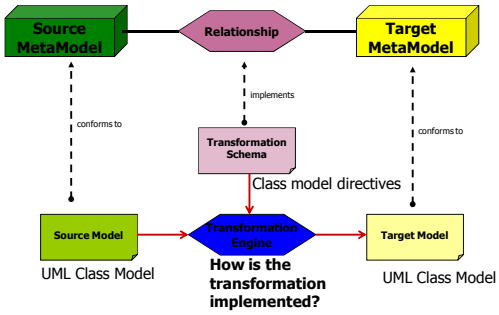
36

Transformation Directives

- The source directive
  - Copy model elements
- The name directive
  - Supply platform-specific name
- The redefine directive
  - Modify copied elements
- The new directive
  - Create new model elements
- The exclude directive
  - Eliminate model elements

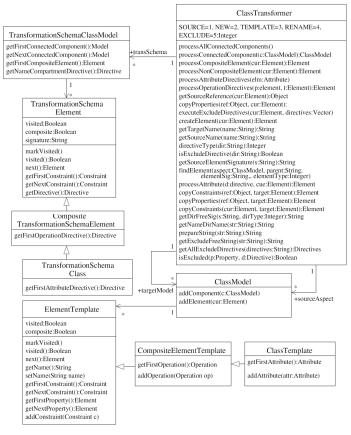
37

An Imperative Approach to Model Transformation



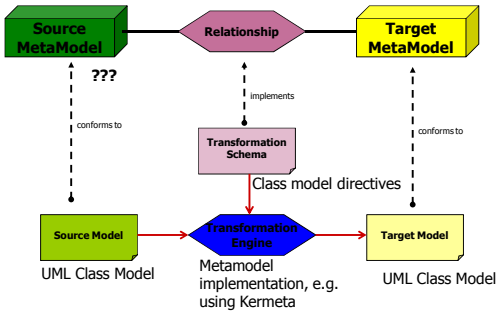
38

Transformation Implementation metamodel



39

An Imperative Approach to Model Transformation



40

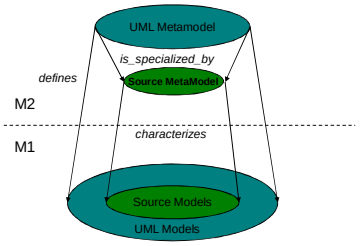
Describing Source MetaModels Using the Role Based Modeling Language

- RBML Features
  - Domain-specific concepts described by roles
  - An RBML role is a specialization of a metamodel class
    - Instances of specialized class represents forms of a domain-specific concept
  - RBML is a variant of the UML that supports descriptions of families of models

41

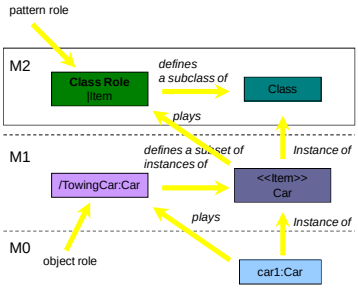
Source Metamodel vs. UML Metamodel

- A source metamodel characterizes a subset of UML models.



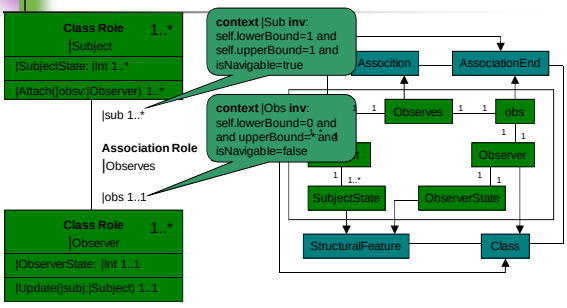
42

Pattern Role



43

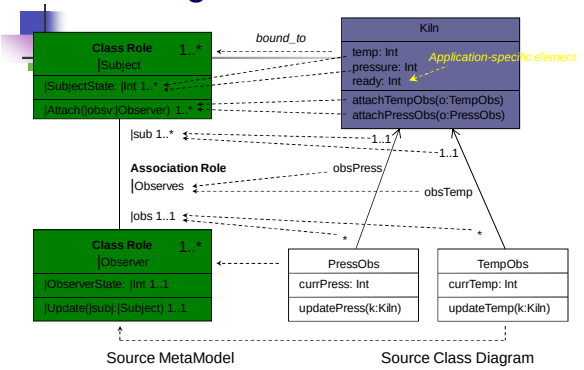
Observer MetaModel



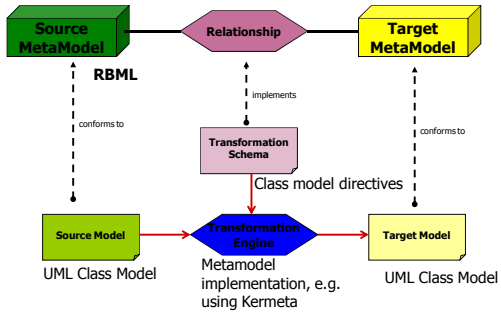
Observer Source MetaModel Metamodel View

44

Conforming Model



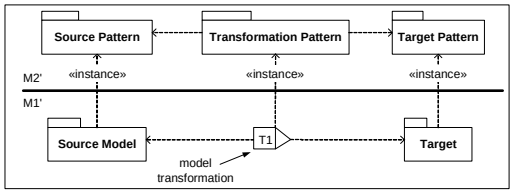
An Imperative Approach to Model Transformation



Specifying Pattern-Based Refactoring

- Worked carried out by Sheena Judson-Miller as part of PhD work
- Focus on specifying transformations that incorporate design patterns into UML design models

Logical View

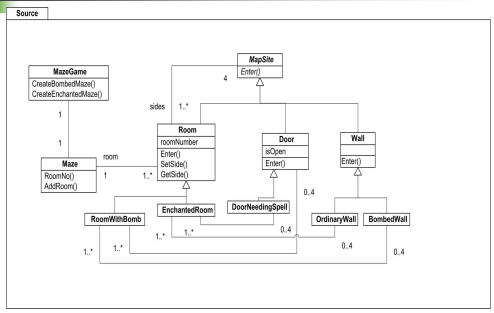


Transformation Pattern

- A transformation is specified in three parts
  - Source pattern: specifies the structural pattern that is the target of the transformation
  - Transformation schema: identifies the source model elements that are removed and the new elements that are added by the transformation.
  - Transformation constraint: specifies the relationships that must exist between source and target model elements

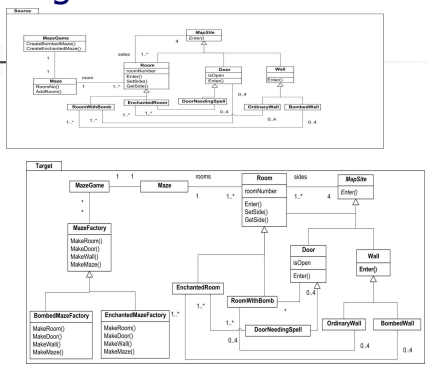
49

An Example: Incorporating the Abstract factory Pattern



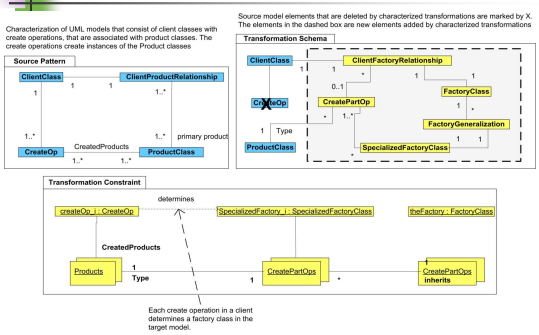
50

Target Model



51

Abstract Factory Transformation Pattern



52

Lessons Learned – Model Transformations

- QVT is not based on significant experience
- Need significant transformation-related experience to move forward in the areas of
  - Specifying transformations
  - Validating transformations
- Repository containing samples and benchmarks are needed to support comparative analyses of techniques and languages
- Specifying, implementing, and validating transformations are currently onerous tasks (suggesting current techniques have an undesirable level of accidental complexity) – they should not be
  - We need to remove the (apparent) accidental complexities associated with the above tasks
  - MDE is in trouble if these turn out to be inherent complexities!

53

Testing UML Models

Approaches for Assuring Validity of UML Design Models

- Inspections and reviews
  - Can be tedious and cumbersome when models are large
- Formal techniques for verifying properties
  - Few can cope with multiple UML views
  - Complementary use of formal and testing techniques can be beneficial

55

Animating and Testing UML Models

- Visualization tool
  - Behavior is animated using object and sequence diagrams
  - Provides quick visual feedback on modeled behavior
- Testing tool
  - Generates test inputs and executes tests on design models
  - Test adequacy determined by UML-based test criteria

56

Testing UML Models: Concern 1

- What information should be in a UML design model to facilitate testing (or what makes a model "testable")?
  - What modeling views are needed?
  - What roles do the modeling views play during testing?
  - How should behavior be described?

What modeling views are needed and what roles do they play?

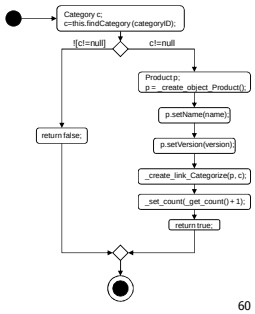
- Class diagram
  - Structural view – classes, associations, attributes
  - OCL constraints – invariants, pre- and post-conditions
  - Used for test case generation and test execution
- Sequence diagrams
  - Used to describe how objects interact to accomplish a specific task
  - Used for test case generation
  - NOT used for test execution
- Activity diagrams
  - Used to describe method bodies

How should behavior be described?

- Action Semantics Proposed in the UML standard
  - No surface language
- The Java-like action language (JAL)
  - Developed at CSU to support specification of method bodies
  - Constructs based only on UML concepts
  - Syntax is close to Java

A JAL Example

```
boolean addProduct(int categoryID, int
    version, String name)
{
    Category c;
    c = this.findCategory (categoryID);
    if(c != null) {
            Product p;
            p = _create_object_Product();
            p.setName(name);
            p.setVersion(version);
            _create_link_Categorize(p, c);
            return true;
    }
    else
            return false;
}
```



Testing UML Design Models: Concern 2

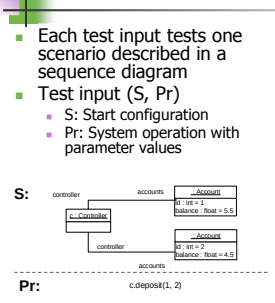
- How can a UML design model be tested?
  - What is a test input?
  - What criteria can be used to determine test adequacy?
  - How can the models be exercised?

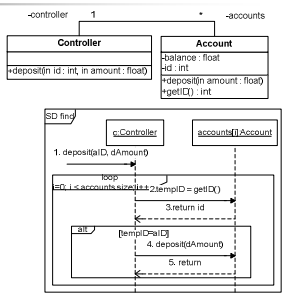
What is a test input?

- Test input: T = <S, Op(p1,p2,…)>
  - S: a start object configuration
  - Op(p1,p2, …) : a system operation with parameter values
- Object interactions that take place as a result of invoking a system operation are described in a single sequence model.

Test input

- Each test input tests one scenario described in a sequence diagram
- Test input (S, Pr)
  - S: Start configuration
  - Pr: System operation with parameter values

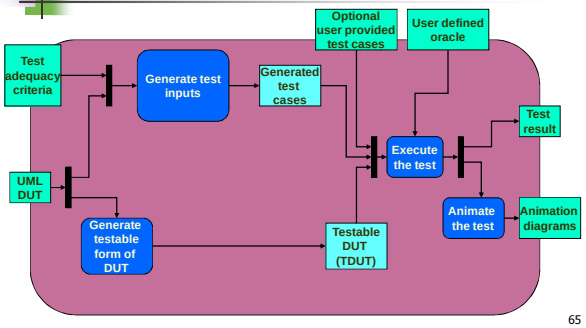




What criteria can be used to determine test adequacy?

- Sequence diagram criteria
  - **Each message on links** : Ensure that every message is executed
  - **Condition**: Ensure every condition evaluates to TRUE/FALSE during some test execution
  - **All message path**: Ensure that every path is executed

How can models be exercised?



Test Failures Reported by UMLAnT

- Initialization errors
  - Uninitialized variables in conditions
  - Uninitialized parameters passed in operation calls
  - Non-existent target object of an operation call
- Pre and Post-conditions that do not hold
- Invalid object configurations produced during tests

Testing UML Design Models: Concern 3

- To what extent can test inputs be generated?

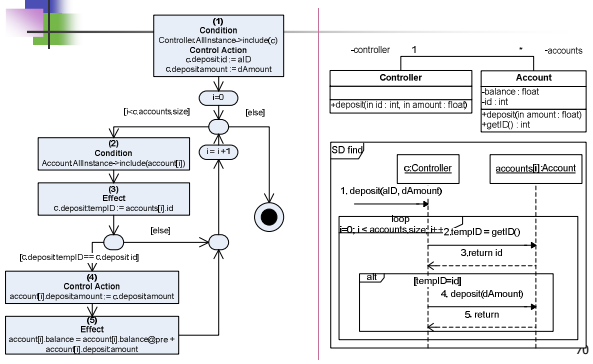
Generating Test Inputs

- Create a graph in which each path through the graph is a representation of a scenario described in a sequence diagram
  - Nodes contain information about the conditions that must be satisfied to reach the node in the graph
- Generate path constraints from the graph
- Form the conjunction of path constraints and class diagram invariants and solve the resulting constraint
  - The solution (if one can be obtained) is a test input

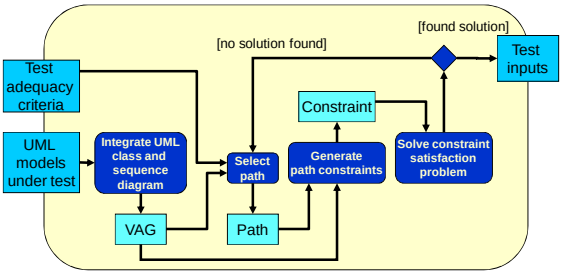
Variable Assignment Graph (VAG)

- VAG – Contains information used to determine inputs that can be used to satisfy sequence diagram criteria
  - Need information that can be used to determine inputs that cover sequence model "paths"
    - Sequence diagram: provides information on sender and receivers of messages, message sequencing, parameter values, return values
    - Class diagram: provides pre and post-conditions for operations associated with incoming messages; information on relationships between classes

Example of a VAG



Test input generation process



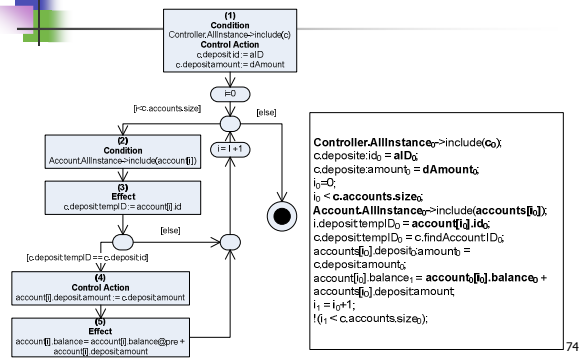
Selecting paths

- Paths are selected based on SD criteria.
  - *Each message on link* → Paths that traverse all VAG nodes.
  - *Condition* → Paths that traverse all VAG edges
  - *All message path* → All VAG paths
- Paths in VAG can be selected using existing techniques

Generate path constraints

- Identify test inputs:
  - System operation parameter values
  - Set of variables that define the start configuration
    - Class instances, attribute values, and association instances that are used before being defined.
- Transform the path into SSA form

Path constraint for the path 1-2-3-4-5



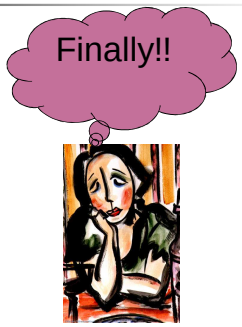
Solving path constraints

- Alloy constraint solver
  - Alloy: first-order relational constraints solver
  - Supports only Integer and Boolean primitive types
- Requires
  - transforming class diagram and path constraints to constraints in Alloy,
  - solving the constraints,
  - transforming Alloy solution to design inputs

Lessons Learned

- UMLAnT can execute, test, and animate UML designs
- Integrates widely used tools, languages – Eclipse, EMF, JUnit, Java, UML

Conclusion



77