Contents

- What’s the problem?
- UML Models
- Executable UML elements
- Executable UML behavior
- The Repository
- Model Compiler Concepts
- How model compilers work
- MDA
What’s the Problem?

Analysts create “high-level” application models…

…print them out…

…then the developers look at the models and write the production code

These models are independent of the implementation technology

Coding adds this “design detail”
What’s the Problem?

But once something is in a machine representation…

…we shouldn’t have to manually reinterpret it (“elaborate it”) into another.

The analyst’s diagrams don’t have to be complete, correct, or consistent models.

Keeping the diagrams in sync with the code is a daunting task.

The rework introduces development errors and obscures analyst errors.
What’s the Problem?

But once something is in a machine representation…

Because elaboration is stupid!

…we shouldn’t have to manually reinterpret it (“elaborate it”) into another.

The Ant appears courtesy of Leon Starr, *Executable UML*
Executable Models

Analysts
• create executable models
• test those executable models

...and compile the models to produce a running system

Developers
• buy, build, or adapt a model compiler for a software architecture
• map model elements to the architecture...
Executable Models

The analyst’s models must be complete, correct, and consistent—and the models are verifiable.

The process always moves forward…

…there’s no editing of the translated code, no round-tripping, no reverse-engineering, no getting out of sync.

The software design decisions are formalized in the mappings and translation archetypes.
Contents

- What’s the problem?
- UML Models
- Executable UML elements
- Executable UML behavior
- The Repository
- Model Compiler Concepts
- How model compilers work
- MDA
What’s UML?

“The Unified Modeling Language is a language for specifying, constructing, visualizing, and documenting artifacts of a software-intensive system”

—The UML Summary
What’s UML?

- Common notation for OOA / OOD
- Goal was to eliminate the notation wars
- One notation for many purposes
- Very large
- Concept of “profiles” for different uses
UML in Practice

Describing a problem “Requirements”

Modeling a solution “Analysis”

Documenting software “Design”
Just because these use the same notation (UML) doesn’t mean these are the same problem.

The models are more than just the observable behavior to the user.

The software design does not follow from the model of the problem.
Each diagram is well-defined, but how do they fit together?
Contents

- What’s the problem?
- UML Models
- Executable UML elements
- Executable UML behavior
- The Repository
- Model Compiler Concepts
- How model compilers work
- MDA
Our Focus

Describing a problem “Requirements”

Modeling a solution “Analysis”

Documenting software “Design”
A profile defines how these diagrams represent models

Information Model
(things in the problem and how they are related)

State Model
(lifecycle of an instance of a class)
UML Profiles

It also defines a set of diagrams that are derived from the basic models

Information Model  
(things in the problem and how they are related)

State Model  
(lifecycle of an instance of a class)

Object Communication Model  
(summary of object interactions)

Execution Trace  
(system behavior under one scenario)

© 2003 Stephen J. Mellor and Marc J. Balcer  All Rights Reserved
Classes

**Class**
*abstraction of common characteristics and common behavior*

**Attribute**
*abstraction of a single characteristic of a class*

directed attribute

**Association**
*abstraction of a relationship between classes*

**Event**
*received by the object’s state machine*
Lifecycles

Statement

- prepareStatement(statementDate)
- applyPayment(transaction)
- dueDateReached
- noPaymentDue

State

- a stage in the lifecycle

Event

- abstraction of a single signal received by the state machine

Transition

- abstraction of a progression from state to state
Each state has a procedure executed upon entry to the state

Actions specify what happens in the state procedure
Expressing Actions

- Need to express the semantics of the problem domain
  - in pseudocode
  - in Java itself?
  - in a Java-esque syntax?
  - in a flowchart?
  - in a dataflow diagram?
  - in pipes-and-filters?
Why not pseudocode?

**Statement Prepared**

entry/
for each transaction not yet billed
post that transaction to the statement

what does “post” mean?

in the world or belonging to this customer?

how do we know?

“the statement”
You talkin’ to me?
You wanna post that transaction to me?
Why not 3GL code?

- Binds the model to a particular implementation
- Requires other non-application decisions to be made at the same time

**Statement Prepared**

entry/
query = "SELECT * FROM TBL_TRANS, ...
set rst=CurrentDB().open(query)
while not rst.EOF
  rst.Edit
  rst("R5StatementPtr") = statementID
  rst.Update
  rst.MoveNext
wend
rst.Close

*(Microsoft Visual Basic...sort of)*
What to do about actions?

- Pseudocode is not executable
- 3GLs bind model to a particular architecture
- We want to write
  - from a problem perspective
  - in terms of the model
  - not presuming a specific implementation
What about actions?

- UML (pre – 1.5) did not have a formalism for actions
- Precise Action Semantics specification supplements the UML
  - functional/dataflow-based
  - established the executable profile
  - essential for executability
UML Action Semantics

- Integrated into UML 1.5
- Fundamental actions on UML elements (classes, associations, …)
- Foundations for writing processing in an executable model
  - in the problem domain
  - does not presume an implementation
Semantics, not Language

- Why not just design an action language?
  - Highly political
  - Many vendors had some proprietary (but not fully compliant) action languages.
  - Syntax issues would obscure semantic issues
- First define a uniform semantics
Action Languages

TALL

foreach transaction in
    self->Account->Transaction[not .isBilled]
    self->Transaction := transaction;
endfor;

OAL

foreach transaction related by
    self->Account[R12]->Transaction[R3]
    where not selected.isBilled
    relate transaction to self across R5;
end for;

- Several different tool-specific languages
- First step is to make all action-semantics compliant
- Syntax is secondary
Specify the logic for each state’s action.

Statement Prepared

entry/
foreach transaction related by
self->Account[R12]->Transaction[R3]
where not selected.isBilled
relate transaction to self across R5;
end for;
currentBalance = 0; 
foreach transaction related by 
  self->Transaction[R3] 
  currentBalance = currentBalance + 
    transaction.netAmount; 
end for;

select many accounts from instances of Account 
where selected.number == self.number; 
return ((cardinality accounts) == 1);
An Executable Model

Classes

Lifecycle of Statement

Statement Prepared
entry/
foreach transaction related by
self->Account[R12]-->Transaction[R3]
where not selected.isBilled
relate transaction to self across R5;
end for;

Statement Prepared state procedure

© 2003 Stephen J. Mellor and Marc J. Balcer All Rights Reserved
Table of contents

- What’s the problem?
- UML models
- Executable UML elements
- Executable UML behavior
- The repository
- Model compiler concepts
- How model compilers work
- MDA
An executable model operates on instances.

**Account**
- number: CardAccountNumber
- currentBalance: Money
- actualCreditLimit: Money
- requestedCreditLimit: Money
- interestRate: Percent
- open()
- approve()
- suspend()
- reinstate()
- close()

**Statement**
- statementDate: Date
- minimumAmountDue: Money
- paymentDueDate: Date
- prepareStatement()
- applyPayment()
- dueDateReached()
- noPaymentDue()
An executable model operates on instances.

**Instances**

- **Statement A**
  - prepareStatement( statementDate )
  - dueDateReached
  - applyPayment( transaction )
  - noPaymentDue
  - Paid On Time
  - Payment Past Due
  - No Payment Due

- **Statement B**
  - prepareStatement( statementDate )
  - dueDateReached
  - applyPayment( transaction )
  - Paid Late

- **Statement C**
  - prepareStatement( statementDate )
  - dueDateReached
  - applyPayment( transaction )
  - Paid Late

**Statement**

- /statementDate : Date
- minimumAmountDue : Money
- paymentDueDate : Date
- prepareStatement() applyPayment() dueDateReached() noPaymentDue()
Execution

- The lifecycle model prescribes execution

When the customer’s payment is received, the statement transitions to the next statement and executes actions.
Executing the Model

- The model executes in response to signals from:
  - the outside,
  - timers (delayed events)
  - other instances as they execute

```java
prepareStatement( statementDate )

Statement Prepared

applyPayment( transaction )

dueDateReached

Paid On Time

Payment Past Due

No Payment Due

applyPayment( transaction )

Paid Late

noPaymentDue
```
Concurrent Execution

All instances execute concurrently.

statement A

statement B

statement C
Multiple events may be received.

Billing Clerk

- receivePayment( accountNumber, date, amount )
- prepareStatement( statementDate )
- applyPayment( transaction )
- dueDateReached
- noPaymentDue
- Paid On Time
- Payment Past Due
- No Payment Due
- Paid Late
- checkCleared
- Accepted
Concurrent Execution

- Concurrent events can produce different outcomes

Two outcomes if the payment is received as the due date passes
Verification

- Test models in the development environment
- Check the models just like code
Table of contents

- What’s the problem?
- UML models
- Executable UML elements
- Executable UML behavior
- The repository
- Model compiler concepts
- How model compilers work
- MDA
Compilation requires that the models be available in a metamodel-based model repository.
The repository holds the developers’ models.
Metamodel

```
<<Imported>>
Domain (1\Domain)
  name: string
  description: DescriptionString

<<Imported>>
AssociationConstraint (1\AssociationConstraint)
  associationName: integer (1\Association)
  description: DescriptionString

<<Imported>>
Datatype (2\Datatype)
  name: string
  description: DescriptionString

<<Imported>>
ClassConstraint (3\ClassConstraint)

<<Imported>>
Class (1\Class)
  classNumber: integer
  name: string
  abbreviation: string
  description: DescriptionString

<<Imported>>
AbstractClass (1\AbstractClass)
  classNumber: integer
  discriminator: string
  isComplete: boolean

<<Imported>>
ConcreteClass (1\ConcreteClass)
  classNumber: integer

<<Imported>>
Transformer (1\Transformer)
  name: string

<<Imported>>
AssociationClass (1\AssociationClass)
  associationNumber: integer
  name: string
  abbreviation: string
  description: DescriptionString

<<Imported>>
AssociationEnd (1\AssociationEnd)
  associationNumber: integer
  classNumber: integer
  phrase: string
  multiplicity: integer

<<Imported>>
BaseAttribute (1\BaseAttribute)
  name: string
  description: DescriptionString

<<Imported>>
DerivedAttribute (1\DerivedAttribute)
  name: string
  description: DescriptionString
  computes value of
  value is computed by

<<Imported>>
AttributeFormula (1\AttributeFormula)
  name: string
  description: DescriptionString
  return type is
  return type is defined return type of
```

OOA of OOA

© 2003 Stephen J. Mellor and Marc J. Balcer All Rights Reserved
Metamodel Instances

- Just like an application model, the metamodel has instances.

<table>
<thead>
<tr>
<th>classID</th>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Customer</td>
<td>A Customer is any individual or business…</td>
</tr>
<tr>
<td>101</td>
<td>Account</td>
<td>An account represents a customer’s financial…</td>
</tr>
<tr>
<td>102</td>
<td>BillingGroup</td>
<td>A set of accounts that are all billed periodically…</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>classID</th>
<th>stateID</th>
<th>stateName</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1</td>
<td>NewAccount</td>
</tr>
<tr>
<td>101</td>
<td>2</td>
<td>In Good Standing</td>
</tr>
<tr>
<td>101</td>
<td>3</td>
<td>Suspended</td>
</tr>
<tr>
<td>101</td>
<td>4</td>
<td>Closed</td>
</tr>
</tbody>
</table>
Repository Instances

Modeling tools create instances in the repository

<table>
<thead>
<tr>
<th>classID</th>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Customer</td>
<td>A Customer is any individual or business...</td>
</tr>
<tr>
<td>101</td>
<td>Account</td>
<td>An account represents a customer’s financial...</td>
</tr>
<tr>
<td>102</td>
<td>BillingGroup</td>
<td>A set of accounts that are all billed periodically...</td>
</tr>
<tr>
<td>103</td>
<td>Statement</td>
<td>A statement is a periodic preparation of the ...</td>
</tr>
</tbody>
</table>
Table of contents

- What’s the problem?
- UML models
- Executable UML elements
- Executable UML behavior
- The repository
- Model compiler concepts
- How model compilers work
- MDA
A model compiler is the automated embodiment of how to turn a model into software under a set of rules.

Analysts create models

Developers program the rules

Model Compiler produces a running system
Model Compiler

…a program that creates code from models…

Analysts create models

Developers program the rules

Model Compiler produces a running system
Model Compiler

…something that makes models executable…

Analysts create models

Developers program the rules

Model Compiler produces a running system
Model Compiler

Analysts create models

Developers program the rules

...it’s a model compiler.

Model Compiler produces a running system
A model compiler reads executable UML models and creates a system with certain dimensions.
Examples

Financial system
- Highly distributed
- Concurrent
- Transaction-safe with rollback
- Persistence with rollback
- J2EE (Java + EJB)

Embedded system
- Single task
- No operating system
- Optimized data access and storage
- C

Telecommunication system
- Highly distributed
- Asynchronous
- Limited persistence capability
- C++

Simulation system
- Mostly synchronous
- Few tasks
- Special-purpose language “Import”
Table of contents

- What’s the problem?
- UML models
- Executable UML elements
- Executable UML behavior
- The repository
- Model compiler concepts
- How model compilers work
- MDA
Archetypes
Translation rules interpreted by a generator against the repository

Execution Engine
A limited set of reusable components sufficient to execute Executable UML

.Model Compiler

.function Class
    Class $(Class.name) :
    public ActiveInstance {
        private:
            .invoke PrivateDataMember(Class);
    };

.function PrivateDataMember
    .param Class Class
    .select many PDM from instances of Attribute related to Class
    .for each PrivateDataMember
        $(PDM.Type) $(PDM.Name);
    .endfor
Archetypes traverse the repository and...

<table>
<thead>
<tr>
<th>classID</th>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Customer</td>
<td>A Customer is any individual or business…</td>
</tr>
<tr>
<td>101</td>
<td>Account</td>
<td>An account represents…</td>
</tr>
<tr>
<td>102</td>
<td>BillingGroup</td>
<td>A set of accounts…</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>classID</th>
<th>stateID</th>
<th>stateName</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1</td>
<td>NewAccount</td>
</tr>
<tr>
<td>101</td>
<td>2</td>
<td>In Good Standing</td>
</tr>
<tr>
<td>101</td>
<td>3</td>
<td>Suspended</td>
</tr>
<tr>
<td>101</td>
<td>4</td>
<td>Closed</td>
</tr>
</tbody>
</table>

... output text.
Archetypes direct the generation of text.

```
.Function Class
Class ${Class.name} : public ActiveInstance {
private:
.invoke PrivateDataMember( Class )
};

.Function PrivateDataMember
.param Class Class
.select many PDM from instances
  of Attribute related to Class
.for each PrivateDataMember
  ${PDM.Type} ${PDM.Name};
.endfor
```

Placeholder introduced by ${...}

Text (which happens to be C++)
The archetype language produces text.

```plaintext
select many states related to instances of
class->[R13]StateChart->[R14]State
where (isFinal == False)
public:
  enum states_e
  { NO_STATE = 0 ,
    for each state in states
      if ( not last states )
        '${state.Name}' ,
      else
        NUM_STATES = '${state.Name}'
  endif
  endfor
};

public:
  enum states_e
  { NO_STATE = 0 ,
    NewAccount ,
    InGoodStanding ,
    Suspended ,
    NUM_STATES = Closed
  };
```
Mappings

- Design-time assignment of different archetypes to distinct elements in the model
- Separate from the model and the archetypes
- “Uniformity != Rigidity”
Table of contents

- What’s the problem?
- UML models
- Executable UML elements
- Executable UML behavior
- The repository
- Model compilers
- Model compiler concepts
- How model compilers work
- MDA
Model Driven Architecture

- A standard for software development using formal, executable and compilable system specifications
- Standards should be based on executable UML including the Action Semantics.
- Interchange can be managed using XML
Executable UML

- **Entry tools:** BridgePoint®, Rose®, others
- **Verification tools:** T-Vec®, S/R®, COSPAN®

Model Compilers turn Executable UML models into systems.
Building a Market

- Design time composability
  - protects IP

The models are a valuable corporate resource that are true business models—they describe the business, not the software.
Building a Market

- Design time composability
  - protects IP
  - allows IP to be mapped to multiple implementations

*Just because we move to a new software platform doesn’t mean that we need to recreate the rules of the bank.*
We can sell “Billing” models that can run on many platforms and support many different businesses.

- Design time composability
  - protects IP
  - allows IP to be mapped to multiple implementations
  - enables a market in IP in software
The Future

- Executable UML enables a market in model compilers.

- Application experts build models....
  - ....select the appropriate model compiler, and...
  - ....compile the model.

- Each model compiler targets a specific software platform.
Conclusion

- Executable UML, using archetypes to direct the generation, enables:
  - early error detection through verification
  - reuse of the execution engine
  - faster performance tuning
  - faster integration
  - faster, cheaper retargeting