MOF 2.0 IDL

CORBA Components based Repositories

Dr. Olaf Kath
kath@ikv.de
Models describe all aspects of IT systems.
General Model Infrastructure

- Business Model
- Usage Model
- Test Model
- Resource Model
- Design Model
- Architecture Model
- Exception Model
- Execution Model
- Deployment Model
Models
and Metamodels
are exchanged at runtime.
General Model Infrastructure

- Business Model
- Usage Model
- Test Model
- Resource Model
- Design Model
- Architecture Model
- Exception Model
- Execution Model

medini

IKV++’s Enterprise Metadata Infrastructure
(CORBA+MOF, Java+JMI, ...)

© IKV++ Technologies AG, 2002
The enago MEDINI Tool Chain

- XMI specifications
- technology independent models
- enago medini UML specifications
- .NET and Web Services environment
- Java 2 EE environment
- CORBA Components environments
- XML-RPC based messaging platforms
- model transformations for specific environments
CORBA Components based MOF 2.0 IDL Mapping

- We outline a mapping of MOF 2.0 compliant models to CORBA IDL
  - Mapping rules are not yet complete nor prototyped
  - Further detailed definition heavily depends on the progress of MOF 2.0 Core
**CORBA Components**

**based MOF 2.0 IDL Mapping**

- We outline a mapping of MOF 2.0 compliant models to CORBA IDL
- We reuse the concepts of the CORBA Component Model
  - Aim: generation of highly performant, highly scalable and reliable repositories, which are automatically deployable
    - Performance, scalability and other, non-functional aspects depend on the underlying container
    - Hint: this could also be useful for a Java mapping with EJBs as mapping target
  - Use of platform mechanisms to simplify the generated IDL definitions, e.g.
    - Use home interfaces for the creation class instances,
    - Use (multiple) receptacles for the representation of references
    - Use event sources/sinks to support...
**CORBA Components**

**based MOF 2.0 IDL Mapping**

- We outline a mapping of MOF 2.0 compliant models to CORBA IDL
- We reuse the concepts of the CORBA Component Model
- The general mapping scheme is defined using abstract interfaces in a way allowing
  - a CCM specific mapping "profile" (outlined in the submission)
  - a non-CCM mapping "profile" (to be defined in a revision of the submission)
  - a federation of both approaches (implied !?)
CORBA Components based MOF 2.0 IDL Mapping

• We outline a mapping of MOF 2.0 compliant models to CORBA IDL
• We reuse the concepts of the CORBA Component Model
• The general mapping scheme is defined using abstract interfaces
• For the definition of the mapping rules, we use OCL
  – We derive one model from the MOF 2.0 Core and the CCM Models
  – We define a mapping rule as invariant
  – For the revision of the submission, we need to consider
    • CWM transformation maps
    • MOF2 QVT
    • Some combination of CWM 2 and MOF 2 QVT
**OCL based Mapping Rules Definition**

**Rules in English**

[9] If the class is not abstract, an IDL component definition is being generated in the same module as the abstract interface with the name concatenate (format_1(<class identifier>), "Component"). The component is declared to support the generated abstract interface for the class.

[11] If the class is not abstract, a home interface declaration for the component with the name concatenate (format_1(<class identifier>), "Home") is being generated, managing the component generated following Rule (9). This home interface contains a factory operation to create a component without parameters with the name concatenate ("create_", format_2(<class identifier>)).

**Constraints in OCL**

[9,11] if not self.isAbstract then

```ocl
self.c_target->size() = 1 and self.home_target->size() = 1 and
self.c_target.identifier = concat(format_1(self.identifier), "Component") and
self.home_target.identifier = concat(format_1(self.identifier), "Home") and
self.container.target = self.c_target.definedIn and
self.container.target = self.home_target.definedIn and
self.home_target.manages = self.c_target and
self.c_target.supports = self.target
```

else

```ocl
self.c_target->size() = 0 and self.home_target->size() = 1 and
Class.allInstances()->forAll(c | not c.isAbstract implies
    c = self.home_target.factory and
    c.identifier = concat("create_", format_2(self.identifier))
)
```

endif
**MOF 2.0 Collections Mapping**

- In MOF 1.4, any collection is mapped to IDL sequences
  - Inconvenient for large collection values in case of get operations

- We map collections to abstract interfaces ...
  
  abstract interface ULongSetIterator {
    unsigned long get_value ();
    void set_value ( in unsigned long value );
    ULongSetIterator next_one();
    // more operations, like begin, end, previous_one
  };

- ... and valuetypes/concrete interfaces
  
  valuetype ULongSetAsValue supports ULongSetIterator {
    private ULongSet value;
    // factory operations, ...
  };

  interface ULongSetAsReference : ULongSetIterator {
    ULongSetAsValue as_value();
  };

© IKV" Technology AG, 2002
MOF 2.0 Classes Mapping

- A class of a model is mapped to
  - an abstract interface definition (the hook for all mapping profiles),
  - a concrete valuetype (to communicate state),
  - a component definition (representing the instances) and
  - a home definition (to create instances and manage their life cycle)

```
module MyPackage {

  abstract interface MyClass {
  }

  valuetype MyClassState supports MyClass {
    }

  component MyClassComponent supports MyClass {
    readonly attribute MyClassState my_class_state;
  }

  home MyClassHome manages MyClassComponent {
    factory create_my_class ();
  }

};
```
**MOF 2.0 Classes Mapping**

- Generalization relations between classes are mapped to inheritance of the abstract interfaces produced for these classes
  - The component and valuertype definitions do NOT inherit, just the abstract interfaces

```plaintext
module MyPackage {

    abstract interface A { };
    abstract interface B : A { };

    valuetype AState supports A { };
    valuetype BState supports B { };

    component AComponent supports A { };
    component BComponent supports B { };

    // home definitions accordingly
};
```
MOF 2.0 Classes Mapping

- Attributes are mapped to
  - access operations of the abstract interfaces
    - like operations in MOF 1.4 for instance/class proxy interfaces
    - get, set, add, modify, ...
  - members of the valuetype declarations
    - For an attribute a_attrib, type long of class A, the valuetype declaration is
      ```
      valuetype AState supports A {
        public long a_attrib;
      }
      ```
  - parameters of the component factory operations
    - The home interface for the above example is
      ```
      home AHome manages AComponent {
        factory create_a ();
        factory create_and_init_a ( in long a_attrib );
        factory copy_from ( in A the_a );
      };
      ```
MOF 2.0 Classes Mapping

- Attributes are mapped to
  - if the upper bound of the attribute is e.g. UNBOUNDED, the generated IDL uses the iterator interfaces for collections
    
```
abstract interface LongSetIterator { /* … */ };
```

```
valuetype LongSetAsValue supports LongSetIterator { /* … */ };
```

```
interface LongSetAsReference : LongSetIterator { /* … */ };
```

```
abstract interface A {
    void get_a_attrib ( out LongSetIterator a_attrib, in boolean as_value );
    void set_a_attrib ( in LongSetIterator a_attrib );
    // ...
};
```

- The same approach is used for references
MOF 2.0 Associations Mapping

- An approach similar to the mapping of classes is used for associations
  - they map to abstract interface, component and home definitions

```
module MyPackage {
  abstract interface A { };
  abstract interface B { };

  abstract interface AB {
    boolean exists ( in A the_a, in B the_b);
    A the_a ( in B the_b);
    B the_b ( in A the_a);
    void add ( in A the_a, in B the_b);
    void modify_the_a ( /*...*/ );
    void modify_the_b ( /* ... */ );
    void remove ( /* ... */ );
  };

  component ABComponent supports AB { };
  home ABHome manages ABComponent { };
```
**MOF 2.0 Packages Mapping**

- **Same approach used**
  - Packages are mapped to abstract interface, component and home definitions
  - For each contained class/association/package, the abstract interface contains an access operation

```java
module MyPackage
{
    abstract interface MyClass;
    component MyClassComponent;
    home MyClassHome;
    abstract interface MyPackagePackage {
        AHome a();
    };
    component MyPackageComponent
        supports MyPackagePackage { };
    home MyPackageHome
        manages MyPackageComponent { };
}
```
Additional Repository Services: Reflection

- Reflection is treated as a repository service
  - Under discussion: optional or required for all model elements
  - Services are generally mapped to facets of the component definitions produced for these model elements
  - For a class A of a model, the component definition is extended

```java
abstract interface A { }
valuetype AState supports A { }
component AComponent supports A {
  readonly attribute AState a_state;
  provides Reflective::RefObject reflective;
}
```
Additional Repository Services: Active Repository

- We propose to add a new feature
  - we call it Active Repository,
  - Is used to initiate the communication of model changes actively by the repository
  - We use the publish/subscribe model of CCM for realization
  - If a class A supports the feature, the produced IDL definitions are

```plaintext
abstract interface A { }
valuetype AState supports A { }

eventtype AChanges : AState { }

component AComponent supports A {
  readonly attribute AState a_state;
publishes AChanges changes_a;
}
```
Conclusions

- Mapping for all elements of any MOF 2.0 compliant model to CORBA IDL
  - Experimentation of the approach promises the ability
  - The produced IDL appears intuitive and simpler than 1.4
  - Contributing: distinction in the mapping of an element and additional features, like reflection and active repository
  - Produced IDL definitions for classes, associations and packages appear very similar
  - The class proxy interfaces of 1.4 do not exist anymore
  - The generation of abstract interfaces allow for a “non-CCM”-profile (TBD)
Conclusions

- Mapping for all elements of any MOF 2.0 compliant model to CORBA IDL
- Reuse of CCM platform features to enable performant and scalable repository implementations
  - CORBA Components Architecture provides all mechanisms to develop highly performant, scalable and automatically deployable component based software systems
  - Using this approach, one can go much further:
    - the PSDL specification for the state attributes of an element can be generated, the persistence support of the container can be used instantly,
    - Lifecycle management issues and identity of the representation of a model element as component can be automatically implemented using the entity component and primary key concept of CCM,
    - The implementation definition itself can be generated from a model as well, using the Component Implementation Framework mechanism,
    - A repository generated from a MOF 2.0 compliant model is ad-hoc automatically deployable.
Conclusions

- Mapping for all elements of any MOF 2.0 compliant model to CORBA IDL
- Reuse of CCM platform features to enable performant and scalable repository implementations

Further Work
- Completion of the mapping rules
- Consideration of CWM transformation maps, MOF2 QVT or some combination of CWM 2 and MOF 2 QVT for the formalization (currently OCL)
- Definition of a non-CCM profile
  - Hopefully together with contributors ???
- Experimentation and prototyping
  - We hope to have a prototype running at the meeting of the revised submission presentation