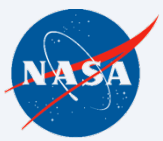


National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

IMCE MOF2 / OWL2 Integration

Nicolas Rouquette
System Architectures & Behaviors Group, 313K

2012-03-20

National Aeronautics and Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

Systems Engineering Domain-Specific Interest Group (SE-DSIG) & Ontologies

- Task list from the OMG March 2011 meeting:
  - We have the opportunity to place a formal mathematical semantic foundation under SysML on a par with the semantic foundation for OWL2 and other math-logic based systems. The result will provide the level of precision needed to:
    - 1. enable users to build models whose semantics is precise, mitigating misunderstanding by human or machine users
    - 2. enable integration of reasoning with SysML development based on a standard of correctness
    - 3. help evolve SysML to meet future expressiveness and integration needs
  - We can achieve this goal by a series of incremental steps, each of which provides incremental value to the SysML community. The approach will leverage development work such done for OWL and other mathematics base systems.
  - Below are suggestions for some discrete tasks which will move the semantic effort forward. The tasks typically review current state, analyze and produce a recommended solution, with guidance for implementation.
- **T1. Extend SysML to include OWL2 class constructions with OWL2 semantics. (Nicolas Rouquette)**
  - **The benefit is that we can use OWL reasoning to fill in class hierarchies. The effort will have side effects of clarifying and extending instances/individuals for SysML. The need to represent structure in SysML models also overlaps with cutting edge research in Description Logic which is the foundation for OWL2.**
- T2. Review and extend SysML Constraint constructions to enable expressing pre and post condition formal requirements on operators declared for a block, such as a pointing operation on a sensor block, and describe the formal semantics in a way that enables proof of correctness of implementations to be generated.
  - (Yvonne Bijan, Roger Burkhart).
- T3. Produce a formal semantics for some portion of SysML behavior constructions.
  - (Yves Bernard).
- T4. Make an approach to the DARPA META program
  - (Rick Steiner).
- T5. Produce a survey of semantic standards to be used for expressing the formal semantics for SysML
  - (Conrad Bock, Rick Murphy).
- T6. Harmonize SysML's QUDV with related ontologies (QUDT, QUOMOS) and particularly with OMG's Date & Time

Outline

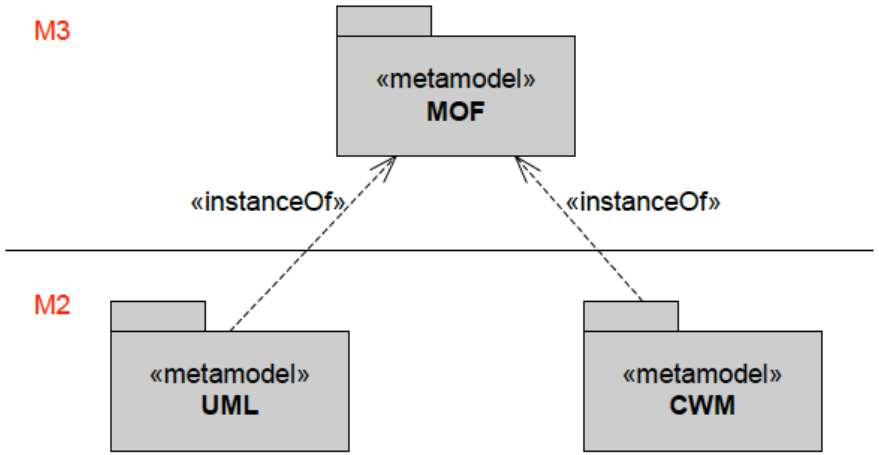
- What is UML2.4.1 ?
- What is MOF 2.4.1 ?
- UML & MOF revisited
- Clarifying the intended semantics of CMOF
  - 5 clues…
- Putting the clues together
- Formally verifying the clues make sense
  - OWL2 & MOF Reconciled with Alloy
  - Demo
- Conclusion

What is UML 2.4.1 ?

- **UML 2.4.1 Superstructure §6.4.1:**

  The abstract syntax is defined by a CMOF model (i.e., the UML metamodel) with each modeling concept represented by an instance of a MOF class or association.

- **UML 2.4.1 Infrastructure §7.5:**

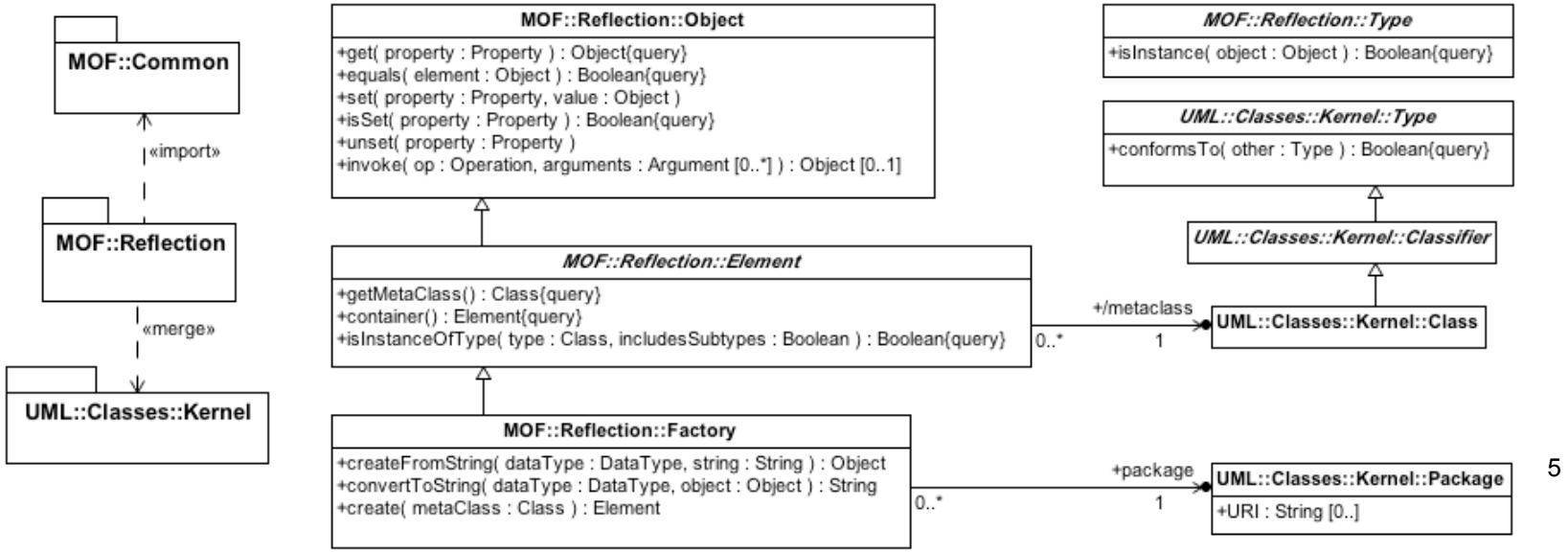


What is MOF 2.4.1 ?

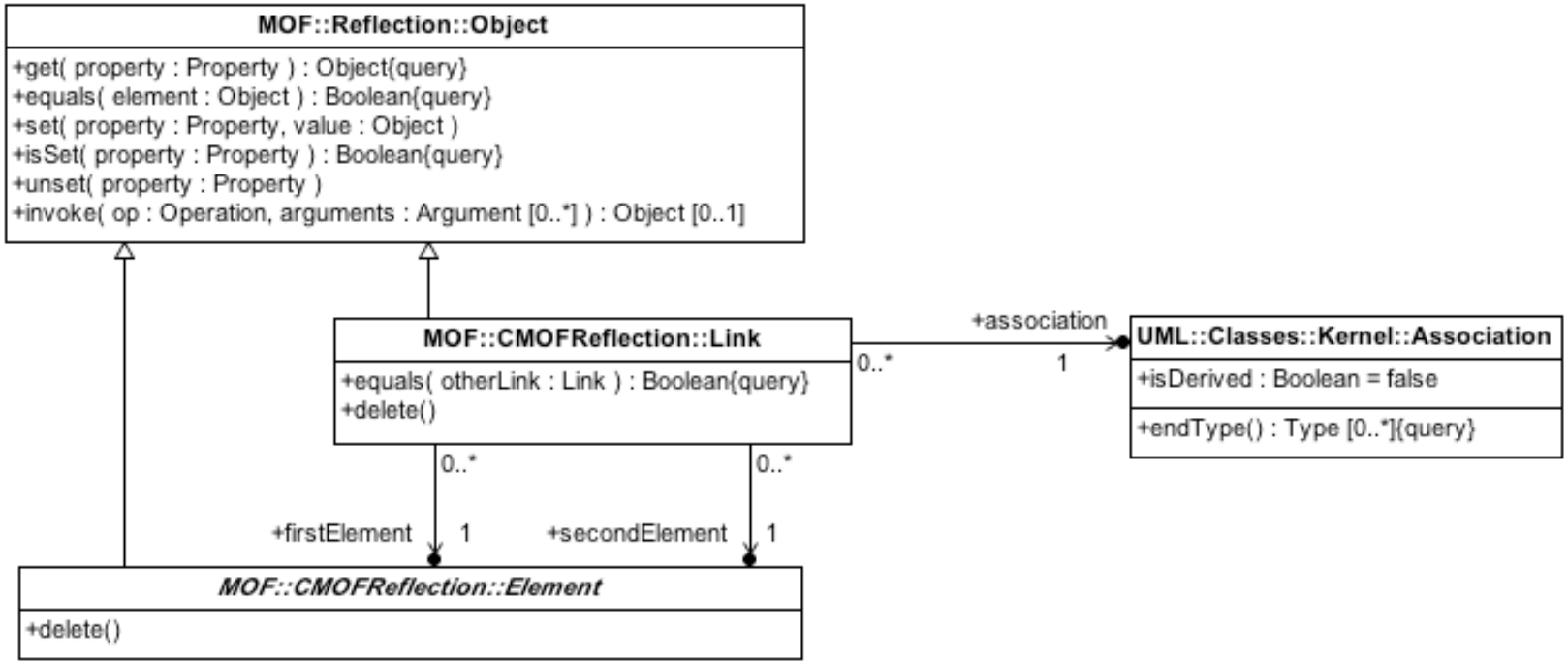
- **MOF 2.4.1 §8.1:**

  In particular, EMOF and CMOF are both described using CMOF, which is also used to describe UML2. EMOF is also completely described in EMOF by applying package import, and merge semantics from its CMOF description. As a result, EMOF and CMOF are described using themselves, and each is derived from, or reuses part of, the UML 2 Infrastructure Library.

- **Correction: EMOF is described in CMOF, not EMOF!**



UML & MOF revisited



- **UML is an instance of the CMOF metamodel**
- **CMOF is defined as UML + extensions**

⇒ **UML is an instance of something larger than UML (CMOF)**

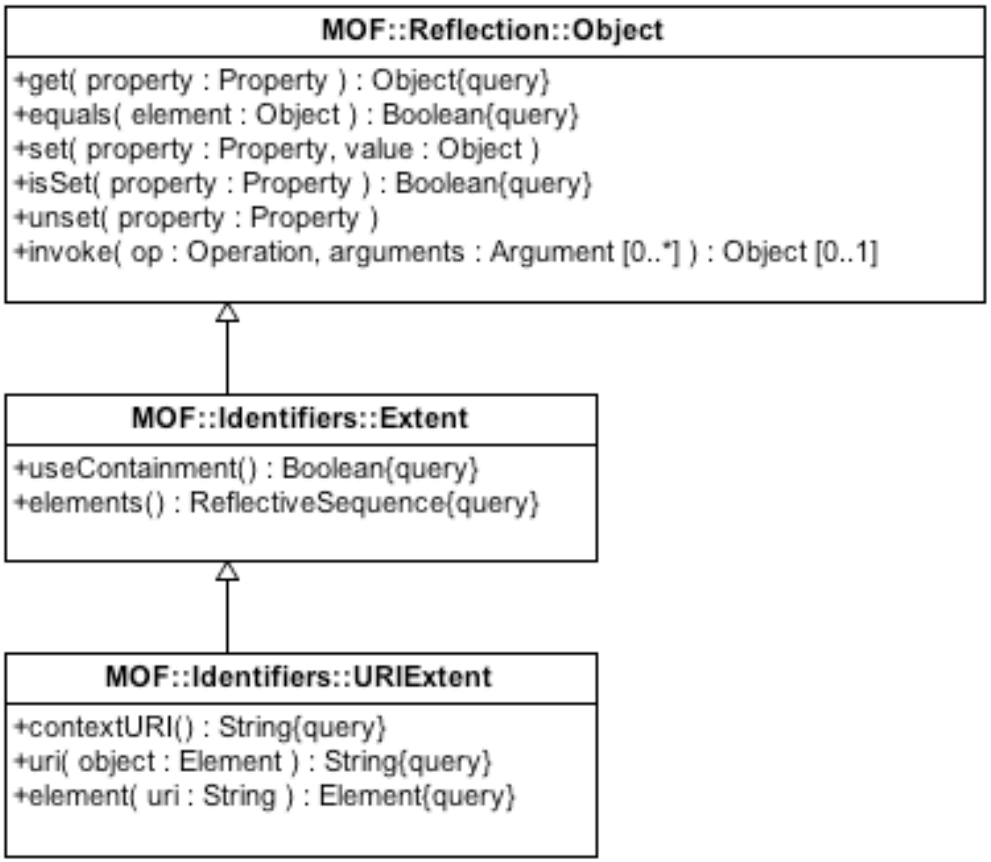
Clarifying the intended semantics of CMOF (With the help of MOF's friends)

- **On surface, this is tricky**
  - **How do we prioritize among entangled issues?**
  - **How do we make progress when there is no practice of CMOF?**
- **Where are MOF's friends?**
  - **Let's look at the clues:**
    1. **MOF Elements have unique Identifiers**
    2. **MOF Links are (incorrectly) specified**
    3. **OWL2's Simplified UML MOF Diagrams**
    4. **OWL2's Mandatory Concrete Syntaxes**
    5. **Metamodeling, *à la* OWL2**

Clue #1: MOF Elements have unique Identifiers

- **§10.1**

  An element has an identifier in the context of an extent that distinguishes it unambiguously from other elements.

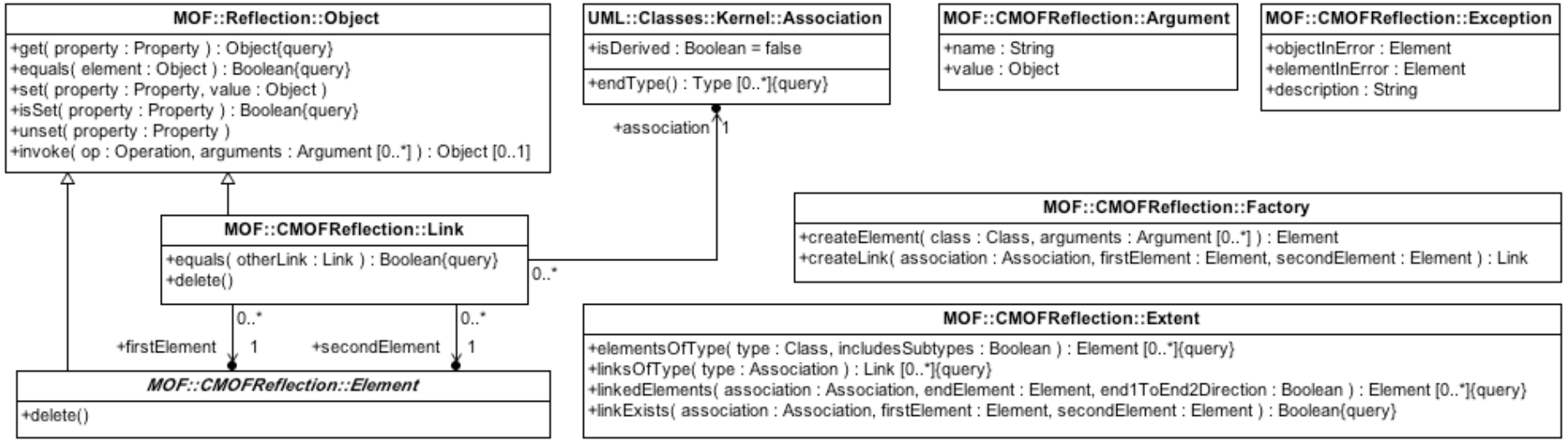


National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

Clue #2: MOF Links are (incorrectly) specified

- **Issue 16270 (mof2core-rtf)**
  - **MOF does not have the correct semantics for links in the presence of association specialization**
- **Issue 16233 (smof-ftf)**
  - **SMOF does not implement dynamic classification of associations**



Clue #3: OWL2's Simplified UML MOF Diagrams

2.1 Structural Specification

The structural specification of OWL 2 consists of all the figures in this document and the notion of structural equivalence given below. It is used throughout this document to precisely specify the structure of OWL 2 ontologies and the observable behavior of OWL 2 tools. An OWL 2 tool *may* base its APIs and/or internal storage model on the structural specification; however, it *may* also choose a completely different approach as long as its observable behavior conforms to the one specified in this document.

The structural specification is defined using the Unified Modeling Language (UML) [*UML*], and the notation used is compatible with the Meta-Object Facility (MOF) [*MOF*]. This document uses only a very simple form of UML class diagrams that are expected to be easily understandable by readers familiar with the basic concepts of object-oriented systems. The following list summarizes the UML notation used in this document.

- The names of the UML classes from the structural specification are written in bold font.
- The names of abstract UML classes (i.e., UML classes that are not intended to be instantiated) are written in bold and italic font.
- Instances of the UML classes of the structural specification are connected by associations, many of which are of the one-to-many type. Associations whose name is preceded by / are *derived* — that is, their value is determined based on the value of other associations and attributes. Whether the objects participating in associations are ordered and whether repetitions are allowed is made clear by the following standard UML conventions:
  - By default, all associations are sets; that is, the objects in them are unordered and repetitions are disallowed.
  - The `{ ordered,nonunique }` attribute is placed next to the association ends that are ordered and in which repetitions are allowed. Such associations have the semantics of lists.

http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/#Structural_Specification

Clue #4: OWL2's Mandatory Concrete Syntaxes

2.2 Syntaxes

In practice, a concrete syntax is needed in order to store OWL 2 ontologies and to exchange them among tools and applications. The primary exchange syntax for OWL 2 is RDF/XML [*RDF Syntax*]; this is indeed the only syntax that *must* be supported by all OWL 2 tools (see Section 2.1 of the OWL 2 Conformance document [*OWL 2 Conformance*]).

While RDF/XML provides for interoperability among OWL 2 tools, other concrete syntaxes may also be used. These include alternative RDF serializations, such as Turtle [*Turtle*]; an XML serialization [*OWL 2 XML*]; and a more "readable" syntax, called the Manchester Syntax [*OWL 2 Manchester Syntax*], that is used in several ontology editing tools. Finally, the functional-style syntax can also be used for serialization, although its main purpose is specifying the structure of the language [*OWL 2 Structural Specification*].

Name of Syntax	Specification	Status	Purpose
RDF/XML	Mapping to RDF Graphs, RDF/XML	Mandatory	Interchange (can be written and read by all conformant OWL 2 software)
OWL/XML	XML Serialization	Optional	Easier to process using XML tools
Functional Syntax	Structural Specification	Optional	Easier to see the formal structure of ontologies
Manchester Syntax	Manchester Syntax	Optional	Easier to read/write DL Ontologies
Turtle	Mapping to RDF Graphs, Turtle	Optional, Not from OWL-WG	Easier to read/write RDF triples

http://www.w3.org/TR/owl2-overview/#Syntaxes

Clue #5: Metamodeling, *à la* OWL2

5.9 Metamodeling

An IRI *I* can be used in an OWL 2 ontology to refer to more than one type of entity. Such usage of *I* is often called *metamodeling*, because it can be used to state facts about classes and properties themselves. In such cases, the entities that share the same IRI *I* should be understood as different "views" of the same underlying notion identified by the IRI *I*.

Example:

Consider the following ontology.

`ClassAssertion( a:Dog a:Brian )`	Brian is a dog.
`ClassAssertion( a:Species a:Dog )`	Dog is a species.

In the first axiom, the IRI *a:Dog* is used as a class, while in the second axiom, it is used as an individual; thus, the class *a:Species* acts as a metaclass for the class *a:Dog*. The individual *a:Dog* and the class *a:Dog* should be understood as two "views" of one and the same IRI — *a:Dog*. Under the OWL 2 Direct Semantics [*OWL 2 Direct Semantics*], these two views are interpreted independently: the class view of *a:Dog* is interpreted as a unary predicate, while the individual view of *a:Dog* is interpreted as a constant.

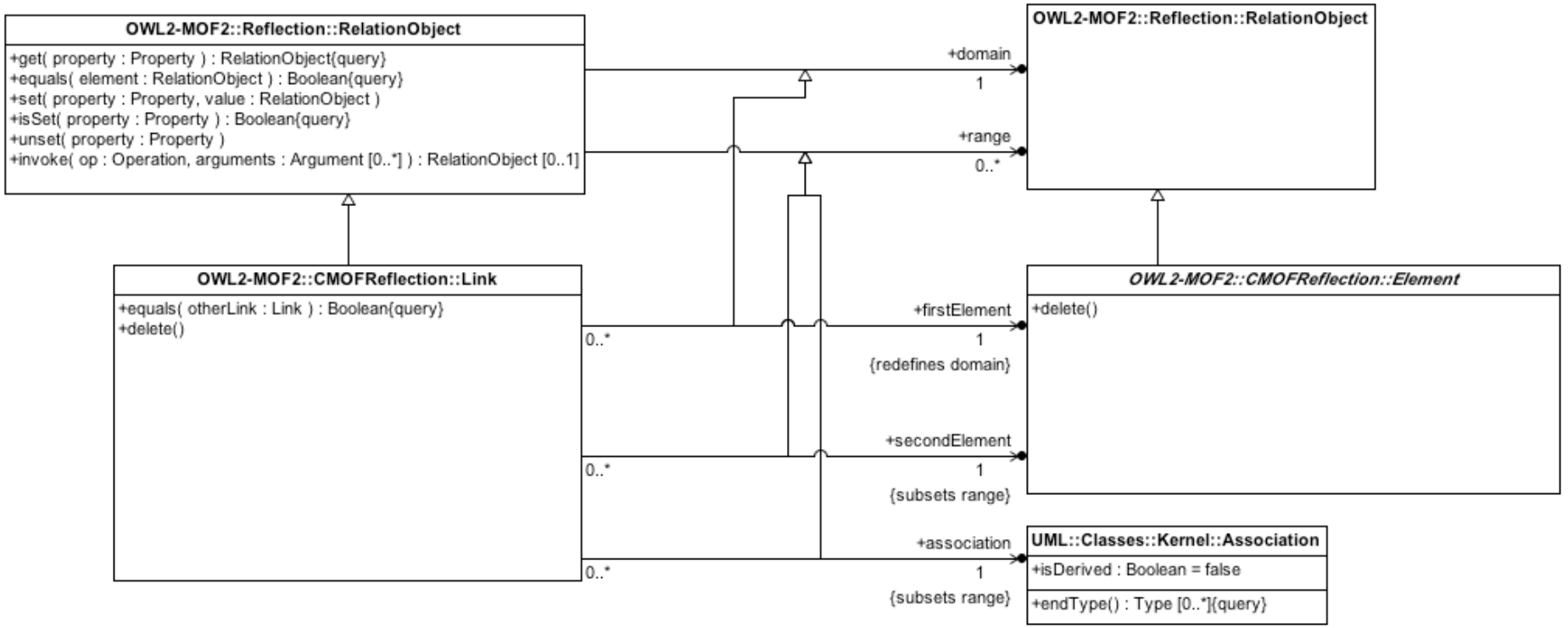
Both metamodeling and annotations provide means to associate additional information with classes and properties. The following rule-of-the-thumb can be used to determine when to use which construct:

- Metamodeling should be used when the information attached to entities should be considered a part of the domain.
- Annotations should be used when the information attached to entities should not be considered a part of the domain and when it should not contribute to the logical consequences of an ontology.

http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/#Metamodeling

Putting the clues together

- **A sketch of MOF2, reconciled**



Formally verifying the clues make sense

- **The verification involves two parts:**
  - **OWL2/MOF2 Integration**
    - **This is the infrastructure developed at JPL's Integrated Model-Centric Engineering (IMCE) project over the last ~ 3 years**
    - **This infrastructure is been extensively analyzed with several kinds of techniques:**
      - **SPARQL queries to audit IMCE OWL2 ontologies w.r.t. well-formedness criteria**
      - **Pellet OWL2-DL reasoning to verify the logical consistency, satisfiability and entailments of IMCE OWL2 ontologies**
      - **QVTO constructions of IMCE OWL2 ontologies for UML 2.4.1 & SysML 1.3**
  - **A new Alloy4 formal model of the reconciled MOF2 stack**
    - **The relational kernel is M4**
    - **CMOF is an M3 metamodel constructed as an instance of M4**
    - **UML, BPMN, SysML.. are M2 metamodel instances of CMOF @ M3**

OWL2 & MOF Reconciled with Alloy: 1) The simplest reflexive M4 metamodel

- **At the top is the simplest reflexive metamodel…**

```
// M4 kernel

// There is only one concept: a Relation with a domain & a range.
// This concept is enough to construct everything in the OWL2 and MOF2 metamodels.
// Both metamodels are simplified UML models (roughly, class diagrams)

// In practice, it is useful to differentiate between
// two kinds of relations depending on the range multiplicy

abstract sig BinaryRelation {
  domain : one BinaryRelation,
  range : one BinaryRelation
}

abstract sig SetRelation {
  domain : one BinaryRelation + SetRelation,
  range : set BinaryRelation
}
```

OWL2 & MOF Reconciled with Alloy: 2) Reconstructing CMOF's M4 metamodel

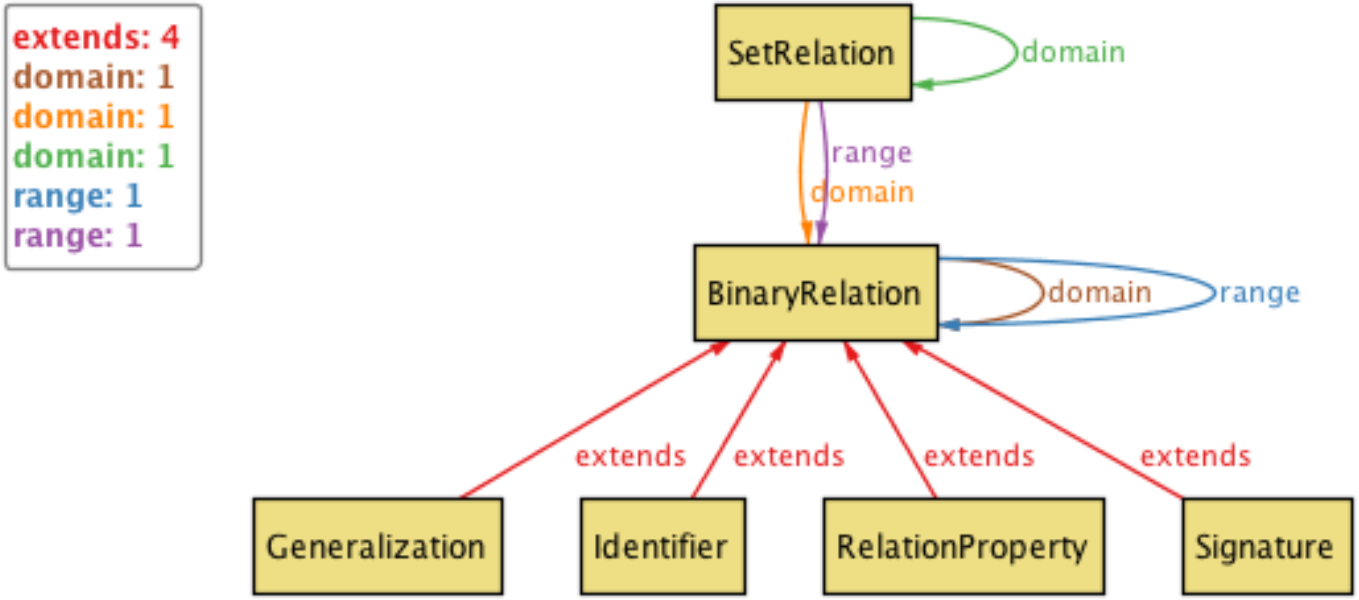
- **M4 library**

```
// M4 library

abstract sig Identifier extends BinaryRelation {}       { domain = this         && range = this }
abstract sig Signature  extends BinaryRelation {}       { domain in Signature   && range in Identifier }
abstract sig RelationProperty extends BinaryRelation {} { domain in Signature   && range in Signature }
abstract sig MetaclassKind extends SetRelation {}       { domain = this         && range in Signature }
abstract sig Generalization extends BinaryRelation {}   { domain in Signature   && range in Signature
  one mk:MetaclassKind | domain + range in mk.@range
}

fact { no disj mk1,mk2:MetaclassKind | mk1.range = mk2.range }
fact { all s:Signature { s.domain = s }}

fun Signature::getIdentifier[] : one Identifier { this.range }
fun Signature::getRelationProperties[] : set RelationProperty { this.~(RelationProperty<:@domain) }

fun Signature::getGeneralizations[] : set Generalization { this.~(Generalization<:@domain) }
fun Signature::getAllGeneralizations[] : set Generalization { this.*~(Generalization<:@domain) }
fun Signature::getGeneralizationAncestors[] : set Signature { this.~(Generalization<:@domain).@range }
fun Signature::getAllGeneralizationAncestors[] : set Signature { this.*~(Generalization<:@domain).@range }

fun Signature::getSpecializations[] : set Generalization { this.~(Generalization<:@range) }
fun Signature::getAllSpecializations[] : set Generalization { this.*~(Generalization<:@range) }
fun Signature::getSpecializationDescendents[] : set Signature { this.~(Generalization<:@range).@domain }
fun Signature::getAllSpecializationDescendents[] : set Signature { this.*~(Generalization<:@range).@domain }
```

National Aeronautics and Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

OWL2 & MOF Reconciled with Alloy: M4 MOF Metamodel



- **This M4 is 100% Reflexive (everything is a relation)**
  - **This metamodel preserves Alloy's reflexive semantics**
- **We can construct MOF cleanly as an M3 metamodel**
  - **M3 specifies "instances" of the M4 relations**

OWL2 & MOF Reconciled with Alloy: CMOF as an M3 metamodel (1/4)

Reconstructing EncapsulatedClassifier (more cleanly)

```
module M3
open M4

// ============ ClassifierType ============

one sig ClassifierTypeKind  extends M4/MetaclassKind{}{ range = ClassifierType }

sig ClassifierType extends M4/Signature {}{
  some getProperties[] implies getProperties[].getOwner[] = this
}

sig ClassifierType_properties extends M4/RelationProperty {}{ domain in ClassifierType && range in AssociationEndProperty }

fun ClassifierType::getProperties[] : set AssociationEndProperty    { this.~(ClassifierType_properties<:@domain).@range }
```

National Aeronautics and Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

OWL2 & MOF Reconciled with Alloy: CMOF as an M3 metamodel (2/4)


```
// ============ Association ============

one sig AssociationKind             extends M4/MetaclassKind{}{ range = Association }

sig Association extends M4/Signature {}{
  getAssociationEnds[] = getAssociationDomain[] + getAssociationRange[]
  this = getAssociationEnds[].getAssociation[]
  getAssociationEndTypes[] = getAssociationEnds[].getType[]
}

sig Association_endTypes extends M4/RelationProperty {} { domain in Association && range in ClassifierType }
sig Association_ends extends M4/RelationProperty {}     { domain in Association && range in AssociationEndProperty }
sig Association_domain extends M4/RelationProperty {}   { domain in Association && range in AssociationEndProperty }
sig Association_range extends M4/RelationProperty {}    { domain in Association && range in AssociationEndProperty }

fact {
  no disj r1,r2:Association_endTypes { r1.domain = r2.domain && r1.range = r2.range }
  no disj r1,r2:Association_ends { r1.domain = r2.domain && r1.range = r2.range }
  no disj r1,r2:Association_domain { r1.domain = r2.domain }
  no disj r1,r2:Association_range { r1.domain = r2.domain }
  all a:Association {
    some disj t1,t2:Association_endTypes | t1.domain = a && t2.domain = a
    some disj e1,e2:Association_ends | e1.domain = a && e2.domain = a
    one d:Association_domain | d.domain = a
    one r:Association_range | r.domain = a
}}

fun Association::getAssociationEndTypes[]   : some ClassifierType         { this.~(Association_endTypes<:@domain).@range }
fun Association::getAssociationEnds[]       : some AssociationEndProperty { this.~(Association_ends<:@domain).@range }
fun Association::getAssociationDomain[]     : one AssociationEndProperty  { this.~(Association_domain<:@domain).@range }
fun Association::getAssociationRange[]      : one AssociationEndProperty  { this.~(Association_range<:@domain).@range }
```

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

OWL2 & MOF Reconciled with Alloy: CMOF as an M3 metamodel (3/4)


```
// ============ AssociationEndProperty ============

one sig AssociationEndPropertyKind  extends M4/MetaclassKind{}{ range = AssociationEndProperty }

sig AssociationEndProperty extends M4/Signature{}{
  getOpposite[] = getAssociation[].getAssociationEnds[] - this
  getOwner[] = getAssociation[] implies this not in getOpposite[].getType[].getProperties[]
  getOwner[] != getAssociation[] implies this in getOpposite[].getType[].getProperties[]
}

sig AssociationEndProperty_association extends M4/RelationProperty {} { domain in AssociationEndProperty && range in Association }
sig AssociationEndProperty_opposite extends M4/RelationProperty {}    { domain in AssociationEndProperty && range in AssociationEndProperty }
sig AssociationEndProperty_owner extends M4/RelationProperty {}       { domain in AssociationEndProperty && range in Association + ClassifierType }
sig AssociationEndProperty_type extends M4/RelationProperty {}        { domain in AssociationEndProperty && range in ClassifierType }

fact {
  no disj r1,r2:AssociationEndProperty_association  { r1.domain = r2.domain }
  no disj r1,r2:AssociationEndProperty_opposite     { r1.domain = r2.domain }
  no disj r1,r2:AssociationEndProperty_owner        { r1.domain = r2.domain }
  no disj r1,r2:AssociationEndProperty_type         { r1.domain = r2.domain }

  all aep:AssociationEndProperty {
    one a:AssociationEndProperty_association | a.domain = aep
    one p:AssociationEndProperty_opposite | p.domain = aep
    one o:AssociationEndProperty_owner | o.domain = aep
    one t:AssociationEndProperty_type | t.domain = aep
}}

fun AssociationEndProperty::getAssociation[]    : one Association                { this.~(AssociationEndProperty_association<:@domain).@range }
fun AssociationEndProperty::getOpposite[]       : one AssociationEndProperty     { this.~(AssociationEndProperty_opposite<:@domain).@range }
fun AssociationEndProperty::getOwner[]          : one Association + ClassifierType { this.~(AssociationEndProperty_owner<:@domain).@range }
fun AssociationEndProperty::getType[]           : one ClassifierType             { this.~(AssociationEndProperty_type<:@domain).@range }
```

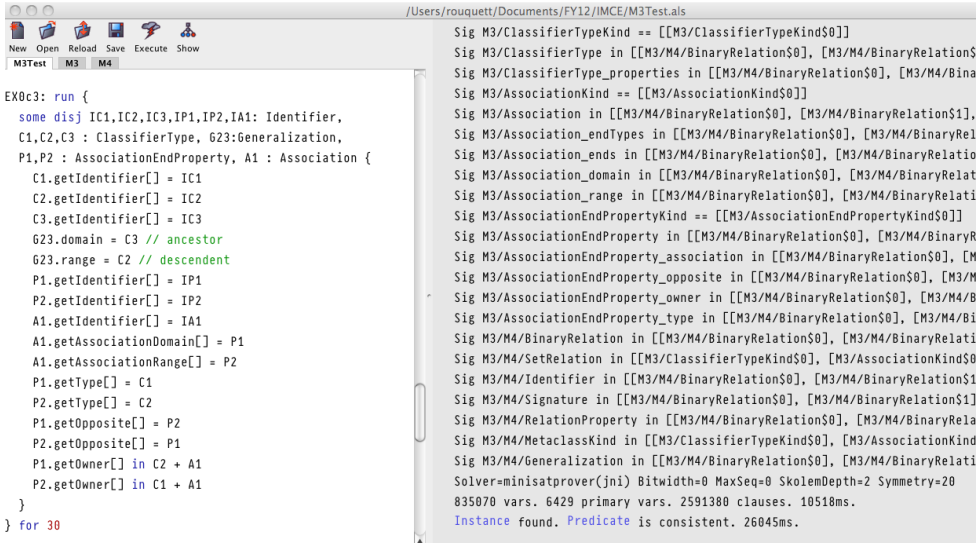
National Aeronautics and Space Administration

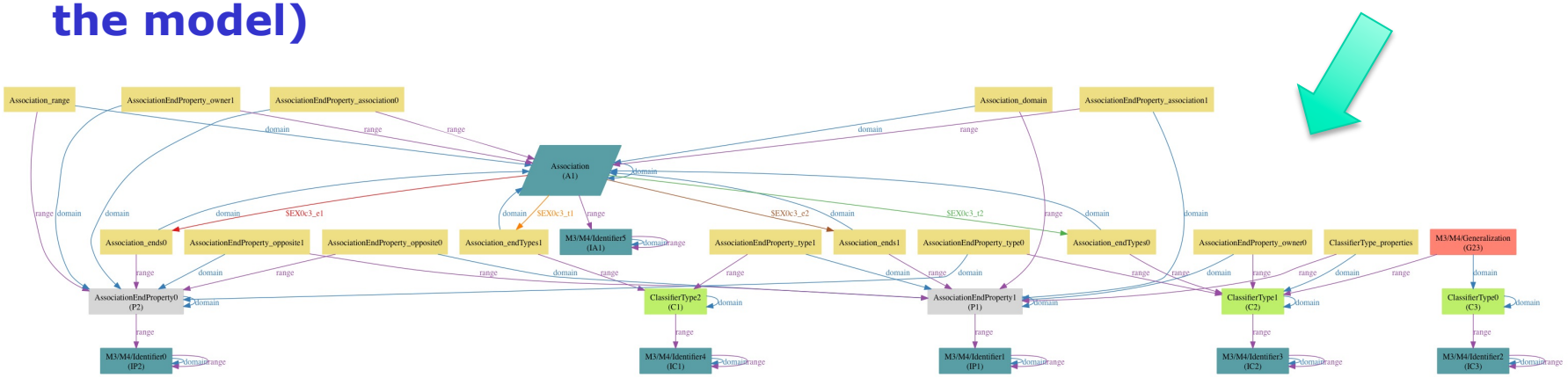
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

OWL2 & MOF Reconciled with Alloy: CMOF as an M3 metamodel (4/4)

Alloy Q: Is there a logically consistent model with:

- **1 Association**
- **3 Classes**
- **1 Generalization**
- **2 Properties**
- **such that… (constraints describing the desired structure of the model)**





Conclusion

- **The Alloy M4/M3 prototype confirms that the proposed fix for CMOF's Element/Link metamodel is technically worthy of consideration**
- **This prototype reflects the key principles used in JPL's OWL2/MOF2 integration infrastructure**
- **The combination of the formal verification with Alloy & the practical evidence from the OWL2/MOF2 integration suggests that it is possible to fix CMOF at the OMG**
- **Next Step**
  - **Discuss the revised CMOF M4/M3 construction in the MOF/ SMOF RTFs**