Model checking manual

How to use and write a model checker with Kermeta

Didier Vojtisek
INRIA

Abstract

This document presents the different concerns about model checking with Kermeta.

Not all aspects of writing or integrating are presented as they may be specific to the user choice or requiring some additional development that aren't available at the time of writing of this document, however the document try to list these potential usage.

The various sample presented in this document are extracted from the model checker written in the context of the Mopcom SoC project.

The document also details how to prepare an uml profile for being used in a checker. In this context, it presents how to optimize the navigation in the uml model thanks to kermeta aspects.

The document is organized as follow. The first chapter explains the base concepts used by a model checker including some specificities of Kermeta. The second chapter show a typical use of a model checker in the end user envrionnment. The third chapter provides some guidelines for writing a model checker. This chapter also presents some limitations when writing a model checker and various leads to get rid of them.

Published Build date: 3-November-2010
$LastChangedDate:: 2010-06-02 10:42:20#$

List of Figures

CHAPTER 1

Base concepts

The global goal of a model checker is to verify that a guiven model is valid for a given intend. Running a check on a model helps the user in detecting an invalid model in the early steps of a modeling process.

1.1. Constraints

The core of a model checker is to verify constraints. A constraint is a condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element.

To be able to build tool that use these constraints, they must be expressed using a machine readable language like OCL of Kermeta.

The language standardized by the OMG is OCL (Object Constraint Language). Kermeta can also natively express the same constraints. In order to use Kermeta runtime (interpreter or java compiler).

Since OCL behavior is a subset of Kermeta behavior, there is a transformation that allows to convert OCL constraint into kermeta ones, so the constraint can be used in any kermeta program (for example in a model checker, but also in other kind of application)

Note

In kermeta 1.3.2, we still need to manually do this conversion if we want to reuse OCL constraints in a kermeta program. However we plan to offer the possibility to directly include the OCL file in a kermeta program (ex: require "my.ocl") so the integration will be smoother.

1.2. Invariant constraints used to complement the metamodel structure

By default, the structure of a metamodel will help a given editor to build a conformant model.

Typically, if you take uml metamodel, the tree editor is smart enough to not propose to add a Package into a Class, but will propose only concepts that can fit in it, for example an Operation.

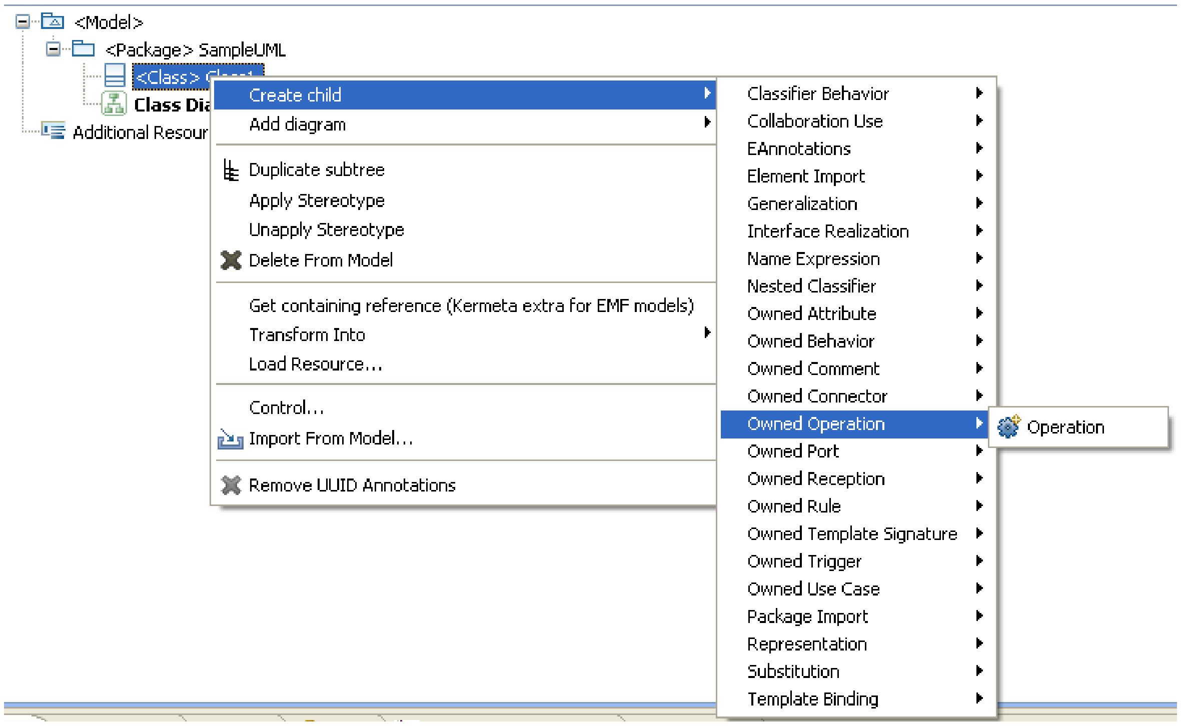


Figure 1.1. Restriction of the proposal for model element creation driven by the metamodel structure

However, some constraints are difficult or impossible to capture using only the metamodel structure.

For example UML metamodel specification is complemented with a lot of constraints, typically expressed in OCL.

Expressed in natural language : Generalization hierarchies must be directed and acyclical. A classifier can not be both a transitively general and transitively specific classifier of the same classifier.

Expressed in OCL :

```
not self.allParents()->includes(self)
```

Example 1.1. Sample of constraint in UMLmetamodel on Classifier (excerpt form UML specification)

A typical translation of this contraint in kermeta will be : (TODO)

1.3. Invariant constraints used to restrict a metamodel for a given intend

In order to implement a given process, it is sometime necessary to add some more constraints on a metamodel.

This will help to capture the intend behind the use of the given metamodel.

For example on a large metamodel like UML, we may want to use only a subset of it and even this subset can be used following some design rules. These addtional rules will help to specify the intend of the model, so it can be processed in a given way.

For example in a top down process, each step of the process may be expressed using UML, but at each step we add some more information and precision (typical MDA process) Even if each step use a valid UML model (regarding its structure and the standard constraints), a tool designed to work on a UML model of a given step might not work with an UML model of another process step.

In order to help the user detect the inconstancy, the basic idea is then to provide a set of invariant that are specific to a given step.

CHAPTER 2

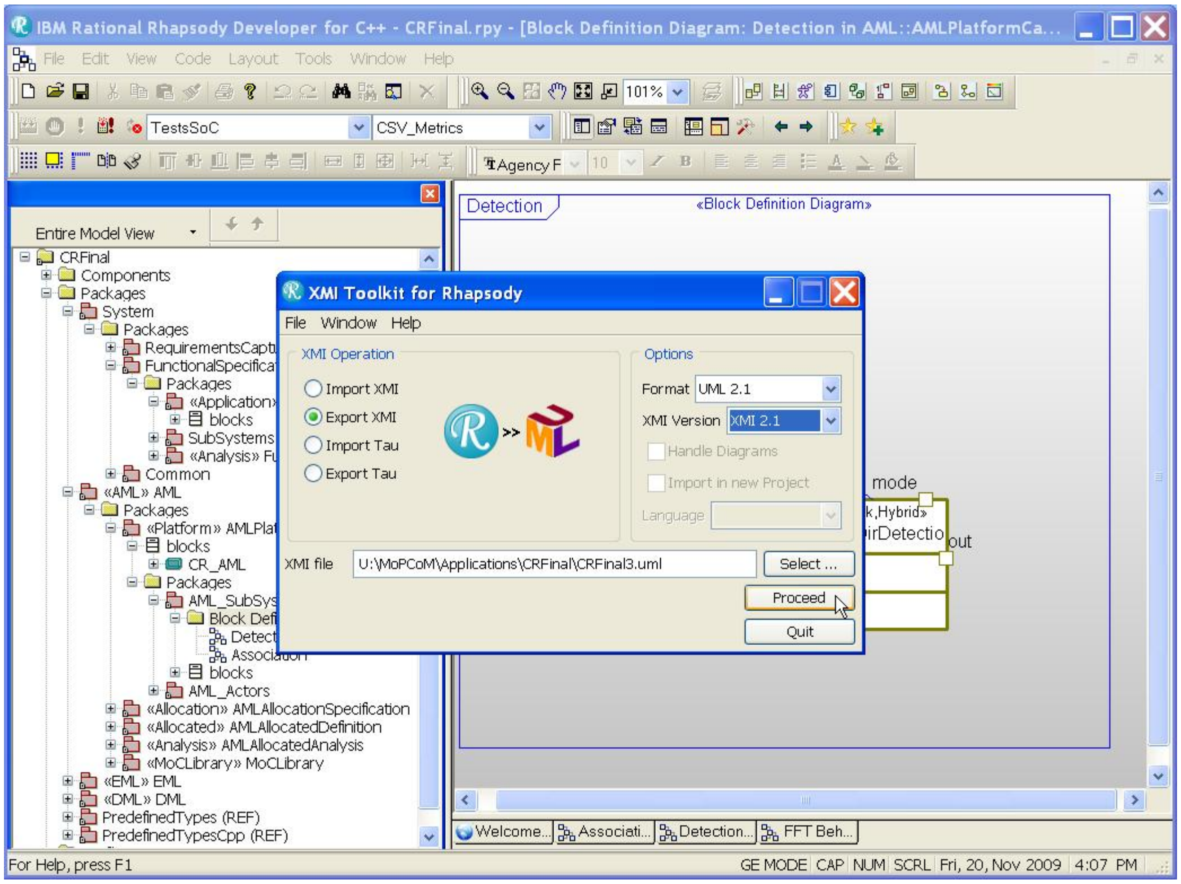
Using a model checker written in Kermeta

This chapter presents various way to use a model checker from the point of view of the end user.

2.1. Get the model to check in the appropriate format

A model checker written in kermeta is dependent on its EMF (Eclipse Modeling Framework) implementation. It is designed to work with a specific metamodel.

If the model to check isn't in the correct format (for example in specific modeler format like Rhapsody) it is necessary to translate it (export) to the checker supported format.



( ) Import XMI [✓ Export XMI] ( ) Import Tau ( ) Export Tau

Format: UML 2.1

XMI Version: XMI 2.1

XMI file: U:\MoPCoM\Applications\CRFinal\CRFinal3.uml

Figure 2.1. Example : Export from Rhapsody modeler to the XMI format supported by EMF

2.2. Integration in a graphical user interface

Typically, the model checker can be integrated in the end user graphical interface. For example, in Mopcom project we have provided some popup actions for uml file resources, that allows to launch the various check on uml models.

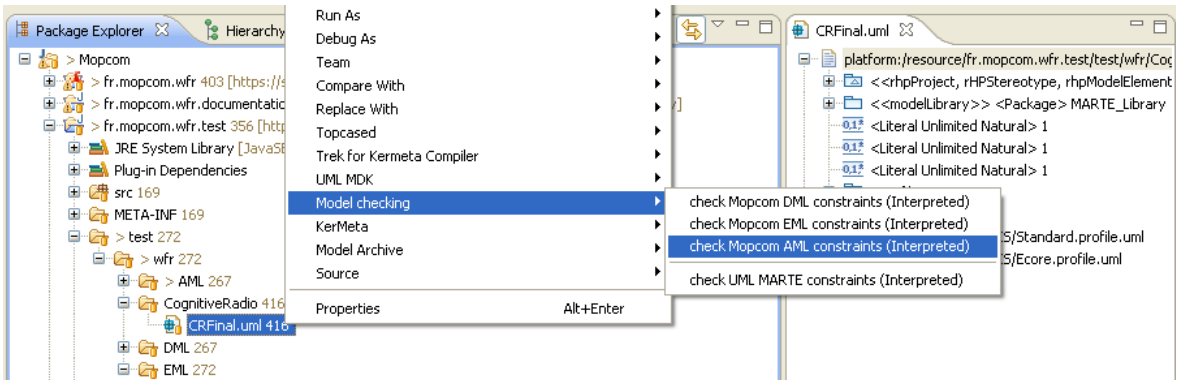

Package Explorer
Hierarchy
> Mopcom
> fr.mopcom.wfr 403 [https://s
> fr.mopcom.wfr.documentatic
> fr.mopcom.wfr.test 356 [http
JRE System Library [JavaSE
Plug-in Dependencies
src 169
META-INF 169
> test 272
> wfr 272
> AML 267
CognitiveRadio 416
CRFinal.uml 416
DML 267
EML 272
Run As
Debug As
Team
Compare With
Replace With
Topcased
Trek for Kermeta Compiler
UML MDK
Model checking
KerMeta
Model Archive
Source
Properties Alt+Enter
check Mopcom DML constraints (Interpreted)
check Mopcom EML constraints (Interpreted)
check Mopcom AML constraints (Interpreted)
check UML MARTE constraints (Interpreted)
CRFinal.uml
platform:/resource/fr.mopcom.wfr.test/test/wfr/Cog
<<rhpProject, rHPStereotype, rhpModelElement
<<modelLibrary>> <Package> MARTE_Library
<Literal Unlimited Natural> 1
<Literal Unlimited Natural> 1
<Literal Unlimited Natural> 1
S/Standard.profile.uml
S/Ecore.profile.uml

Figure 2.2. Integration example of Mopcom model checkers using popup menu on uml files

Tip

You can find the procedure to provide such popup menu thet runs a kermeta program and the typical associated java code is presented in the logo tutorial (TODO add a link to the online version)

More advanced GUI integration are possible depending on the user integrated environment. For example, we may use the ability of eclipse to detect file changes to automatically run the check and report the failed invariant in the problem view. However, automatic check must be considered with care, since the user may need to have an incorrect model during some phase of his modeling process. A good practice would be to always propose to disable automatic check.

As the model checkers are sometime specific to a given development step in a process, we can also imagine to customize the user environment to propose and activate only the relevant checkers for the current process step.

2.3. Run the checker as a standalone application

Like any kermeta program, a model checker written in Kermeta can be run as a standalone application, for example on the command line. However, in this case, the main operation of the checker must ensure to report the failed invariants in a textual way.

For more information about running a kermeta application from the command line, please refer to the kermeta UI user guide http://www.kermeta.org/documents/ui_user_guide/

CHAPTER 3

Writing a checker

This section presents how to write the invariant constraints and build a model checker from them.

Using Kermeta, checking a model is done in 3 steps :

- Declare which concept will be know and checked. Ie. the metaclasses know by your kermeta program.
- Retreive the model to check. (typically with a resource load)
- for all the root elements of the resource call the checkAllInvariant to recursively verify the model by checking the constraints that apply to a given element.

  Optionnally, we may want to verify only a subset of the model by selecting.
- Reports the elements that violate an invariant to the user.

  Ideally, with a user friendly message or interface.

3.1. Declaring Metaclasse

In Kermeta, you need to declare which metaclasses are known in your Kermeta program. This is acheived by using the `require` keyword which acts like an import. Then all class definition that are "required" will be available and known by your program. Otherwise you won't be able to load the model Ie. you can create an instance only of classes for which you know the definition !

3.1.1. General case

For checking your own metamodel, this is usually simple as you only need to add something like :

```
require "yourmetamodel.ecore"
```

You can directly require the ecore file because, if you need to have an extended version of a ClassDefinition, you can simply augment it using kermeta aspects.

Tip

If your ecore is deployed as a plugin you can also require the nsUri of the root package of your ecore inn ordr to get the version in EMF memory (in its registry). For ex:

```
require "http://www.eclipse.org/emf/2002/Ecore"
```

will get the version in memory, whereas

```
require "platform:/plugin/org.eclipse.emf/ecore/model/Ecore.ecore"
```

will get the file in the plugin org.eclipse.emf.ecore. Also note that these two require are incompatible in the same kermeta program since they connect to two similar but potentially different version of the same ClassDefinitions.

3.1.2. ClassDefintion of UML Profiles

In UML profiles, the stereotypes are a bit special because declaring a new stereotype definition is equivalent to declaring a new ClassDefinition that connects to the UML element (a uml::CLassifier, a uml::Package, a uml::LifeLine, etc).

The following figures show how a typical profiles is represented in a uml modeller (ie. what a uml end user see in his modeller) and what is actually in memory at the instance level (ie. at the metamodel level point of view, what is manipulated by a model transformation or model checker tool)

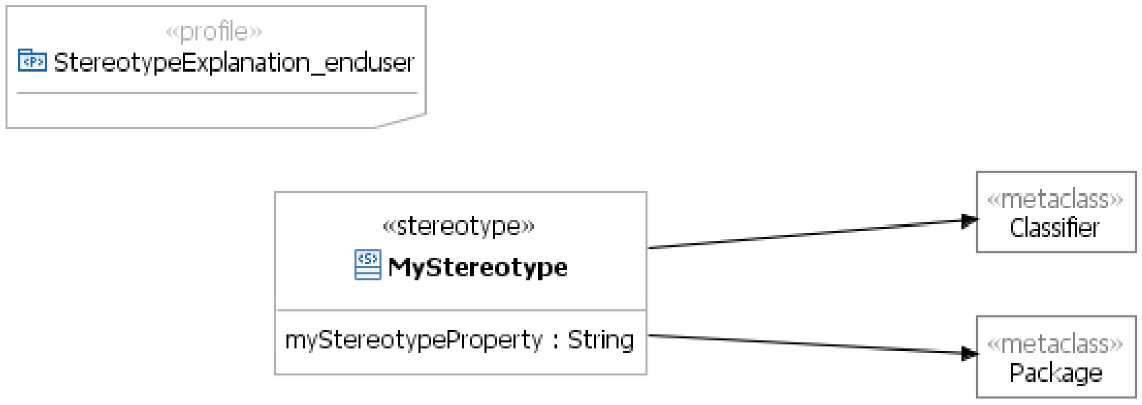
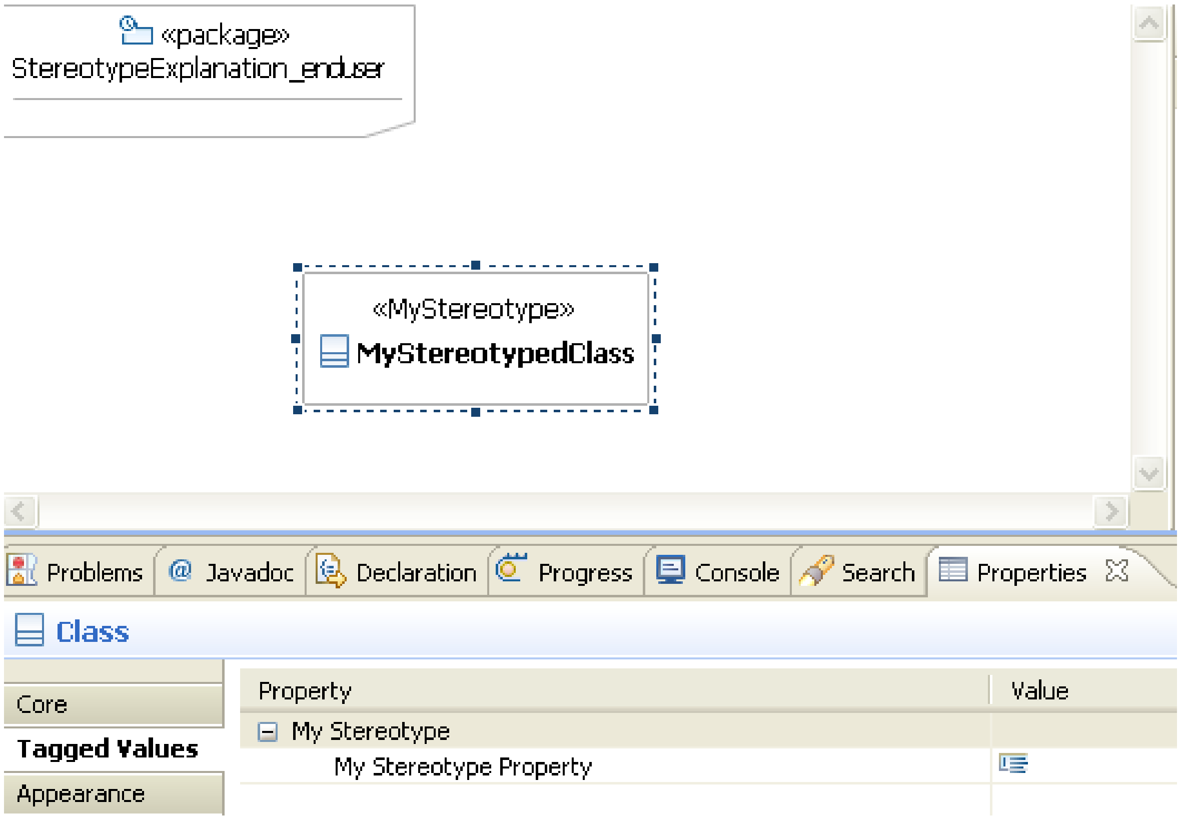


Figure 3.1. Stereotype declaration and instance as seen in a classical uml modeller



Property	Value
My Stereotype	
My Stereotype Property	

Figure 3.2. Stereotype use in a classical uml modeller

You can notice in (Figure 3.2, “Stereotype use in a classical uml modeller”) that the use of the stereotype is presented using some custom presentation, here it use the default presentation: name of the applied stereotype between double lower and greater, the stereotype atributes appears in a special tab in the property view.

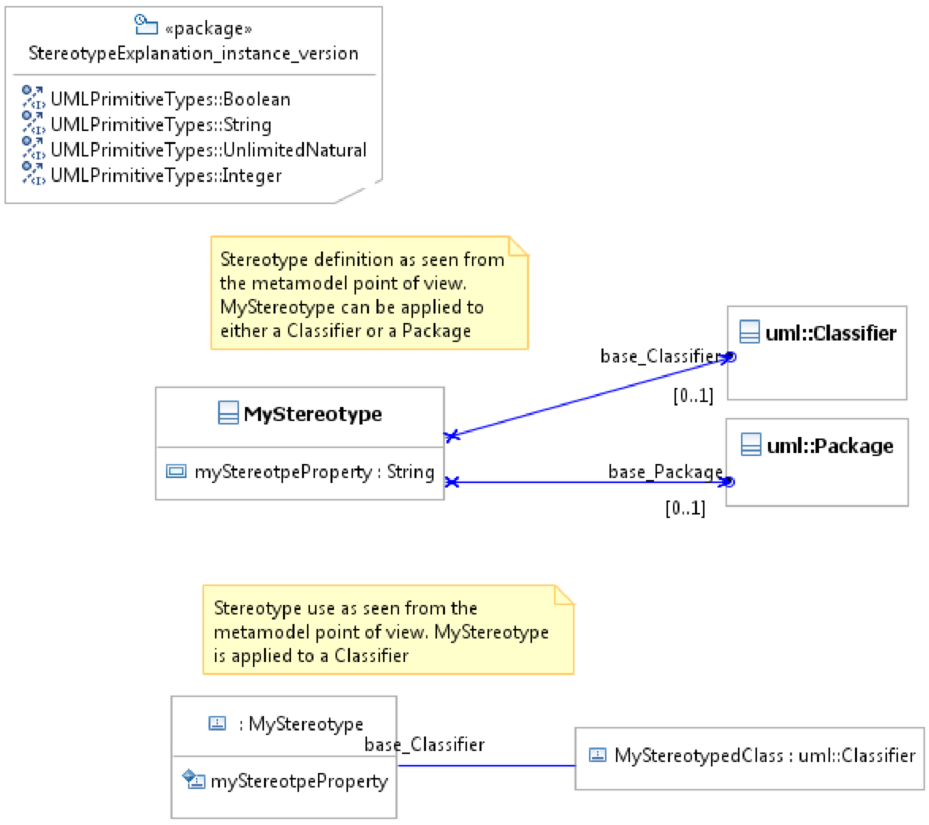

«package»
StereotypeExplanation_instance_version
UMLPrimitiveTypes::Boolean
UMLPrimitiveTypes::String
UMLPrimitiveTypes::UnlimitedNatural
UMLPrimitiveTypes::Integer
Stereotype definition as seen from the metamodel point of view. MyStereotype can be applied to either a Classifier or a Package
uml::Classifier
base_Classifier
[0..1]
MyStereotype
myStereotpeProperty : String
uml::Package
base_Package
[0..1]
Stereotype use as seen from the metamodel point of view. MyStereotype is applied to a Classifier
: MyStereotype
base_Classifier
myStereotpeProperty
MyStereotypedClass : uml::Classifier

Figure 3.3. Stereotype declaration and instance as seen from the metamodel point of view (ie. in an "ecore" modeller)

Note

This instance representation isn't specific to Kermeta, all modelling tools (including editors, model transformation tools, ...) use this representation internally even if they provide a simplified graphical representation.

You can notice several points :

- the stereotype object isn't contained by an uml element. Actually, they are stored at the root of the xmi file (usually side by sides with the root uml::Model element that contain the uml model)
- this link is navigable only in one direction, from the stereotype to the uml element.

Actually, the stereotype acts like a kind of decoration on top of the uml element, this was a requiremnt in uml specification in order to be able to load a uml model even if a specific tool don't know how to deal with a specific profile, the underlying uml model is still valid and can be processed.

This has some impacts on navigating in the model. For example, if from an uml element you want to know if it exists a stereotype that "decorate" the uml element, you'll have to navigate up to the resource [1] and then search within all the ob ject there if there is one that have a `base_XXX` property that points to the uml object

...

```
operation getClassifierStereotypes(aClassifier : uml::Classifier) : kermeta::standard::Sequence<kermeta::standard::Object> is do
  var containingResource : kermeta::persistence::Resource init aClassifier.containingResource❶
  result := containingResource.select{ o |
    var base_ClassifierProperty : kermeta::language::structure::Property
    var classDefinition : kermeta::language::structure::ClassDefinition
    classDefinition ?= o.getMetaClass.typeDefinition
    base_ClassifierProperty := classDefinition.allAttribute.detect { o | o.name.equals( "base_Classifier" ) } ❷
    o.get(base_ClassifierProperty) == aClassifier❸
  }
end
```

❶ We have to look into the resource, in this case we guess (but we may be wrong!) that the stereotype is contained by the same resource (xmi file) as the uml element

❷ since we only know that the stereotype has a property named "base_Classifier" we have to use reflexivity

❸ If true then we have a stereotype of `aClassifier`

Example 3.1. Trivial but inefficient navigation in uml profiles

This is quite inefficient even on such simple request. A traditionnal approach would have been to use a cache (for exemple in a Hashtable), but even that isn't really satisfying. We will see in section Constraints on UML profiles how we can simplify the navigation using kermeta aspects.

3.2. Retrieving the model to check (Model load)

The goal of this step is to get the model you want to check. It can be coupled to another process or model transformation written in Kermeta, however in most case, a simple resource load is usually enough to build a standalone checker.

3.2.1. Model load general case

As of Kermeta 1.3.2, loading a model is done using two concepts : a Resource which usually represent a file, and a Repository which represent a a context for a set of Resources.

TODO kermeta code sample

```
var inputRepository : kermeta::persistence::EMFRepository init kermeta::persistence::EMFRepository.new ❶
var inputResource : kermeta::persistence::EMFResource
inputResource ?= inputRepository.createUMLResource(fileToLoad, "platform:/plugin/org.company.yourmetamodel/model/yourmetamod
inputResource.load()❷
```

❶ use a new repository for all the resources to load

❷ Load the resource, if it depends on other resources they will be loaded too in the same Repository

[1]this will work only if you suppose that the stereotype is contained by the same resource as the uml object. Otherwise, you'll also have to search within all the resources known by the containing Repository !

After these instructions, the inputResource can be navigated to retreive all the model elements. This resource is a collection that contains all the element at the root of the loaded file.

```
inputResource.one ❶
inputResource.each{ rootElement | // do something on each element at the root of the Resource }❷
```

❶ For example, if you know that there is always one root element in your model, navigate by getting only `one`.

❷ or process the collection of root elements for example with a `each` (or you can also use `select`, `collect`, etc operations availables on collections)

Example 3.2. Sample of load UML resource with profile

3.2.2. Loading UML model with profiles

TODO enabling profile, special UML resource

```
var inputRepository : kermeta::persistence::EMFRepository init kermeta::persistence::EMFRepository.new ❶
inputRepository.registerEcoreFile("platform:/plugin/org.eclipse.uml2.uml/model/UML.ecore") ❷
inputRepository.registerEcoreFile("platform:/plugin/org.eclipse.uml2.uml/model/UML_21.ecore")
inputRepository.ignoreLoadErrorUnknownProperty := true ❸
inputRepository.ignoreLoadErrorUnknownMetaclass := true
var inputResource : kermeta::persistence::EMFResource
inputResource ?= inputRepository.createUMLResource(fileToLoad, "platform:/plugin/org.eclipse.uml2.uml/model/UML.ecore")
inputResource.load()
```

❶ use a new repository for all the resources to load

❷ Optionnal, helps to find load UML model when not run in eclipse. May also help in compiled mode

❸ Optional, These options help to load UML model for wich we don't have translated all stereotype into proper ecore classdefinition. Setting these option to true will simply ignore the model elements that cannot be loaded.

After these instructions, the inputResource can be navigated to retreive all the model elements. This resource is a collection that contains all the element at the root of the loaded file.

Example 3.3. Sample of load UML resource with profile

TODO show a figure explaining the location of the stereotypes at the root of the file

3.3. Writing constraint

This second step is the core part of the checker. It consist in writing the invariant constraints associated to the metaclasses of the model we want to check.

Actually, the context of the invariant (ie. the metaclass containing the invariant) constitute a kind of filter for the invariants. On a given element, only invariants that apply to this element will be checked. This also constitutes the basis for the organisation of the constraints because they will natively be displayed in their containing class.

For every combinaison of metamodel and intent, we need to write a set of constraints.

3.3.1. General case

As stated above, the invariant naturally applies to a metaclass. Kermeta aspect allows to reopen a metaclass definition, then adding an invariant to an existing metaclass is quite simple.

```
package uml;   // name of the package for these definition, the same as the one we want to aspectize


require kermeta
        // require the UML2 metamodel
require "http://www.eclipse.org/uml2/3.0.0/UML"

aspect class Classifier    // use aspect keyword in order to reopen the metaclass uml::Classifier
{
   inv inheritanceMustBeDirectedAndAcyclical is do
      not self.allParents().includes(self)  // the expression in the invariant must return false if the invariant is violated
   end
}
```

This sample adds an invariant to the metaclass uml::Classifier that checks that the inheritance is directed and doesn't have a cycle.

Example 3.4. Example of invariant definition on UML metamodel

Warning

Be careful, in this sample, the operation allParents is abstract in the metamodel. You must ensure to provide an implementation for it otherwise you'll get a NotImplementedException.

In that sample, you can use the uml2 MDK and require the following file somewhere in your kermeta program:

```
require "platform:/plugin/org.kermeta.uml2/src/kermeta/uml2_behavior.kmt"
```

TODO : Implement this allParent() in uml MDK!!!

A similar problem is when using derived properties for which you need to provide a behavior.

Note

Please note that the invariant name (inheritanceMustBeDirectedAndAcyclical in the above sample) may not be unique, however a good practice is to use a name as explicit as possible

3.3.2. Constraints on UML profiles

When using UML profiles you may have some specificties.

TODO enabling profile for adding invariant on it

TODO

Opposite on stereotype

TODO more generally use aspects to simplify a check algorithm (complexity and/or readability)

3.3.3. Model checker product lines (ie. Organizing invariant constraints)

TODO use kermeta aspect to select the set of constraints for a given intent, typical folder organisation in Mopcom

3.4. Launching the check and giving feedback on failed constraints

Kermeta offers several operation that allows to

3.4.1. General case

TODO provide the natural language version in the documentation associated to the invariant

TODO report only the first one or all

3.4.2. Feedback for constraints on UML Profiles

3.5. Limitations and possible enhancements

This chapter list some issues that my complexify the creation of a model checker. If possible, it also presents some cues towards resolving these problems.

> **Note**
>
> As this section presents some innovative ideas, if you have implemented some of them, please let us know and consider contributing your work to this document.

3.5.1. Load of large model

TODO

Leads : Load on demand/lazy loading

Leads : selective load

3.5.2. Many or complex constraints

TODO

Leads : compilation

Leads : incremental check

3.5.3. Test of model checker themselves

TODO

Leads : Provides big representative sample models, then modify them to make them invalid; use getViolated-Constraints and add assert for all the contraints you want to see

Leads : Generate test models

3.5.4. Advanced Model checker product line

TODO

Leads : use model type + aspect to adapt compatible metamodels and then reuse existing invariants

Leads :

3.5.5. Automatic Generation of constraint from a set of sample model

TODO

Leads : from a set of sample model representative of an intent, we can generate the constraints that restrict the metamodel to only the used elements.

Leads :