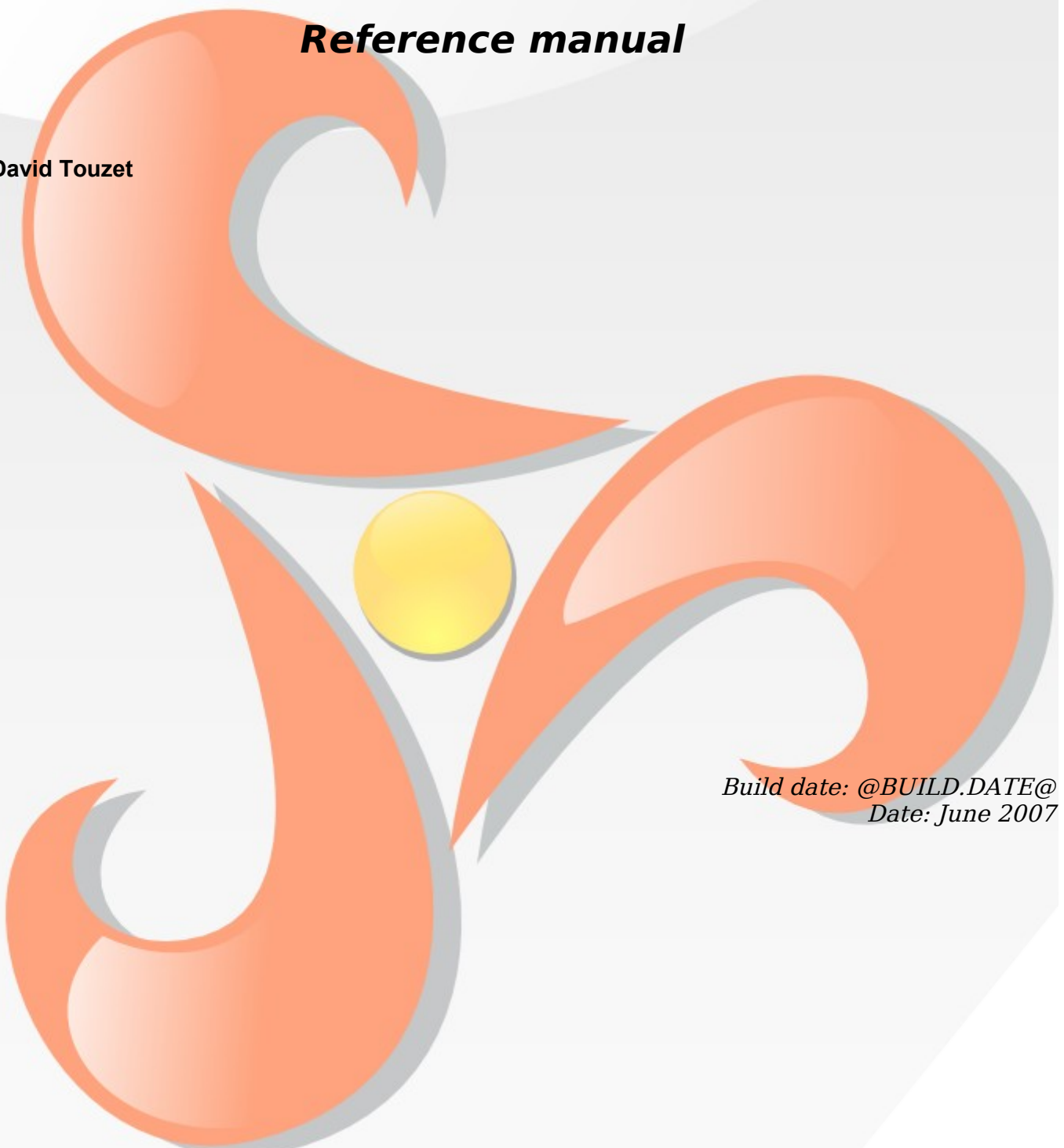INRIA

KerMeta Transformation Rules

Reference manual

David Touzet

Metadata

The metadata gather information allowing a better spreading of the informations through various media: databases, cd, internet and intranet repositories...

The following information will be considered as metadata.

Document information

Title	KerMeta Transformation Rules
Subtitle	Reference manual
Volume number	
Series number	
Keywords	Transformation language, Kermeta
More information	Licence EPL
Document date	05/06/2007
Status	
Access conditions	
Access conditions revision Date	16/05/2005
Contract number	
ISRN	

Conference

Title	title
References	Reference
Sponsor	sponsor
Dates	
Address	

Authors

Author N°1

Firstname	David
Name	Touzet
Role	
Quality	
Corporation	INRIA
Corporation acronym	
Corporation division	Triskell
Address	dtouzet@irisa.fr

Author N°2

Firstname	
Name	
Role	
Quality	
Corporation	
Corporation acronym	
Corporation division	
Address	

Author N°3

Firstname	
Name	
Role	
Quality	
Corporation	
Corporation acronym	
Corporation division	
Address	

Author #4

Firstname	
Name	
Role	
Quality	
Corporation	
Corporation acronym	
Corporation division	
Address	

Corporate Authors

Corporate Author N°1

Corporation name	INRIA
Corporation acronym	
Corporation division	Triskell
Address	
Post office box	
ZIP code	
City	
State	France
Phone	
Fax	
E-mail	
Web site	http://www.inria.fr

Contract Sponsors

Contract sponsor N°1

Corporation name

Corporation acronym

Corporation division

Address

Post office box

ZIP code

City

State

Phone

Fax

E-mail

Web site

Organisme Commanditaire N°2

Corporation name

Corporation acronym

Corporation division

Address

Post office box

ZIP code

City

State

Phone

Fax

E-mail

Web site

Last modification : 2006-20-09 zdrey

Abstract

This document introduces the KTR transformation tool.

Preface

Table of contents

Table of contents

1   Introducing KTR

KerMeta Transformation Rule (KTR) is a declarative model transformation language. It has been built on top of the model-oriented KerMeta programming language.

2   KTR project

The KTR project is available on the KerMeta CVS, in the *ktr_projects* folder. The folder currently contains three KTR plugins:

- *fr.irisa.triskell.ktr.model*, the main KTR project containing the KerMeta code for the KTR transformation engine;
- *fr.irisa.triskell.ktr.model.edit*, generated using EMF;
- *fr.irisa.triskell.ktr.model.editor*, generated using EMF.

The main KTR plugin (*fr.irisa.triskell.ktr.model* plugin) is organized as follows:

- *src/kermeta/* folder contains all KTR KerMeta code. Subfolder *util* contains some helpers related to Ecore and KerMeta metamodels (should be respectively moved to the Ecore and KerMeta MDKs), whereas subfolder *ktr* contains KTR specific files.
- *src/ecore/* folder contains Ecore related resources. *KTR.ecore* metamodel has been generated from the *KTR.kmt* file. *nsuri* of the root *ktr* package has been changed to http://www.kermeta.org/Ktr. *KTR.genmodel* has been generated from the *KTR.ecore* file with two modifications: the field *model directory* (from the *Model* menu) is set to *fr.irisa.triskell.ktr.model/build/java/*, and the property *base package* of the *ktr* package is set to *fr.irisa.triskell*.
- *build/java/* folder contains the model java code generated from the *KTR.genmodel* file.
- *src/java* folder contains additional java code (currently only contains the KTRPlugin class).

Note that both *fr.irisa.triskell.ktr.model.edit* and *fr.irisa.triskell.ktr.model.editor* projects have been generated by means of the KTR *genmodel* without any additional modification.

3   Writing a KTR transformation

3.1   Introduction

A KTR transformation appears as a model that conforms to the KTR metamodel (defined in the *src/kermeta/ktr/KTR.kmt* and the corresponding *src/ecore/KTR.ecore* files). A KTR transformation is defined by means of the following concepts:

- InterpretationPatternModel: the KTR root element, that contains the KTR rules to be applied in an *ordered* list.
- InterpretationRule: the definition of a KTR rule. It has a *name*, a *source* (of RulePattern type) and a number of *productions* (of RuleProduction type). It can additionally define a number of *variables* of RuleVariable type. Finally, it has an optional *superRule*.
- RulePattern: this element allows to define the model elements that are matched by its containing rule. It has a *name*, a *sourceElement* (an Ecore class) that defines the type of the matched elements, an optional *sourceModel* that corresponds to the name of an input model (this can be used to restrict matching to the elements of a given input model), and a *condition* , expressed

as a KerMeta expression, that matched elements must validate.

- RuleProduction: this element allows to define which kind of target elements are allocated by the containing rule, and the way they are initialized. A RuleProduction has a *name,* a *targetElement* (an Ecore class) that defines the type of the element to be allocated, an optional *targetModel* that corresponds to the name of an output model (this can be used to specify to which output model the allocated element will be assigned), and an *initExpr*, expressed as a KerMeta expression, that defines the way the allocated element must be initialized.

- SingletonRuleProduction: this element is a subtype of RuleProduction. It has an *idExpr*, expressed as a KerMeta expression, that is used to uniquely identify an allocated singleton target model element.

- RuleVariable: this element allows factorizing computations while executing an interpretation rule. It makes it possible to assign into a variable the value of an expression that will be available for the duration of the rule execution. A RuleVariable has a *name*, an *initExpr* (corresponding to a KerMeta expression) specifying the way the value of the variable is computed, and a *varType* (pointing to an Ecore class) defining the expected type of the *initExpr*.

3.2 The *module* global variable

KTR relies on the model-oriented KerMeta language for the specification of conditional and initialization expressions. KTR defines a global variable, the *module* variable, that can be used in the scope of any of these expressions.

The global variable module provides access to two useful services:

- *allInstancesOf(kermeta::reflection::Class) : set<Object>*. This operation accepts a class as parameter and computes the set of all input model elements that are instance of this class. Note that returned value is a set of Objects, and that returned elements will have to be individually casted before being able to call any operation specific to the provided class.

- *resolve(inElt, ruleName, productioName) : Object*. This operation aims to provide access, from a given interpretation rule, to the output model elements that are generated by any interpretation rule from a given input model element. It accepts three parameters:
  - the input model element (*inElt*) of type Object;
  - the name of the targeted interpretation rule (*ruleName*);
  - the name of the targeted rule production (*productionName*) of the targeted rule.

If resolve succeeds, the operation return an Object corresponding to the searched output model element. Otherwise (in case the provided input model elements does not match the targeted interpretation rule), the operation returns *void*.

3.3 Writing an interpretation rule

A KTR interpretation rule allows a KTR developer to define 1) which output model elements will be produced when matching a given kind of input model elements that validate the specified condition, and 2) how these output model elements must be initialized. Each interpretation rule has a name that must be unique among the set of interpretation rules.

An interpretation rule is composed of a rule pattern, along with some optional rule variables and rule productions. These different contained elements are named, and it is assumed all these names are unique with respect to the different kinds of elements

(pattern, variables, and productions) contained by the rule.

3.3.1 Defining a RulePattern

A rule pattern must define the type of model elements that are matched by the rule by means of the *sourceElement* property. This property must point to an existing class of an input Ecore metamodel.

A pattern provides two ways for restricting the set of matched elements. The first filtering mechanism corresponds to the pattern *condition*. A condition must be a boolean KerMeta expression. Variables available in the scope of the condition are:

- the *module* global variable;
- the currently matched input model element, through the pattern name.

The second restricting facility is associated with the *sourceModel* field. This property enables to specify the name of the model (with respect to the model names specified in the input ModelItems, as exposed in Section 4) from which matched elements must come from. When no source model is specified, the rule will match input elements of the specified type from all declared input models.

3.3.2 Defining a RuleVariable

A interpretation rule may define a number of rule variables. These variables can be used to factorize some KerMeta expressions. Note that rule variables are re-computed each time an input model element is successfully matched by the interpretation rule (i.e. validate type, condition, and source model constraints). As a consequence, rule variables cannot be used in the scope of the condition expression of rule patterns.

A rule variable is characterized by its *initExpr* which corresponds to a KerMeta expression. The value of this KerMeta expression must be of the type specified by the *varType* property (*varType* points to a class of either an input or output Ecore metamodel). The KTR engine uses this information to dynamically check (at runtime) that the type of a computed init expression corresponds to the declared type of the variable (its *varType*).

Variables available in the scope of this init expression are:
- the *module* global variable;
- the currently matched input model element, through the name of the pattern of the containing rule.

Note that previously defined variables cannot be used for the specification of other variables.

3.3.3 Defining a RuleProduction

A rule production defines the type of the model element to be allocated when the pattern of the containing rule is successfully matched. This is achieved through the *targetElement* property, which must point to an existing class of an output Ecore metamodel.

Initialization of the allocated elements is specified by means of the KerMeta expression contained by the *initExpr* property. In the scope of this expression, the newly allocated model element must be referred to by means of the name of the rule production. Variables available in the scope of this init expression are:

- the *module* global variable;
- the currently matched input model element, through the name of the pattern of the containing rule;

- all the output model elements declared by the containing interpretation rule, through the name of their respective rule productions;
- all the rule variables declared in the scope of the containing rule, through their respective names.

Note that when specified, the *targetModel* property defines the target model to which the newly allocated model element will be assigned. This property is mandatory when the run interpretation has more than one output model (as defined by the output ModelItems presented in Section 4).

3.3.4   Defining a SingletonRuleProduction

A singleton rule production is a specific type of rule production. It allows developers to define a rule production ensuring that no other *equivalent* output model element will be assigned to the same output model. Equivalence between model element is here based on the value of an Id that is defined in the *idExpr* property (containing a KerMeta expression) of the singleton rule production. Variables available in the scope of this Id expression are:

- the *module* global variable;
- the currently matched input model element, through the name of the pattern of the containing rule;
- all the rule variables declared in the scope of the containing rule, through their respective names.

When using a singleton rule production, the KTR engine ensures that no duplicated elements of the same type, and whose Id expression resolves to the same value, will be added into the same output model from any singleton rule production.

Developers should pay attention to the fact that using one singleton rule production is not sufficient to ensure the uniqueness of the allocated elements. Therefore, using normal rule productions for generating the same kind of elements can raise the allocation of duplicated model elements.

3.3.5   Rule inheritance

The KTR engine provides support for rule inheritance: each interpretation rule can inherit from another interpretation rule. This rule inheritance mechanism allows developers to simply define refined rules:

- the condition of a sub rule corresponds to the conjunction of its own condition and the condition of its super rule;
- rule production expressions of a sub rule have access to rule variables and rule productions of its super rule;
- Id expressions of the singleton rule productions of a sub rule have access to the rule variables and the rule productions of its super rule.

Rule inheritance imposes some constraints over the rule pattern of sub rules:

- the name of the rule pattern of a sub rule must be the same than the one of its super rule;
- the type of the *sourceElement* of the rule pattern of a sub rule must be either the same or a subtype of the *sourceElement* of the rule pattern of its super rule.

3.4 Tips

Fully qualified names must be used to refer to classes and data types of input and output models.

Attributes, references, variables names that correspond to KerMeta keywords must be protected by means of the "~" character (as in a simple KerMeta program).

4 Running a KTR transformation

KTR transformations can only be launched from a programmatic way. This could be achieved by means of the API provided by the KTRInterface class (available at *platform:/plugin/fr.irisa.triskell.ktr.model/src/kermeta/ktr/KTRInterface.kmt*).

Before running a KTR transformation, make sure that metamodels of both transformation input and output models are imported (by means of *require* statements) by the KerMeta program that calls the KTR interface.

4.1 KTR programmatic interface

KTRInterface allows running a KTR transformation either in simple (*run* operation) or traceability (*runWithTrace* operation) mode. Both operations share a number of parameters:

- the URI of the KTR transformation to be run;
- the ModelItem sequence corresponding to input models;
- and the ModelItem sequence corresponding to output models.

The *runWithTrace* operation accepts an additional parameter that corresponds to the URI of the Trace model that is produced by the transformation.

ModelItem elements are used to fully describe the models handled by the transformation. Each ModelItem associates three model properties:

- *mdlURI*: the URI of the model (from which it can be loaded);
- *mMdlURI*: the URI of the corresponding metamodel (can be either a valid file path or a symbolic URI);
- *mdlName*: the name of the model (used to identify the model during the transformation). This optional value becomes mandatory in the following cases:
  - for input ModelItems for which model filtering is used during the transformation;
  - for all output ModelItems in case the KTR transformation has several output models.

It is up to the calling KerMeta program to correctly build the sequences of input and output ModelItems that are passed to the KTRInterface operations.

Note that the ModelItem definition has been included in the *KTRInterface.kmt* file as part of the *ktr::interface* package, so that programmatically calling a KTR transformation simply requires to include a single file.

4.2 KTR launching sample

This section illustrates the way the KTR interface should be used for programmatically calling KTR transformations. The provided code sample exemplifies the launching a KTR transformation having a single input model, and producing a couple of target models.

Being able to make use of the KTR interface facilities requires to first import the

KTRInterface definition (lines 2-3). KTR transformations can then be launched by means of an instance of the *KTRLauncher* class (line 6). Before running the transformation, the KTR launcher must be parameterized with both the lists of input and output models of the transformation.

```
1       // Import of KTRInterface definition
2       require "platform:/plugin/fr.irisa.triskell.ktr.model/src/kermeta/ktr/KTRInterface.kmt"
3       using ktr::interface
4
5       // Creation of the KTR transformation launcher
6       var launcher : KTRLauncher init KTRLauncher.new
7
8       // Parameterizing transformation's input model
9       var inMdls : seq ModelItem[1..*] init Sequence<ModelItem>.new
10      var inMdl : ModelItem init ModelItem.new
11      inMdl.mdlURI := inMdlUri
12      inMdl.mMdlURI := inMMdlUri
13      inMdl.mdlName := "in"
14      inMdls.add(inMdl)
15
16      // Parameterizing transformation's output models
17      var outMdls : seq ModelItem[1..*] init Sequence<ModelItem>.new
18      var outMdl1 : ModelItem init ModelItem.new
19      outMdl1.mdlURI := outMdlUri1
20      outMdl1.mMdlURI := outMMdlUri1
21      outMdl1.mdlName := "out1"
22      outMdls.add(outMdl1)
23
24      var outMdl2 : ModelItem init ModelItem.new
25      outMdl2.mdlURI := outMdlUri2
26      outMdl2.mMdlURI := outMMdlUri2
27      outMdl2.mdlName := "out2"
28      outMdls.add(outMdl2)
29
30      // Launching KTR transformation
31      launcher.run(ktrMdlUri, inMdls, outMdls)
```

Input and output models must be specified in two sequences of *ModelItem* elements (lines 9 and 17). ModelItems allow to define the properties of the models (either input or output) involved in the transformation to be run: their URI (*ModelItem.mdlUri*), the URI of their metamodel (*ModelItem.mMdlUri*), and the name by which they will be referred to in the scope of the transformation (*ModelItem.mdlName*).

5   KTR interpretation process

The KTR interpretation process can be decomposed into five successive steps (see the operation *execute* of the class *InterpretationPatternModel*):

- the pre processing pass aims to initialize all the computation variables embedded by the KTR model;
- the first interpretation pass aims at allocating all output model elements from input ones, and initializing them;
- the second interpretation pass aims at initializing output model elements that were not initialized during the first pass (because of unresolved references between output model elements at the initialization step);
- the result building pass aims to assign the model elements that have been allocated into their respective output models;
- the post processing pass aims to reset a number of computation properties for speeding up the saving of both output and trace models.

These different steps are detailed in the following subsections.

5.1 Pre processing

The aim of the pre processing step is to initialize all KTR model properties that are used for performing the interpretation. Pre processing code is specified in the *preProcessing* operation of class *InterpretationPatternModel*. It mainly consists in three different actions:

- global hashtables are initialized;
- KerMeta expressions are parsed so that their correctness can be checked. This is achieved by allocating and initializing a dynamic expression for each defined KerMeta expression;
- references to Ecore classes (used to specify the KTR model) are resolved into their corresponding KerMeta classes (used for performing the interpretation).

5.2 First interpretation pass

First interpretation pass tries to apply each rule having no super rule to the set of input model elements. Rules are considered in the order they are defined in the KTR model.

Applying an interpretation rule onto an input model element first consists in checking if this input model element validates the conditions specified by the rule pattern. In case these conditions are validated (type, condition and optional model filtering), the rule is executed, and its sub rules are applied to the same input model element, in the order they are defined in the KTR model.

Once its conditions are validated, the execution of a rule can be decomposed in four successive steps: initialization of the rule variables, allocation of the output model elements, optional allocation of the trace element that links the matched model element to the allocated ones, and initialization of the allocated output model elements.

As a first step, all variables declared for the rule are initialized by running their respective *initExpr*. The interpreter also gets all the variables inherited from the direct and indirect super rules of the current rule.

Second step of a rule execution is the allocation of the output model elements associated with the different rule productions. The interpreter allocates a new output model element for each basic rule productions (operation *allocate* of class *RuleProduction*). For SingletonRuleProductions, the interpreter first tries to find an already allocated model element that is equivalent to the element to be allocated before allocating a new one (operation *getOrAllocate* of class *SingletonRuleProduction*).

Last step of a rule execution is the initialization of the allocated output model elements. For this purpose, the interpreter has to get the output model elements context that is inherited from the super rules of the rule. Note that the initialization step (operation *initialize* of class *RuleProduction*) is skipped for singleton productions that did not allocate a new output model element.

Initialization of a model element allocated by a production rule consists in running the *initExpr* of the production rule. It may happen that this init expression contains references to model elements that have not been allocated yet by the interpreter (when calling the *module.resolve* operation). In such a case, the input model element and the production rule are cached so that this init step can be run again in the second interpretation pass (once all output model elements will be allocated).

5.3 Second interpretation pass

At the end of the first interpretation pass, a number of allocated output model

elements remains uninitialized because of unresolved references to output model elements that were not allocated yet. The second interpretation pass aims to initialize these output model elements.

To this end, the interpreter iterates over the set of cached couples of input element and rule production. For each entry of this cache, it rebuilds the full interpretation context (matched input model element, as well as the rule variables and the rule productions, including inherited ones) and relaunches the initialization operation from the cached production rule onto the associated cached input model element.

5.4 Result building

At this stage, the interpretation process is completed. The aim of this step is to sort the allocated output model elements in order to assign each of them to the appropriate output model.

5.5 Post processing

A post processing pass, specified in the *postProcessing* operation of class *InterpretationPatternModel*, has been defined for speeding up the saving of the output and trace models produced by the interpretation.

Due to KerMeta internal design issues, saving a model implies updating all the models contained by the repository of the model to be saved. Although the KTR model does not need to be saved, its content is consequently updated, including the contents of the hashtables properties and the structure of the parsed dynamic expressions (may be very long).

To avoid such unnecessary computations, the post processing pass reset all useless properties of the KTR model. This concretely means that:

- global hashtables are cleared;
- dynamic expressions are set to *void*;
- references to Ecore classes are also set to *void* to remove dependencies to Ecore metamodels, thus avoiding their contents to be updated along with the saved output models.

6 KTR future issues

Here is a list of possible future developments:

- declaration of global variables (visible from all elements of the transformation). Should be initialized before the first interpretation pass starts. This implies that their init expression will only be allowed to navigate input models.
- declaration of functions, thus enabling to factorize duplicated KerMeta code, used at different points of the transformation.
- definition of multiple rule productions, that enables to produce as many output model elements as the number of elements contained by a nary property of the matched input model element.
- Optimization of the result building pass – one possible way is to sort the generated output model elements according to their target model at the time they are allocated.

Appendices