Model Driven Architecture
Action Semantics and Action Languages

Prof. Dr. Peter Thiemann

Universität Freiburg

28.06.2006
OMG sanctioned approach to define the low-level behavior of modeling elements

Action semantics **defines how** to perform a transformation of the object graph (side effects)

- a state transition
- an operation

State chart models

- provide a higher-level view
- formalize object lifecycle
- orchestrate method invocations

contrast with:

OCL **specifies what** happens (no side effects)

- the result of an operation
- postcondition after an operation
An action is the fundamental unit of behavior specification.
An action takes a set of inputs and converts them into a set of outputs [...].
The most basic action provides for implementation-dependent semantics, [...].
[...] primitive actions are defined [so] as to enable the maximum range of mappings.
[...] they either carry out a computation or access object memory
This approach enables clean mappings to a physical model, [...].
In addition, any re-organization of the data structure will leave the specification of the computation unaffected.
Action Semantics/Why have it?

- build complete and precise models
- formal proofs of correctness of a problem specification
- high-fidelity model-based simulation and verification
- enables reuse of domain models
- stronger basis for model design and eventual coding
- support code generation to multiple software platforms.

From “Software-platform-independent, Precise Action Specifications for UML”, UML'99
Basic idea: specify computation so that it is
- data driven and
- inherently parallel
- (sequential execution through data dependency or explicit control dependency)
- independent of concrete syntax
Basic building blocks:

- **Pins**: input and output ports of an action; with type and multiplicity
- **Variables**: intermediate results
- **Data flow**: connects the output pin of one action to the input pin of another
- **Control flow**: explicit ordering constraint for action pairs
- **Actions**: for object manipulation, memory operations, arithmetic, message passing, etc.
- **Procedures**: packaging of actions with input and output pins
Life-cycle of an action

- **Waiting.** Initial state after creation of action execution.
- **Ready.** Action execution with all inputs available and all control dependencies in state **Complete**.
- **Executing.** Compute outputs from inputs.
- **Complete.** Values of output pins determined, signal to control-flow dependent actions.
Computation actions *e.g.* mathematical functions (left undefined by standard)

Composite actions building blocks for control structures like loops and conditionals

Read and write actions access, navigate, and modify model-level constructs (objects, links, attribute slots, and variables)

Collection actions $\Rightarrow$ iterators for actions
From: UML Action Semantics for Model Transformation Systems, Varró and Pataricza (uses obsolete 1.5 metamodel)
Action Semantics/Object Actions
Action Semantics can describe object graph transformations

Current support by tools rather poor (wrt editing, code generation)

Too low-level for actual programming (machine independent intermediate code)

⇒ higher-level language required to define the meaning of operations

⇒ Action Languages
Action Languages

“Executable UML”
(Mellor and Balcer, Addison Welsey, 2002)

- Programming Languages geared towards specifying detailed operational behavior
  - Specify algorithmic aspects
  - Abstract from implementation choices/design decisions

- Operate directly on UML data model

- Independence of the SW platform
  - no concrete representation
  - no pointer manipulation
  - no tricks
Types
- All modeling elements of type `UML::Classifier`
- Primitive types

Multiplicity is respected
(often restricted to multiplicities: 0..1, 1, *, 1..*)

Uniqueness

Ordering

Attributes can be read and written

Local variables treated like attributes
Action Languages
Main Mapping Choices

- A class may map to
  - a class declaration in an OO programming language
  - a structure declaration in a programming language
  - a CORBA IDL
  - an EJB
  - a database table
  - …

- An association may map to
  - a link between objects
  - a pointer
  - a hashtable
  - a database table

- A generalization may map to
  - a subclass definition
  - a link
Manipulate instances of classes: objects

Create:

```java
thePub = new Publisher
    {name="AW", address="Boston"};
```

just creation, no constructor

Read attribute:

```java
thePub.name
```

Write attribute:

```java
thePub.name = "MGH";
```

Delete:

```java
delete thePub;
```

debatable if this should be left to the programmer
Manipulate instances of associations: links

Create:

```plaintext
association.add{ end=obj-ref, end=obj-ref };
```

Delete:

```plaintext
association.delete{ end=obj-ref, end=obj-ref };
```

Traverse:

```plaintext
obj-ref.association
obj-ref.association-end
obj-ref.class
```

N-ary associations?

Association classes?
over members of classes

```java
for(x : class) { ... }
for(x : class) where condition { ... }
```

over navigable associations

```java
for(class x : obj-ref.association) { ... }
for(class x : obj-ref.association)
    where condition { ... }
```

over associations

```java
for({ end=obj-ref, end=obj-ref } : association) { ... }
for({ end=obj-ref, end=obj-ref } : association)
    where condition { ... }
```
Example: Using the Action Language
Data Model of a Bookshop

Category
- name

Book
- title
- abstract
- price

Author
- name

OrderItem
- quantity

Order

Customer
- name
- password

Address
- street
- town
- country

CreditCard
- cardnumber
- expiry
Category getCategory (String category) {
  for (cat : Category)
    where cat.name = category {
      return cat;
    }
  return new Category { name= category };}

Example: Using the Action Language

Add a New Book

```java
Book newBook (String title, Number price,
            String category, String author) {
    Author theAuthor =
        new Author { name= author };
    Category theCategory =
        getCategory (category);
    Book theBook =
        new Book { title= title, price= price };
    BookHasAuthor.add
        { book= theBook, author= theAuthor };
    BookHasCategory.add
        { book= theBook, category= theCategory };
    return theBook;
}
```
Example: Using the Action Language

A Database Mapping

- All classes mapped to database table
  - Object reference mapped to primary key value
  - for (...) where mapped to select
  - new mapped to insert

- Generic choice for associations: table with foreign keys
  - add mapped to insert
  - traversal mapped to select
  - delete mapped to delete
Support for Relations in Programming Languages

- Action Languages are unusual compared to other programming languages where
  - Support for objects, inheritance etc is abundant
  - Support for relations is virtually non-existent!
- Notable exceptions
Relations in Programming Languages

Why?

- Common agreement: relations are useful for conceptual modeling
- Later phases elide relations
  - Relations are implemented on an ad-hoc basis
    - Collaborating methods and attributes in participating classes
    - Collection class “Relation” holding sets of $n$-tuples
  - Relationship patterns exist
  - Relations implemented by model transformation
    - writers of protected regions must know the transformation:
      - name conventions
      - attribute types
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relationship as Attribute</td>
<td>Unidirectional, many-one or one-one relationship</td>
</tr>
<tr>
<td>Relationship Object</td>
<td>Large, complex relationship</td>
</tr>
<tr>
<td>Collection Object</td>
<td>Unidirectional, one-many relationship</td>
</tr>
<tr>
<td>Active Value</td>
<td>Globally important one-one relationship</td>
</tr>
<tr>
<td>Mutual Friends</td>
<td>Bidirectional relationship</td>
</tr>
</tbody>
</table>
Scope: unidirectional, one-one or many-one relationships

Very common

Changes only of local importance

Bookshop example:
  - Book → Category
  - Customer → CreditCard
  - Customer → Address

⇒ Represent by an attribute in the source class
Relationship Patterns
Relationship Object

- **Scope**: large, complex relationships
  - Many participating objects (peers)
  - Bidirectional

- **Implementation using attributes possible but**
  - the relationship is dispersed
  - it is hard to spot in the program
  - thus hard to maintain

- **The relationship object**
  - contains all methods and attributes to maintain the relation
  - may contain subordinate objects which are not visible outside
  - mediates between all objects participating in the relation

- **Bookshop example**: OrderItem ↔ Order
Scope: unidirectional, one-many relationships

Very common

⇒ Represent by an attribute in the “one” object which holds the “many” objects in a collection object

Example:

- Book → Author
  - . . . but the other direction is also needed in this case

Particular kind of Relationship Object
Relationship Patterns

Active Value

- One-one relationship
- With notification if one of the related objects changes
- Example: Window is in one-one relationship with the value of each input field
- Active Value: An object that reifies a single variable
- With setter and getter and change detection via Observer
Bidirectional Relationship

- All participating objects are equally important
- Change at one end requires change at other end
- Example: Book ↔ Author

Mutual Friends has to steps

- Splitting the relationship in two unidirectional ones
- Keep the moieties consistent
  - one end is the leader, the other the follower
  - leader administers all changes
  - the follower delegates all changes to the leader

Simplest instance: bidirectional one-one relationship represented by two attributes
Conclusion

- Are patterns good or bad?
- Patterns point to drawbacks of programming languages
  - Composite Pattern: lack of sum types
  - Visitor Pattern: lack of suitable extension mechanisms
  - Relationship Patterns: lack of support for patterns in PLS
- Even worse
  - if other target models are considered (e.g. database tables)
  - if multiple target models are considered (e.g., a relation between a database entity and a POJO)
- Relationship manipulation should be part of an action language