AoUrnToRam

Developer's Guide

Author

Stéphane Leblanc

Version 1.0.0 (2012-04-30)

Table of Contents

Setting Up the AoUrnToRam Development Environment

Setup Eclipse

1. **Download the Eclipse Modeling Tools**
   Note: Use the 64-bit version if your CPU/OS supports it.
   Latest tested version: 3.7 Indigo
   http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/indigosr2
2. **Install "Kermeta IDE/Kermeta compiler" from the Eclipse update site**
   Latest tested version: Kermeta compiler 1.4.0
   http://www.kermeta.org/update
3. **For performance reasons, Kermeta requires more memory than standard eclipse applications**
   In order to increase the heap size, set -Xmx3000M in $EclipseDir$\eclipse.ini.
   On 32-bit CPU/OS, the maximum value is -Xmx1200M
   For details, see this post.
4. **Install "Web Tools Platform (WTP)" from the Eclipse update site**
   (Required to compile jUcmNav)
   Latest tested version: 3.3.2
   http://download.eclipse.org/webtools/repository/indigo/
5. **(Optional) Install the RAM Tool**
   Contact the RAM Tool development team to obtain the RAM Tool.

Check Out the Source Files

1. **Checkout svn://cserg0.site.uottawa.ca/projetseg/trunk**
2. **Import aoUrnToRam, codeGen, ReactiveWorkflow, seg.jUCMNav, seg.jUCMNav.aoUrnToRamPlugin**
   File - Import - General - Existing Project into Workspace
3. **Import the launch configurations**
   Import - Run/Debug - Launch Configurations and then select the aoUrnToRamWorkspace/toImport folder
4. **Open the Kermeta perspective in order to get the Kermeta menu**
   Window - Open Perspective- Others

Launch AoUrnToRam

Use the run button (Play sign down arrow - Run configuration) to launch:

- **jUcmNav**
  Eclipse Application - jUcmNav
  Right click on the UCM - Export - Export whole URN file - File type: Ram Reactive Workflows
- **Launch all integration tests**
  (Does not depend on jUCMNav)
  Kermeta Constraint Application - Integration
- **Launch all unit tests**
  (Does not depend on jUCMNav)
  Kermeta Constraint Application - TDD All
- **See Table 1 for other launch configurations**

Who is This Guide For?

This guide is intended for developers who want to maintain and extend the AoUrnToRam transformation. **The guide assumes that the reader has previously read** AoUrnToRam from the Outside In to have an overall understanding of the Aspect-oriented User Requirement Notation (AoURN), the Reusable Aspect Models (RAM) and how an AoURN model can be transformed into RAM models. The guide also assumes that the reader is familiar with object-oriented programming, but it does not require you to have any knowledge of Kermeta. However, if you are not familiar with aspect-oriented programming, you should read section 1.5 and section 2.20 of the Kermeta manual before you start reading this guide. Finally, if you are new to Kermeta, having a look at the Kermeta manual before you start coding would be a good idea.

Transformations Overview

Although users perceive AoUrnToRam as one transformation, AoUrnToRam is really a composition of many transformations. Figure 1 provides an overview of how these transformations collaborate to transform Aspect-oriented Requirements into Design Models. Note that iwToIwInsertInputProcessingNodes has to be executed before iwToIwLinkSteps. The other transformations can be executed in any order.

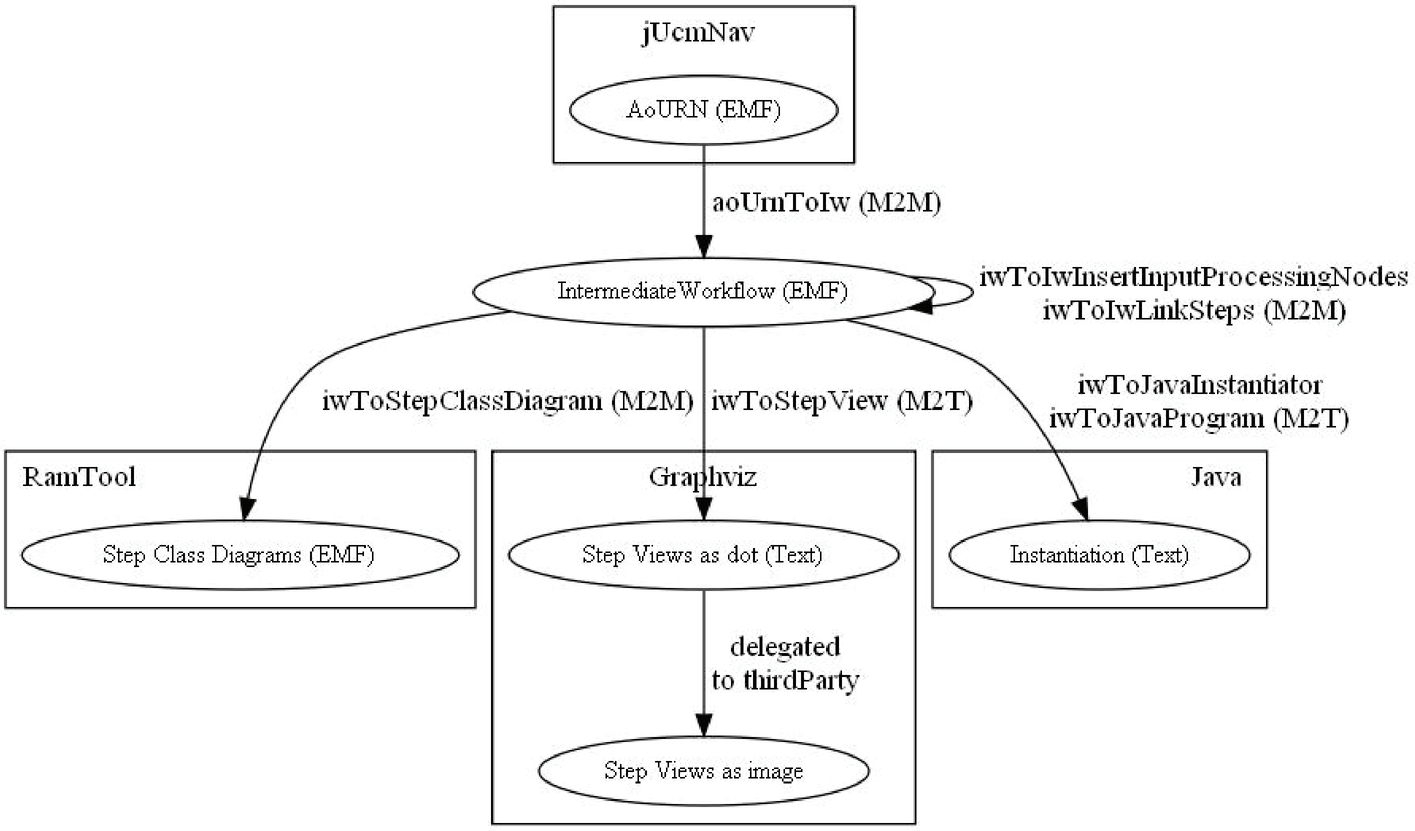


Figure 1. Transformation Overview

Source Code Organization

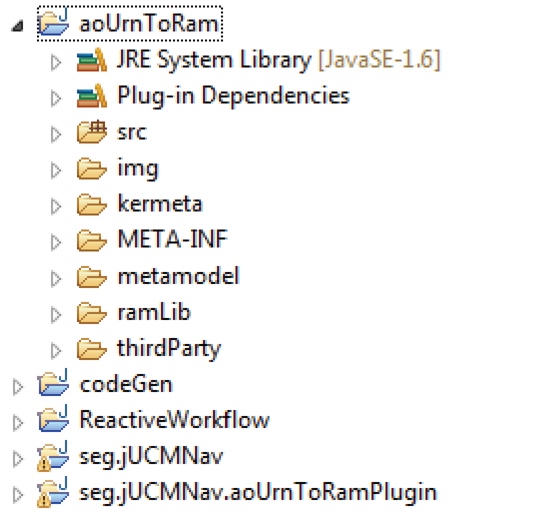

aoUrnToRam
JRE System Library [JavaSE-1.6]
Plug-in Dependencies
src
img
kermeta
META-INF
metamodel
ramLib
thirdParty
codeGen
ReactiveWorkflow
seg.jUCMNav
seg.jUCMNav.aoUrnToRamPlugin

Figure 2. Workspace Organization

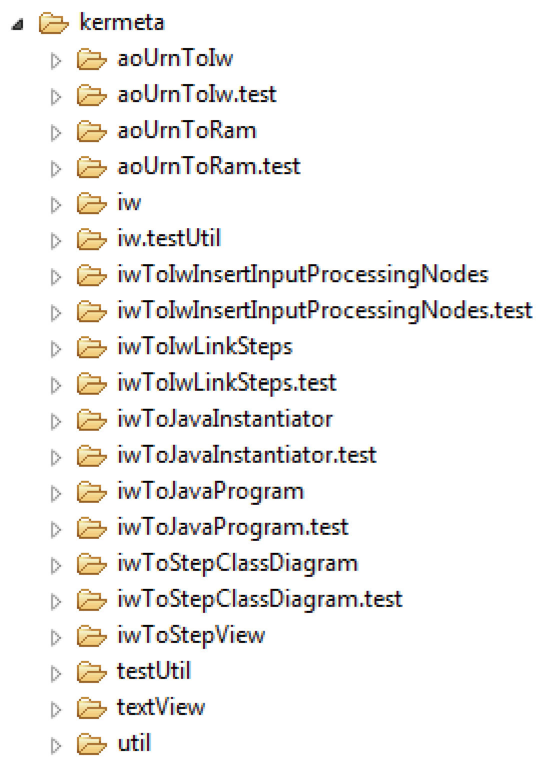

kermeta
aoUrnToIw
aoUrnToIw.test
aoUrnToRam
aoUrnToRam.test
iw
iw.testUtil
iwToIwInsertInputProcessingNodes
iwToIwInsertInputProcessingNodes.test
iwToIwLinkSteps
iwToIwLinkSteps.test
iwToJavaInstantiator
iwToJavaInstantiator.test
iwToJavaProgram
iwToJavaProgram.test
iwToStepClassDiagram
iwToStepClassDiagram.test
iwToStepView
testUtil
textView
util

Figure 3. Kermeta Code Organization

The main project in the aoUrnToRamWorkspace is aoUrnToRam. It is possible to invoke Java code from Kermeta code in order to overcome the limitations of the Kermeta language. The src folder contains this Java code. The Kermeta code of aoUrnToRam depends on this Java code; therefore, it is required to add the aoUrnToRam project to the runtime Java Classpath when creating a new launch configuration (see Figure 4. Runtime ClasspathFigure 4). Moreover, any exceptions thrown from the java code is logged into aoUrnToRamWorkspace\aoUrnToRam.javaExternalCall.log.

The img folder contains all the images used by GraphViz to generate the step views. The kermeta folder contains the source files that define the transformations. The kermeta folder will be expanded and explained in details later in this section. The AoURN metamodel, the Intermediate Workflow metamodel and the RAM metamodel are in the metamodel folder. The RAM models on which depend the step class diagrams yielded by the AoUrnToRam transformation are in the ramLib folder. The GraphViz executable is in the thirdparty folder.

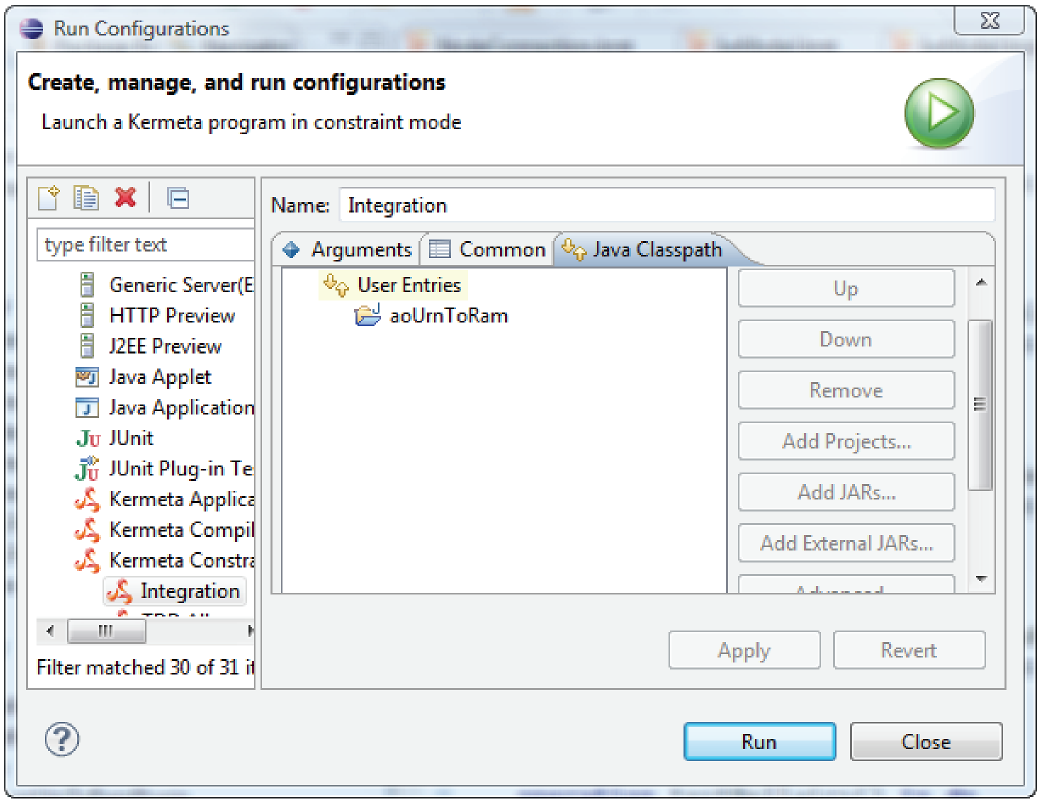
The codeGen project contains the Java source files of the code generator created to support the development of AoUrnToRam.

The ReactiveWorkflow project contains fake classes that can be used as substitute for the Reactive Workflow Framework, which does not exist yet. This allows for compiling and executing the Java code yielded by the AoUrnToRam transformation (for more details, see the Integration Testing Section).

The project seg.jUCMNav contains the source of jUCMNav. JUCMNav is the editor for AoURN models. The project seg.jUCMNav.aoUrnToRamPlugin is a jUCMNav plugin that allows for launching the AoUrnToRam transformation from jUCMNav.

As shown in Figure 3, each transformation of Figure 1 has its own folder. The aoUrnToRam transformation is the main entry point from where all the other transformation are invoked. Each transformation has its corresponding *.test folder that contains the related tests. The input files for the integration tests are in aoUrnToRam.test\jucm.

The folder iw contains the code shared by all the transformations having an Intermediate Workflow model as source and the folder iw.testUtil contains the code shared by all the unit tests related to a transformation having an Intermediate Workflow model as source. The folder testUtil contains the code shared by all tests. The folder textView contains the code shared by all M2T transformations (Model to Text). Finally, the folder util contains extensions to the Kermeta framework shared by all the transformations.


Run Configurations

Create, manage, and run configurations

Launch a Kermeta program in constraint mode

Name: Integration

Figure 4. Runtime Classpath

Tests

AoUrnToRam uses three test strategies: Exploratory Testing, Integration Testing and Unit Testing. Table 1 lists the launch configuration required to launch each test suite. The remainder of this section explains each test strategy in details.

Table 1. Test Launch Configurations

Name	Type	Purpose
jUcmNav	Eclipse Application	Exploratory Testing
Integration	Kermeta Constraint Application	Integration Testing
TDD All	Kermeta Constraint Application	Unit Testing
TDD AoUrnToIw	Kermeta Constraint Application	Unit Testing
TDD IwToIwInsertInputProcessingNodes	Kermeta Constraint Application	Unit Testing
TDD IwToIwLinkSteps	Kermeta Constraint Application	Unit Testing
TDD IwToStepClassDiagram	Kermeta Constraint Application	Unit Testing
TDD IwToJavaInstantiator	Kermeta Constraint Application	Unit Testing
TDD IwToJavaProgram	Kermeta Constraint Application	Unit Testing

Unit Testing

So far, AoUrnToRam has been developed using the Test Driven Development methodology (TDD). The name of this methodology has misled many to believe that it was a testing technique. In fact, TDD is a design technique that has the side effect of creating a lot of unit tests. These tests are good at asserting that the system behaviour respects the developer's intents. However, they are not useful to discover unexpected erroneous behaviours of the system under test. This is the main reason why this strategy is used in conjunction with integration testing and exploratory testing.

Providing an introduction to TDD is outside of the scope of this guide. However, TDD has a tremendous potential for helping future developers to maintain and extend AoUrnToRam. However, if you are not familiar with TDD, the learning curve should not be underestimated. This is true even if you are an experimented developer. Unit testing has its own challenges and can turn into a double-edged sword faster than one may think.

After the first three months of development, AoUrnToRam had 100 unit tests. At this point, the unit tests became a real maintenance burden. During the development of iwToStepView, we even stopped writing tests completely. This is the reason why no unit tests exist for iwToStepView. However, the problematic tests have been refactored since then and the number of unit tests has now reached 238 without creating any maintenance problems or slowing down the development.

These tests are not perfect. Still, they provide 238 examples for developers who may not be familiar with TDD and willing to try this methodology. You will notice that the methods provided through the TestHelpers (TestFactory and CustomAsserts) make it much easier to write unit tests. Still, if you are not familiar with writing unit tests, the Part 1 of XUnit test patterns[1] would be a great place to start. If you are curious to know more about the TDD methodology, Test Driven Development: By Example[2] by Kent Beck is a good book.

The naming of the test is a crucial matter for unit testing. Test names can be thought of as micro-requirements written by and for developers. Indeed, good names allow developers to understand the purpose of a test without reading the test code. In order to provide the benefit describe above, test names must respect the following convention. Note that SUT stands for System Under Test.

Test Class

1. SUT transformation name
2. "_"+SUT class name
3. "_"+SUT fixture (optional)
4. "TestCase"

Example: LinkSteps_IwNodeTestCase

Test Method

1. "test"
2. SUT method name
3. "_"+SUT input/preState important characteristics (optional)
4. "_should"+expected behaviour

Example: testDeepFirstSearch_Unexplored_ShouldLinkCurrentStep

[1] Meszaros, Gerad, xUnit test patterns: refactoring test code, Addison-Wesley, 2007
[2] Beck, Kent, Test-Driven Development: by Example, Addison-Wesley, 2002

Moreover, the traceability between a test and the features can be ensured by using the @feature attribute. Unit tests are very specific; thus, they are seldom related to more than one feature. When in doubt, use the feature for which the test was created first.

Example:
@feature "FeaNamingOfRAMStep"
operation testDeepFirstSearch_Unexplored_ShouldLinkCurrentStep() is do ...

Instead of manually adding the @feature to each new test, one can use any text editor (e.g. Sublime Text Editor) and the following regular expressions in order to add the @feature attribute to all the tests that are not linked with a feature.

Find: (^[\s\t]*\n)([\s\t]*)(operation test)
In: *TestCase.kmt
Replace: \n\2@feature "FEATURE_ID_HERE"\n\2\3

When these conventions are respected, the test names convey the essential information concerning the test. It is even possible to parse the test cases source file to get the essential information concerning all unit tests. This information can be used as micro-requirements by the developers. Table 2 provides an example for the feature FeaWorkflowInstantiationWithoutStubs which is solely related to the transformation IwToJavaInstantiator. The list of all features, their description, and the list of all test cases broken down by features can be found on the project wiki.

Table 2. Test Name as Requirements

SUT Class	Special Fixture	SUT Method	Pre State Characteristics	Expected Behaviour
IwCustomizableNode		AppendBuildStatement	NotShared	ShouldDeclareAndInitializeStepNode
IwCustomizableNode		AppendBuildStatement	Shared	ShouldDeclareAndInitializeStepNode
IwEndPoint		AppendBuildStatement		ShouldDeclareAndInitializeWorkflowNode
IwInput		AppendBuildStatement		ShouldDeclareAndInitializeWorkflowNode
IwModel		ToJavaInstantiator	ManyConcerns	ShouldInvokeToWorkflowInstantiatorOnWorkflows
IwModel		ToJavaInstantiator	ManyWorkflows	ShouldInvokeToWorkflowInstantiatorOnWorkflows
IwNodeConnection		AppendLinkStatement	NodeToNode	ShouldLinkSourceToTarget
IwNodeConnection		AppendLinkStatement	OrForkToNode	ShouldLinkSourceToTargetWithCondition
IwOrFork		AppendBuildStatement		ShouldDeclareAndInitializeWorkflowNode
IwOutput		AppendBuildStatement		ShouldDeclareAndInitializeStepNode
IwStartPoint	Bound	AppendBuildStatement		ShouldNotDeclareAndInitializeWorkflowNode
IwStartPoint	NotBound	AppendBuildStatement		ShouldDeclareAndInitializeWorkflowNode
IwStub		AppendBuildStatement		ShouldDeclareAndInitializeWorkflowNode
IwWorkflow	Build	ToWorkflowInstantiator		ShouldAppendClass
IwWorkflow	Build	ToWorkflowInstantiator		ShouldAppendImports
IwWorkflow	Build	ToWorkflowInstantiator		ShouldAppendPackage
IwWorkflow	Build	ToWorkflowInstantiator		ShouldInitCustomizableClassSubPackage
IwWorkflow	Build	ToWorkflowInstantiator		ShouldInitWorkspacePath
IwWorkflow	Link	ToWorkflowInstantiator	2Branches	ShouldInvokeAppendAddNextNodeStatementsOnNodeConnections
IwWorkflow	Link	ToWorkflowInstantiator	SequenceOf3Nodes	ShouldInvokeAppendAddNextNodeStatementsOnNodeConnections
IwWorkflow	Link	ToWorkflowInstantiator		ShouldAppendLinkMethod
IwWorkflow	Link	ToWorkflowInstantiator		ShouldAppendLinkNodesToNextNodesMethod

Integration Testing

The unit testing strategy is striving to test classes in isolation. In contrast, integration testing takes some interesting AoURN models as input and tests the transformation as a whole. The main purpose of this strategy is to discover erroneous behaviours of the system under test overlooked by the developer. When a problem is discovered, it is possible to formalize the expected behaviour with one or many unit tests and to fix the problem. Moreover, some aspects of the transformation that are not easily testable with unit testing are covered by the integration testing. For example, all the operations that depend on the file system such as loading EMF models, saving EMF models, saving text files and generating images with GraphViz, are not covered by unit testing but are covered by integration testing.

Moreover, the integration tests are not self-checking tests. That is, running the integration tests will not yield a pass/fail verdict as the unit tests do. At this point, the developer must use a diff viewer (e.g. Beyond Compare or other tools) to compare the actual output of the integration tests with a previous output. The integration tests could become self-checking tests, but automating this comparison has not been a priority so far.

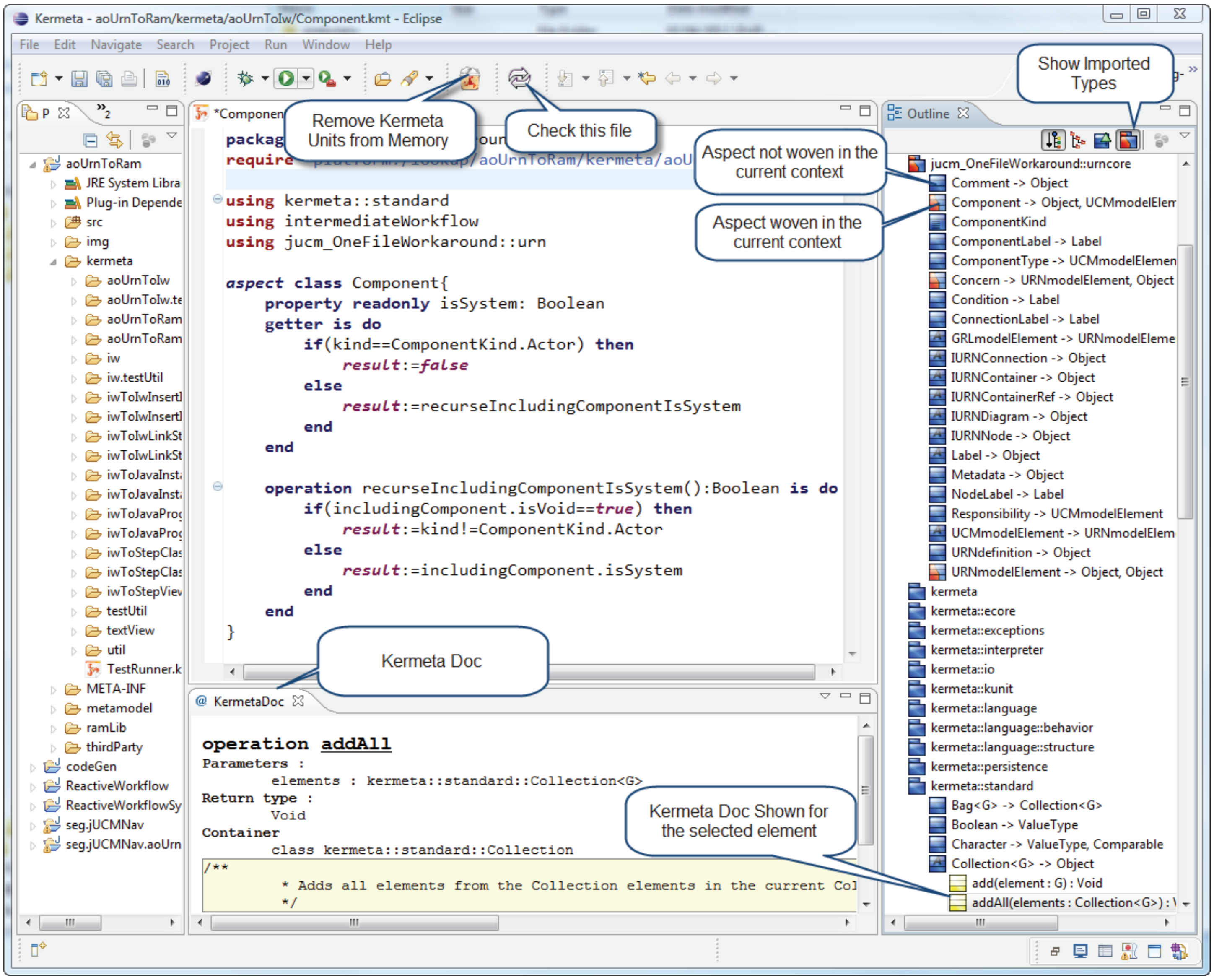
Also, the java code yielded by the integration tests can be compiled and executed in order to assert that the AoUrnToRam transformation does not create errors detectable by the Java compiler. This can be done by creating a Java project that depends on ReactiveWorkflow (Figure 1). Then, import the Java code from the aoUrnToRam.test\actual folder to the src folder of your Java project. Your Java project should compile and it should be possible to launch any Program.java generated by AoUrnToRam.

Finally, when the step class diagrams models are saved, the references must be serialized by using Universal Unique IDentifiers (UUIDs). Otherwise, the RAM Tool will not be able to load these step class diagrams. A UUID is essentially a random 16-byte number that is probabilistically guaranteed to be unique. From a testing perspective, using UUIDs has the undesirable consequence of making the AoUrnToIwRam transformation non-deterministic. That is, different UUIDs are serialized each time an integration test is executed. This significantly complicates the file comparison process since files that differ only by their UUIDs must be considered identical. To overcome this problem, index-based references are used for integration testing although they are not supported by the RAM Tool. The risk of overlooking a problem in the AoUrnToRam transformation because of the noise introduced by the UUIDs is much higher than the risk of finding a bug in the EMF framework, which supports both index-based and UUID references.

Exploratory Testing

The integration testing strategy does not consider the user interface. Thus, problems related to the integration with jUcmNav/Eclipse are not covered. The good news is that aoUrnToRamPlugin is rather minimalist. Its responsibility is mainly to invoke the AoUrnToRam transformation tested by the integration testing strategy. Thus, the aoUrnToRamPlugin should be very stable. The unit tests and the integration tests cover most of the code of the AoUrnToRam project. The integration with jUcmNav/Eclipse must be tested manually. Moreover, one must pay a special attention to ensure that the step class diagrams models are saved using UUID references since this is not covered by the integration tests.

Kermeta IDE


Kermeta - aoUrnToRam/kermeta/aoUrnToIw/Component.kmt - Eclipse
Remove Kermeta Units from Memory
Check this file
Show Imported Types
Aspect not woven in the current context
Aspect woven in the current context
Kermeta Doc
Kermeta Doc Shown for the selected element
Outline
KermetaDoc

```
packag
require    platform:/lookup/aoUrnToRam/kermeta/aoU

using kermeta::standard
using intermediateWorkflow
using jucm_OneFileWorkaround::urn

aspect class Component{
    property readonly isSystem: Boolean
    getter is do
        if(kind==ComponentKind.Actor) then
            result:=false
        else
            result:=recurseIncludingComponentIsSystem
        end
    end

    operation recurseIncludingComponentIsSystem():Boolean is do
        if(includingComponent.isVoid==true) then
            result:=kind!=ComponentKind.Actor
        else
            result:=includingComponent.isSystem
        end
    end
}
```

```
operation addAll
Parameters :
          elements : kermeta::standard::Collection<G>
Return type :
          Void
Container
          class kermeta::standard::Collection
/**
          * Adds all elements from the Collection elements in the current Col
          */
```

Figure 5. Kermata IDE

Although Kermeta is an interpreted language, a significant amount of time is required to load Kermeta Units into memory before running a Kermeta program. Once the Kermeta Units are loaded into memory, Kermeta inspects the changes made when a file is saved and updates the Kermeta Units. This speeds up the loading of Kermeta Units for subsequent execution. However, if the source files are modified from outside Eclipse (e.g. by using a source version control such as Tortoise SVN), you will need to use the "Remove Kermeta Units from Memory" button to explicitly reload all units into memory.

Also, the load process and the execution become slower and slower as more Kermeta Units get loaded into memory. Running "TDD All" or "Integration Tests" loads all Kermeta Units into memory. Thus, removing Kermeta Units from memory before working on a smaller transformation will reduce the load/execution time. When the TDD methodology is used, unit tests are launched very often. Executing a test suite more than 30 times in 60 minutes is not an exception. Thus, reducing the load/execution time is a critical issue. So far, it takes more or less 5 seconds to load/execute the unit test suite of one specific transformation, which is acceptable. The only exception is "TDD AoUrnToIw" which takes less than 10 seconds. Speeding up the "TDD AoUrnToIw" test suite would definitely be an improvement. Moreover, loading/executing "TDD All" and "Integration" takes less than 2 minutes which is not problematic since these tests are executed far less often. Note that execution time is reported in the test results, but this time does not take the loading time into account. If you are experiencing poor performance on your machine, ensure that the heap size of Eclipse was increased as explained in the Setting Up the AoUrnToRam Development Environment section.

The "Check this file" button is used to check for problems (similar to compilation-time errors) in a specific file. This process is time consuming and only reports problems found in the current file. If a file depends on many files, each dependency must be opened and checked one by one. By far, the most efficient way to check for errors in a transformation is to launch the corresponding test suite. If the transformation contains problems, Kermeta will report the first error in a message box. The most efficient way to read this error message is to copy the message from the message box to your favorite text editor. Do not expect a file name and a line number to localize the problem. You will have to rely on the error message to guess the location of the problem and then use the "Check this file" button to confirm that the error was really where you guessed. This process can be extremely painful. However, Kermeta is able to identify most problems when you save a file; thus, preventing from going through the painful process described above.

Moreover, Kermeta has a debugger. However, it has so many problems that it is better to avoid using it. Fortunately, having to rely on the debugger is far less frequent when TDD is used.

Also, Kermeta Intellisense is simply not usable. The best way to search for the operations available for a type is to use the outline. Clicking on "Show Imported Types" will display all imported packages including the Kermeta packages. Also, when an operation is selected in the outline, the corresponding documentation is presented in the KermataDoc.

Finally, as shown in Figure 5, classes with woven aspect in the current context are identified with a half red half blue icons. This is very useful to ensure that the aspects were woven as expected.

Design Decisions

Namespace Alias vs. Class Name Prefix

In general, namespaces are useful to restrict the scope in which name clashes could occur. If a name clash occurs, the class name prefixed with the namespace (fully qualified name) can be used to explicitly state which class should be used. However, using fully qualified names should be the exception and not the rule because they add complexity to the code. Therefore, one should use a class name that is not expected to cause a name clash in the normal use of the class.

In the context of the intermediateWorkflow namespace, most classes are expected to clash with either the classes of the source metamodel or with the classes of the destination metamodel. Therefore, each class of the intermediateWorkflow is prefixed with "Iw" to avoid relying on fully qualified name all the time.

Aspect vs. Visitor

Sébastien Mosser's lesson about model transformation with Kermeta has strongly influenced how model-to-model transformation (M2M) are performed in AoUrnToRam. The lesson introduces the build/link pattern presented in the next section. However, the implementation of the build/link pattern in AoUrnToRam differs from Mosser's implementation. Both implementations use aspects to weave references to the destination metamodel in the source metamodel. However, Mosser's implementation uses the visitor pattern[3] to encapsulate the transformation logic in a

[3] Gamma et al., Design Pattern Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, p. 331

class (the visitor) that neither belongs to the source metamodel nor to the destination metamodel. On the other hand, AoUrnToRam uses aspects to weave the transformation logic into the source metamodel.

When the visitor is responsible for the transformation logic, the visitor class suffers from feature envy: "The whole point of objects is that they are a technique to package data with the processes used on that data. A classic [code] smell is a method that seems more interested in a class other than the one it is in. The most common focus of the envy is the data"[4]. In Mosser's approach, the visitor has all the processes, and the source metamodel, all the data. In contrast, this problem does not occur when the transformation logic is woven into the source metamodel. Moreover, the visitor pattern relies on the double dispatch mechanism which also adds to the complexity of the code. Furthermore, since the visitor contains all the transformation logic, the complexity of the visitor class increases as the complexity of the transformation increases. Thus, the visitor class quickly becomes a bloated class.

On the other hand, using aspects to weave the transformation logic in the source metamodel causes problems when more than one transformation are woven in the same source model. The problem occurs when more than one aspect declare an operation with the same name. For instance, if two model-to-model transformations were implemented on the same source model, both transformations will probably define a build and a link method. In this case, Kermeta simply uses the first loaded operation without notifying the developer of the problem. This is essentially a problem with the Kermeta language. Although using the visitor pattern avoids this problem, the visitor pattern causes more harm than good in the AoUrnToRam context. To avoid the problem we simply added a prefix to the name of operation when more than one aspect are woven into the same metamodel. This measure is heavy, but it completely avoids the undesirable interactions between aspects that would be difficult to detect otherwise. This problem was discovered late in the development of AoUrnToRam. Thus, some transformations do not use operation name prefixes although they should.

Table 3. Aspect Interaction Workaround

Transformation	Prefix
IwToIwInsertInputProcessingNodes	None (ip)
IwToIwLinkSteps	None (ls)
IwToStepClassDiagram	None (sc)
IwToStepView	None (sv)
IwToJavaInstantiator	Ji
IwToJavaProgram	Jp

Build/Link Pattern from Model-to-Model Transformation

The Build/Link pattern can be summarized as follows:

1. **Build**
   Iterates through all the elements of the source model in any order.
   Each source element is responsible for building the corresponding destination elements.
2. **Link**
   Iterates through all the elements of the source model in any order.
   Each source element is responsible for linking the corresponding destination elements with the other elements of the destination model.

[4] Fowler et al., Refactoring: Improving the Design of Existing Code, 1999, p. 80

According to Mosser, this pattern is a classical approach for compilation. The transformations AoUrnToIw and IwToStepClassDiagram are based on this pattern. Moreover, we believe that the build/link pattern can be applied to any model-to-model transformation. This pattern has two major benefits:

1. To promote a clear separation between the building concern and the linking concern.
2. To prevent from depending on an object that has not been created yet.

To illustrate the Build/Link Pattern we propose to use an example issued from the AoUrnToIw transformation. For the sake of the example, we are only considering how the node and connections of a simple AoURN model are transformed into their Intermediate Workflow representation.

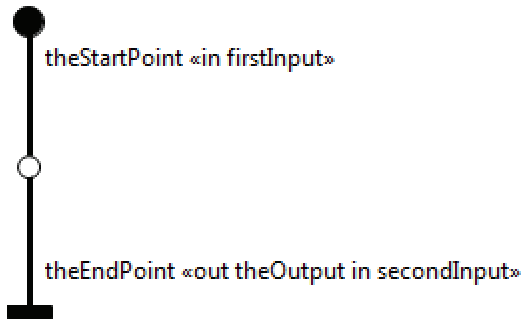

theStartPoint «in firstInput»
theEndPoint «out theOutput in secondInput»

Figure 6. Input of the Transformation (Use Case Map)

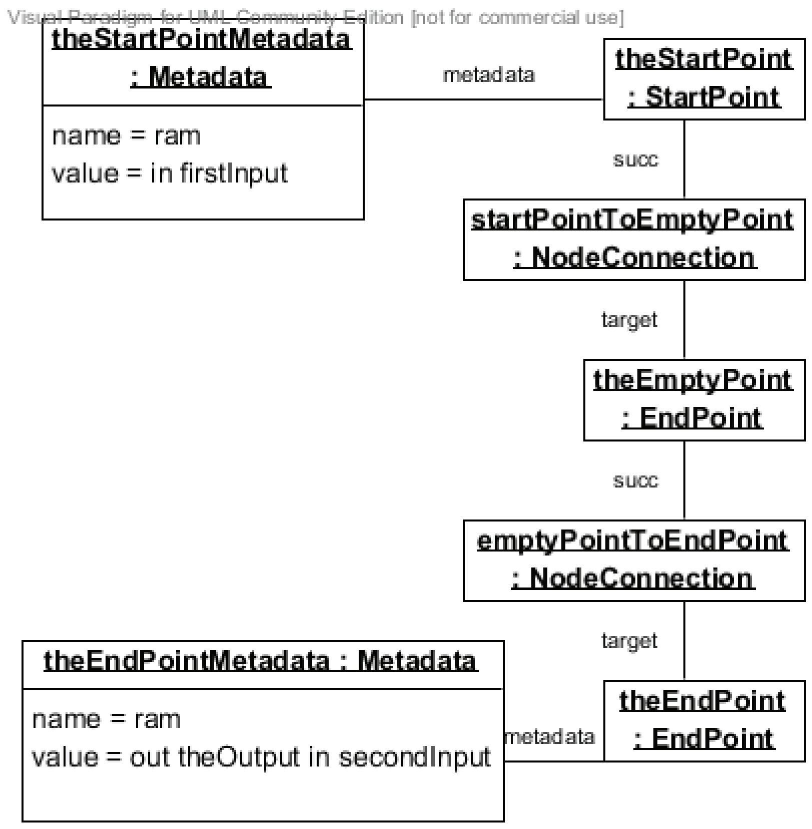

Visual Paradigm for UML Community Edition [not for commercial use]
theStartPointMetadata : Metadata
name = ram
value = in firstInput
metadata
theStartPoint : StartPoint
succ
startPointToEmptyPoint : NodeConnection
target
theEmptyPoint : EndPoint
succ
emptyPointToEndPoint : NodeConnection
target
theEndPointMetadata : Metadata
name = ram
value = out theOutput in secondInput
metadata
theEndPoint : EndPoint

Figure 7. Input of the transformation (Object Diagram)

Figure 7 uses an object diagram to provide a visual representation of the source model (AoURN). Observe how the metadata is used to define the input and output of the workflow. The fact that the "out" appears before the "in" in theEndPointMetadata may be confusing. This has to be read from the user perspective: the user receives "theOutput" from the system before providing the "secondInput" to the system.

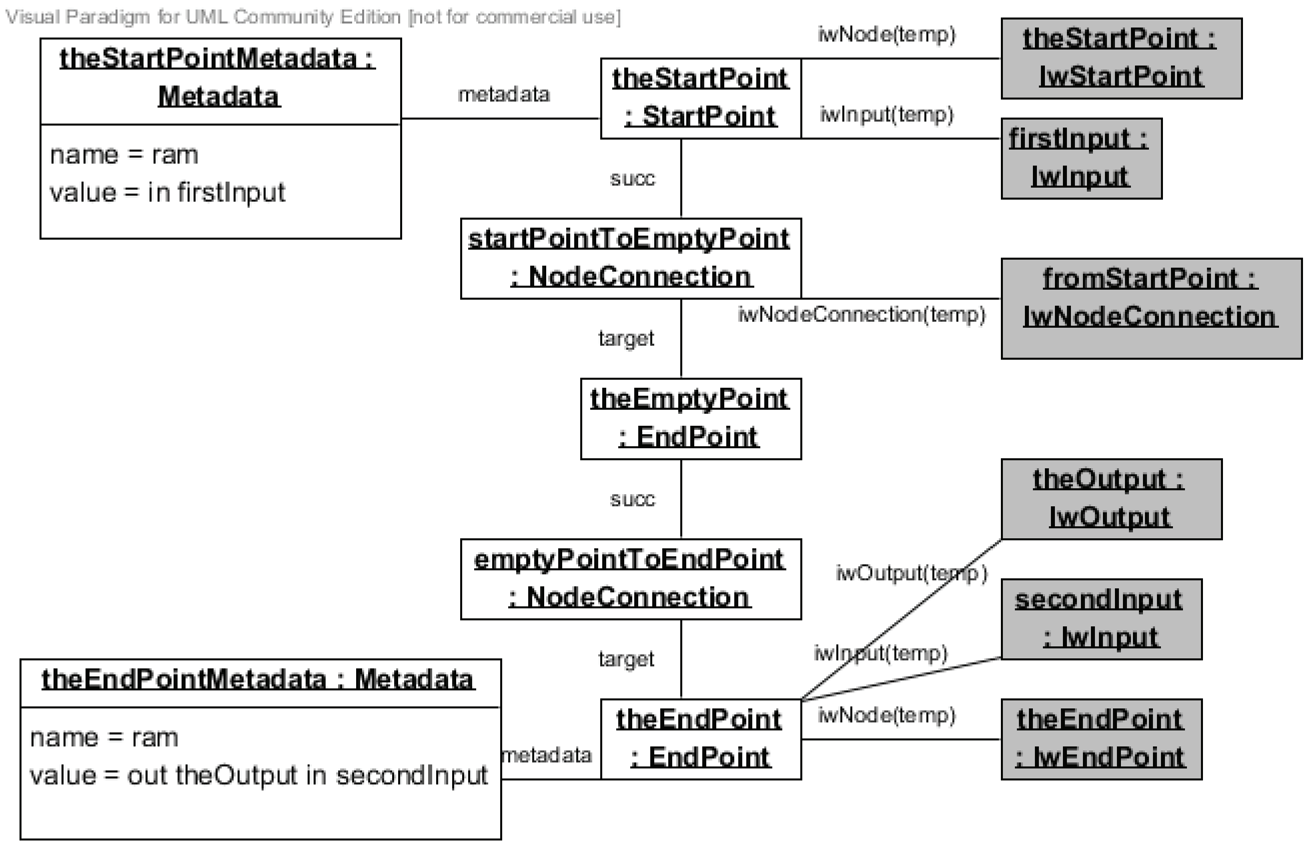

Visual Paradigm for UML Community Edition [not for commercial use]
theStartPointMetadata : Metadata
name = ram
value = in firstInput
metadata
theStartPoint : StartPoint
iwNode(temp)
theStartPoint : IwStartPoint
iwInput(temp)
firstInput : IwInput
succ
startPointToEmptyPoint : NodeConnection
iwNodeConnection(temp)
fromStartPoint : IwNodeConnection
target
theEmptyPoint : EndPoint
succ
theOutput : IwOutput
emptyPointToEndPoint : NodeConnection
iwOutput(temp)
secondInput : IwInput
iwInput(temp)
target
theEndPointMetadata : Metadata
name = ram
value = out theOutput in secondInput
metadata
theEndPoint : EndPoint
iwNode(temp)
theEndPoint : IwEndPoint

Figure 8. After the Build Phase

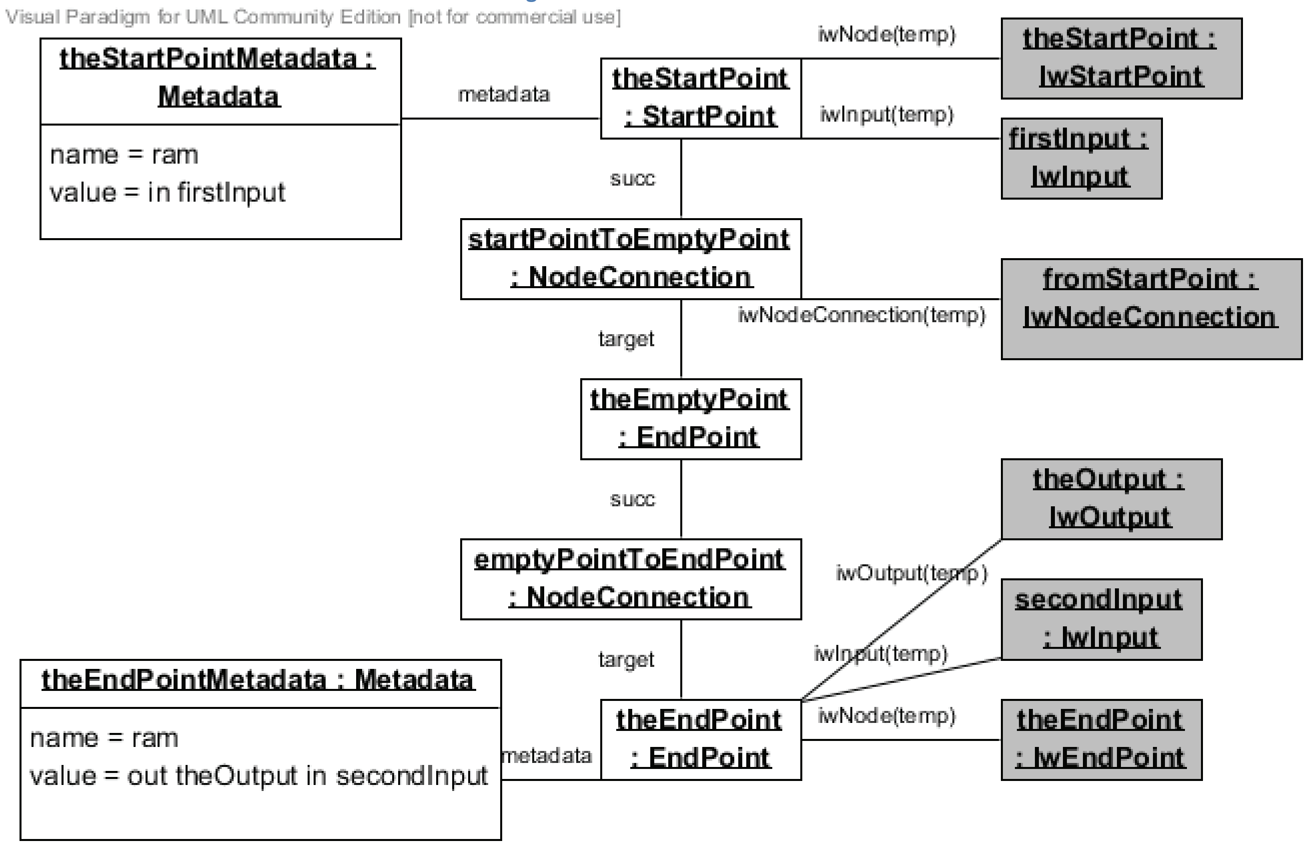

Visual Paradigm for UML Community Edition [not for commercial use]
theStartPointMetadata : Metadata
name = ram
value = in firstInput
metadata
theStartPoint : StartPoint
iwNode(temp)
theStartPoint : IwStartPoint
iwInput(temp)
firstInput : IwInput
succ
startPointToEmptyPoint : NodeConnection
iwNodeConnection(temp)
fromStartPoint : IwNodeConnection
target
theEmptyPoint : EndPoint
succ
theOutput : IwOutput
emptyPointToEndPoint : NodeConnection
iwOutput(temp)
secondInput : IwInput
iwInput(temp)
target
theEndPointMetadata : Metadata
name = ram
value = out theOutput in secondInput
metadata
theEndPoint : EndPoint
iwNode(temp)
theEndPoint : IwEndPoint

Figure 8 presents the state of the source model after the Build phase. Observe that the white object instances belong to the AoURN Model and that the gray object instances belong to the Intermediate Workflow model. The links between the AoURN model and the Intermediate Workflow model are only there to support the Link phase and can be removed once the transformation is completed.

You will note that startPointToEmptyPoint built a destination object while emptyPointToEndPoint did not. The reason is that the EmptyPoints are not transformed to the Intermediate Workflow model. If their source is not represented in the Intermediate Workflow model, NodeConnections are not responsible for building a destination object. Moreover, observe that one AoURN object can yield many Intermediate Workflow objects. For example, theEndPoint:EndPoint is responsible for building 3 Intermediate Workflow objects. Furthermore, many AoURN objects (e.g. theEndPoint and theEndPointMetadata) can collaborate to build the Intermediate Workflow objects.

The important point is that the source objects (AoURN) should not collaborate with the destination objects (Intermediate Worklfow) during the build phase. Respecting this rule prevents from depending on an object that has not been created yet. Therefore, it allows for building and linking the destination objects in any order. This benefit should not be underestimated, especially for complex transformations.

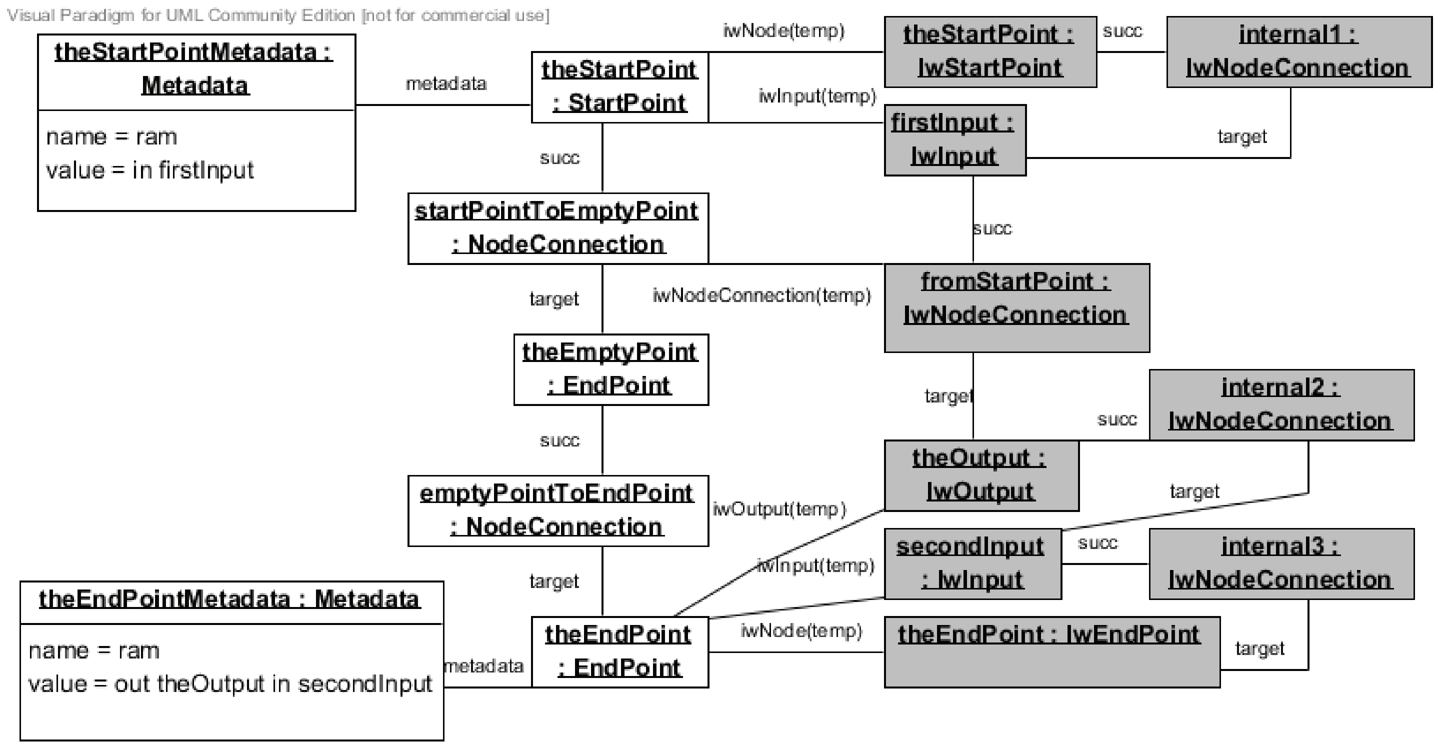

Visual Paradigm for UML Community Edition [not for commercial use]
theStartPointMetadata : Metadata
name = ram
value = in firstInput
metadata
theStartPoint : StartPoint
iwNode(temp)
iwInput(temp)
theStartPoint : IwStartPoint
succ
internal1 : IwNodeConnection
firstInput : IwInput
target
succ
startPointToEmptyPoint : NodeConnection
succ
target
iwNodeConnection(temp)
fromStartPoint : IwNodeConnection
theEmptyPoint : EndPoint
target
succ
internal2 : IwNodeConnection
succ
theOutput : IwOutput
emptyPointToEndPoint : NodeConnection
iwOutput(temp)
target
secondInput : IwInput
succ
internal3 : IwNodeConnection
iwInput(temp)
target
theEndPointMetadata : Metadata
name = ram
value = out theOutput in secondInput
metadata
theEndPoint : EndPoint
iwNode(temp)
theEndPoint : IwEndPoint
target

Figure 9. After Link Phase

The state of the source model after the Link phase is presented in Figure 9. First, observe how fromStartPoint has been linked in the destination model. StartPointToEmptyPoint has collaborated with theStartPoint in order to obtain its exit node (firstInput). Moreover, StartPointToEmptyPoint has collaborated with theEmptyPoint, emptyPointToEndPoint and theEndPoint in order to obtain the next entry node (theOutput).

Also, you will note that internal1, internal2 and internal3 have been built during the linking phase. This does not respect the Build/Link pattern. However, one has to be pragmatic when using a pattern. Internal connections are always built and linked by the same source node. Thus, it has no consequences on the benefits provided by the Build/Link pattern. Moreover, it would be easy to build the internal connections during the build phase if it causes problems in the future.

If the IwNodeConnection is not internal (e.g. fromStartPoint), the same logic cannot be applied. How stubs are built and linked is not presented in this example. However, a stub binding needs to be linked with a node connection. Thus, if a node connection is not internal, it needs to be built during the build phase in order to prevent from linking a binding to a node connection that has not been created yet during the link phase.

Finally, observe that secondInput is directly connected to theEndPoint. In such case, an implicit input processing node must be added between secondInput and theEndPoint. However, another transformation (iwToIwInputProcessingNodes) is responsible for inserting the implicit input processing nodes.

Pretty Printer Pattern for Model-to-Text Transformation

The pattern used to transform models to text is more intuitive than the Build/Link pattern. However, in general, model-to-text transformations are more complex than model-to-model transformations. If the source model is too different from the textual destination, consider creating a model of the textual destination. Then, transform the source model into the model of the textual destination and then transform the model of the textual destination into the textual destination. For example, the text ("dot" language) required by GraphViz to generate a Step View image is much closer to an Intermediate Workflow model than to an AoURN model. Thus, it makes sense to transform an AoURN model to an Intermediate Workflow model before attempting to create the "dot" representation of a Step View.

Kermeta proposes two approaches for model-to-text transformation: Kermeta Emitter Template (KET) and pretty printer. The former is a template-based approach which is very similar to other Java template framework such as JET, JSP and Velocity and the later is a code-based approach. As stated in the KET manual, "Using KET is efficient when templates contain more text than expressions (i.e., a text with holes). If you need more computation than text, you should consider writing a pretty printer directly in Kermeta".

All models to text transformation of AoUrnToRam (iwToStepView, iwToJavaInstantiator and iwToJavaProgram) use the pretty printer approach. The first reason is that they require more computation than static text and the second reason is that the pretty printer approach is easier to unit test than KET templates.

Validation

Some AoURN models are invalid from the AoUrnToRam perspective. For example, for all stubs in an AoURN model, each in-path and out-path of a stub must be bound; otherwise the AoURN model is invalid. By design, the source AoURN model must be validated before invoking the transformation. The behaviour of AoUrnToRam is unpredictable when an invalid AoURN model is provided as source. The constraints to be validated before invoking the AoUrnToRam transformation are listed in the feature list. Each validation feature is prefixed with "FeaValidate".

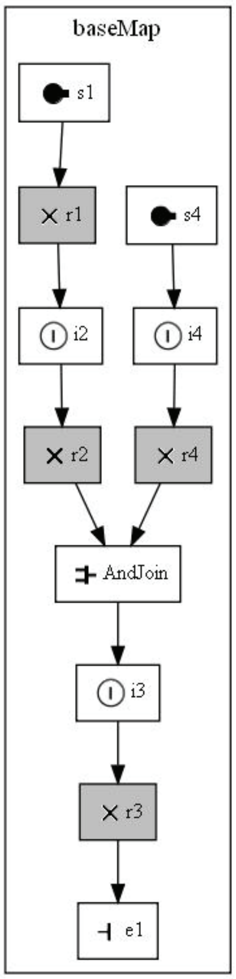
At this point, none of these constraints are implemented. Thus, it is impossible for the users to know for sure if their AoURN models are invalid (yielding unpredictable results) or valid.

Linking Steps

The transformation iwToIwLinkSteps is responsible for linking each node with a step. The algorithm used for this transformation is an adaptation of the depth-first search algorithm. Three examples are provided in order to demonstrate how the algorithm differs from a classic depth-first search. In the tables presented below, each column represents a node and each row shows which node is linked to which step at a specific point in time. When a link is added or updated, it is shown in bold face. Also, the graphs presented in this section were manually created to provide a visual representation of a complete workflow system. These graphs are not step views.

Merging Steps

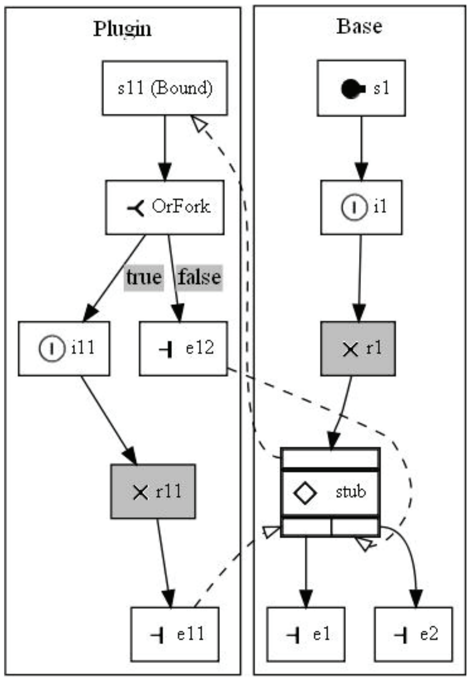
Time	s1	r1	i2	r2	AndJoin	i3	r3	e1	s4	i4	r4
1	**s1**										
2	s1	**s1**									
3	**i2**	**i2**	**i2**								
4	i2	i2	i2	**i2**							
5	i2	i2	i2	i2	**i2**						
6	i2	i2	i2	i2	i2	**i3**					
7	i2	i2	i2	i2	i2	i3	**i3**				
8	i2	i2	i2	i2	i2	i3	i3	**i3**			
9	i2	i2	i2	i2	i2	i3	i3	i3	**s4**		
10	i2	i2	i2	i2	i2	i3	i3	i3	**i4**	**i4**	
11	i2	i2	i2	i2	i2	i3	i3	i3	i4	i4	**i4**
12	**i2_i4**	**i2_i4**	**i2_i4**	**i2_i4**	**i2_i4**	i3	i3	i3	**i2_i4**	**i2_i4**	**i2_i4**



- The nodes (s1, r1) precede the first input (i2) received by the system. At time 3, the current step is renamed with the name of the first input. Also, the same case occurs at time 10.
- When the second input is reached at time 6, a new step (i3) is created instead of renaming the current step.
- Also, at time 8 the exploration has explored all nodes starting from s1. In this case, the algorithm continues with the next start point (s4) that is not bound by a stub from the same concern.
  Finally, at time 12 the exploration hits an node that belongs to another step. In that case, the two steps are merged yielding a step named i2_14.

Plug-in from the Same Concern

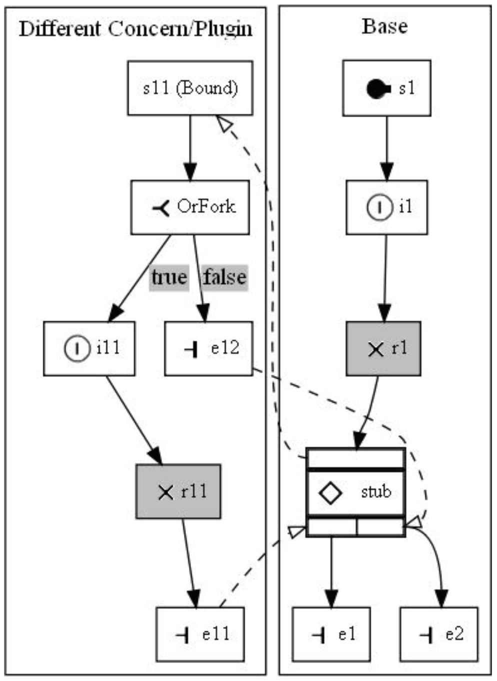
	Base						Plugin					
Time	**s1**	**i1**	**r1**	**stub**	**e1**	**e2**	**s11**	**OrFork**	**i11**	**r11**	**e11**	**e12**
1	**s1**											
2	**i1**	**i1**										
3	i1	i1	**i1**									
4	i1	i1	i1	**i1**								
5	i1	i1	i1	i1			**i1**					
6	i1	i1	i1	i1			i1	**i1**				
7	i1	i1	i1	i1			i1	i1	**i11**			
8	i1	i1	i1	i1			i1	i1	i11	**i11**		
9	i1	i1	i1	i1			i1	i1	i11	i11	**i11**	
10	i1	i1	i1	i1	**i11**		i1	i1	i11	i11	i11	
11	i1	i1	i1	i1	i11	**i1**	i1	i1	i11	i11	i11	
12	i1	i1	i1	i1	i11	i1	i1	i1	i11	i11	i11	**i1**



- At time 5, unlike for other nodes the exploration does not use the outgoing connections (solid arrows) to continue the exploration when a stub is reached. Instead, the in-binding (dashed arrows) are used.
- Also, at time 10 the out-binding is used to continue the exploration when the end-point e11 is reached. A similar case occurs at time 12.
- Moreover, observe that the stub is linked with the step before continuing the exploration through the in-binding (time 4), not after continuing the exploration through the out-bindings (time 10 and 12). However, when the exploration continues through the out-bindings of a stub, this stub is linked with the current step as an outbound stub (See Intermediate Workflow Metamodel - Global View). In the example, stub is linked with i1 but is an outbound step of i1 and i11.

Plug-in from Different Concerns

	Base						Plugin					
Time	s1	i1	r1	stub	e1	e2	s11	OrFork	i11	r11	e11	e12
1	**s1**											
2	**i1**	**i1**										
3	i1	i1	**i1**									
4	i1	i1	i1	**i1**								
5	i1	i1	i1	i1	**i1**							
6	i1	i1	i1	i1	i1	**i1**						
7	i1	i1	i1	i1	i1	i1	**s11**					
8	i1	i1	i1	i1	i1	i1	s11	**s11**				
9	i1	i1	i1	i1	i1	i1	**i11**	**i11**	**i11**			
10	i1	i1	i1	i1	i1	i1	i11	i11	i11	**i11**		
11	i1	i1	i1	i1	i1	i1	i11	i11	i11	i11	**i11**	
12	i1	i1	i1	i1	i1	i1	i11	i11	i11	i11	i11	**i11**



- This example is exactly the same as the previous one except that the plug-in is part of a different concern than the base workflow. In that case, the in-binding and out-binding are not used. Instead, the exploration continues as if the stub was a normal node (time 5 and 6). Note that aspect markers are always processed as normal nodes even if their plug-in is part of the same concern.
- Also, at time 6 the exploration has explored all nodes starting from s1. The exploration continues with s11 since this start point is not bound by a stub from the same concern.

Code Generation

Some code generators were written in order to solve some recurring problems encountered during the development of AoUrnToRam. Each of these code generator is discussed in this section.

Table 4. Code Generation Launch Configuration

Name	Type	Launch From
Update Ref	Java Application	Current file must be a _Ref.kmt file.
Gen TestRunner	Java Application	Current file can be any sibling of the TestRunner.CodeGen.kmt file to be generated.
Generate Jucm_OneFileWorkaround.ecore	Java Application	Can be launched from anywhere, but the launcher configuration must be customized.

Update Ref

Kermeta uses "require" statements to structure the code base. For example, aoUrnToRam/kermeta/TestRunner.kmt depends on aoUrnToRam/kermeta/testUtil/CustomTestRunner.kmt. Thus, the header of TestRunner.kmt must contain the following statement: require "platform:/lookup/aoUrnToRam/kermeta/testUtil/CustomTestRunner.kmt". Managing these require statements is time-consuming and error-prone.

The _Ref.kmt files are used in order to mitigate this problem. A _Ref.kmt file depends on each file of a transformation and each file of the transformation depends on the _Ref.kmt file. Thus, the require statement can be managed at the transformation level instead of on a file per file basis. Moreover, only the _Ref.kmt file needs to be referenced from outside the transformation in order to use the whole transformation. The "Update Ref" launch configuration can be used to automatically refresh the "require" statement of the _Ref.kmt file.

The _Ref.kmt files create a lot of cyclic dependencies. However, the Kermeta Manual state that "you can create cyclic dependencies of files, the environment will deal with that". One may suspect that Kermeta would be able to load one large source file than many small source files. However, _Ref.kmt files were introduced when the AoUrnToRam code base was reorganized from a one file per transformation basis to a one file per class basis and no significant increase on the time required to loads all Kermeta Units into memory was measured.

Note: Use platform:/lookup instead of platform:/resource or platform:/plugin in the "require" statements. Using platform:/lookup is a shortcut to say use either platform:/resource (for development) or platform:/plugin (for production).

Gen TestRunner

The KUnit framework distributed with Kermeta does not allow to automatically detect the test classes. Indeed, each test class must be added to a test suite manually. Again this is a time-consuming and error-prone process that can be avoided by using a code generator. Running "Gen TestRunner" will create a TestRunner that executes a test suite that includes all test classes contained in the $TransformationName$.test folder. The test classes must respect the naming convention explained in the Unit Testing section. If the TestRunner has problems (compiler-like errors), make sure that the test classes respect the naming convention.

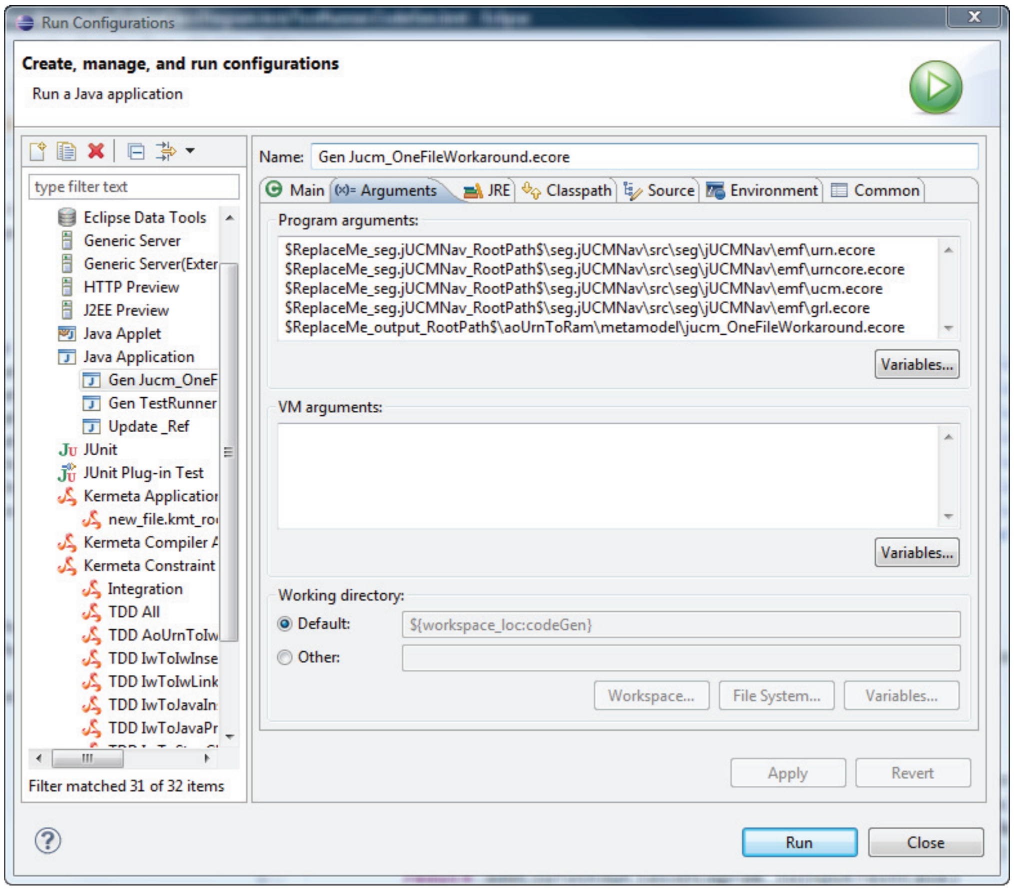
Generate Jucm_OneFileWorkaround.ecore

A bug in Kermeta prevents from loading a metamodel if many files with cyclic dependencies between them are used to define the metamodel, which is the case for the AoURN metamodel. Indeed, four files (urn.ecore, urncore.ecore, ucm.ecore and grl.ecore) are used to define the AoURN metamodel and each of these files references the three other files. Indeed, having a bidirectional association between two classes that are defined in two different .ecore files creates a cyclic dependency between these files.

To work around this problem, the four .ecore files that define the AoURN metamodel are merged into one file: jucm_OneFileWorkaround.ecore. The AoUrnToRam transformation depends on jucm_OneFileWorkaround.ecore while jUCMNav depends on the four original .ecore files. Note that jucm_OneFileWorkaround.ecore is designed to be used by the AoUrnToRam transformation only. Thus, jucm_OneFileWorkaround.ecore should not be registered to the EMF registry in order to prevent side effects between AoUrnToRam and jUCMNav.

Each time the AoURN metamodel changes, jucm_OneFileWorkaround.ecore must be maintained in order to reflect the four original .ecore files. Again, a code generator was written to avoid this time-consuming and error-prone process.

Unlike the other launch configuration for code generation, "Generate Jucm_OneFileWorkaround.ecore" has to be customized to your development environment. The reason is that the four original .ecore files and the jucm_OneFileWorkaround.ecore do not belong to the same project. Thus, one must customize $ReplaceMe_seg.jUCMNav_RootPath$ and $ReplaceMe_output_RootPath$ for their development environment (as shown in Figure 10) before "Generate Jucm_OneFileWorkaround.ecore" can be executed.



Name: Gen Jucm_OneFileWorkaround.ecore

Program arguments:

```
$ReplaceMe_seg.jUCMNav_RootPath$\seg.jUCMNav\src\seg\jUCMNav\emf\urn.ecore
$ReplaceMe_seg.jUCMNav_RootPath$\seg.jUCMNav\src\seg\jUCMNav\emf\urncore.ecore
$ReplaceMe_seg.jUCMNav_RootPath$\seg.jUCMNav\src\seg\jUCMNav\emf\ucm.ecore
$ReplaceMe_seg.jUCMNav_RootPath$\seg.jUCMNav\src\seg\jUCMNav\emf\grl.ecore
$ReplaceMe_output_RootPath$\aoUrnToRam\metamodel\jucm_OneFileWorkaround.ecore
```

Working directory: [✓ Default: ${workspace_loc:codeGen}] Other:

Figure 10. Gen Jucm_OneFileWorkaround.ecore Must be Customized

Interfacing with the RAM Tool

The AoUrnToRam transformation code is part of the same code base as the AoURN metamodel. In contrast, the RAM metamodel is part of another code base since the RAM Tool is developed at the McGill University. The AoUrnToRam transformation does not completely depend on the RAM Tool but only on the RAM metamodel. When a new version of the RAM tool is released, the RAM metamodel must simply be copied to the aoUrnToRam\metamodel folder. Moreover, as explained in the integration testing section, the RAM Tool does not use the default EMF serialization mechanism. The code responsible for serializing the RAM models must be copied from the code base of the RAM Tool to aoUrnToRam\source\ca\mcgill.

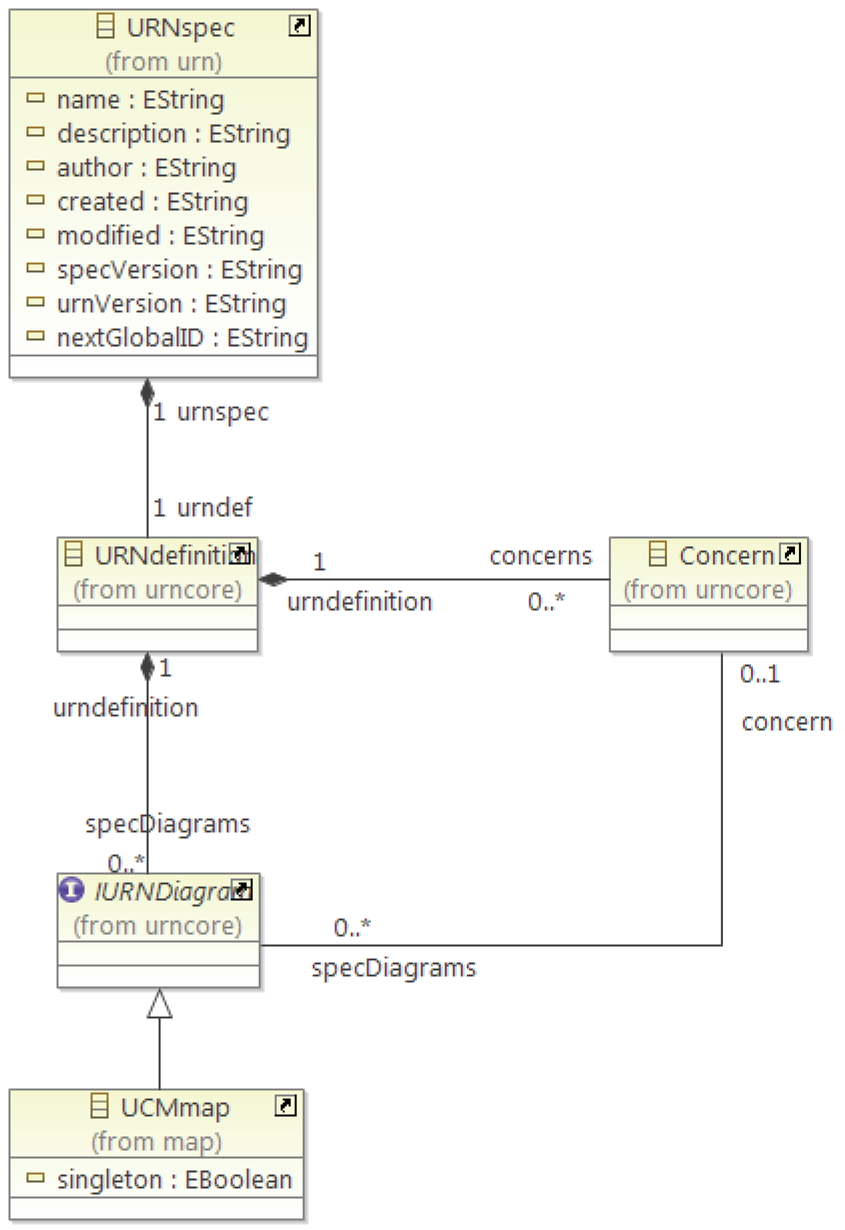
Kermeta requires the RAM serialization mechanism to be registered in the global EMF registry. Since the global registry is shared by all Eclipse application, a problem will occur if both the AoUrnToRam transformation and the RAM Tool registered the same serialization mechanism to the global EMF registry. Fortunately, the RAM Tool is not an Eclipse application but a standalone application. Thus, this problem should not occur.

Finally, the RAM models generated by the AoUrnToRam transformation depend on the RAM models of the RAM Workflow Middleware. The RAM models of the RAM Workflow Middleware must be copied to aoUrnToRam\ramLib

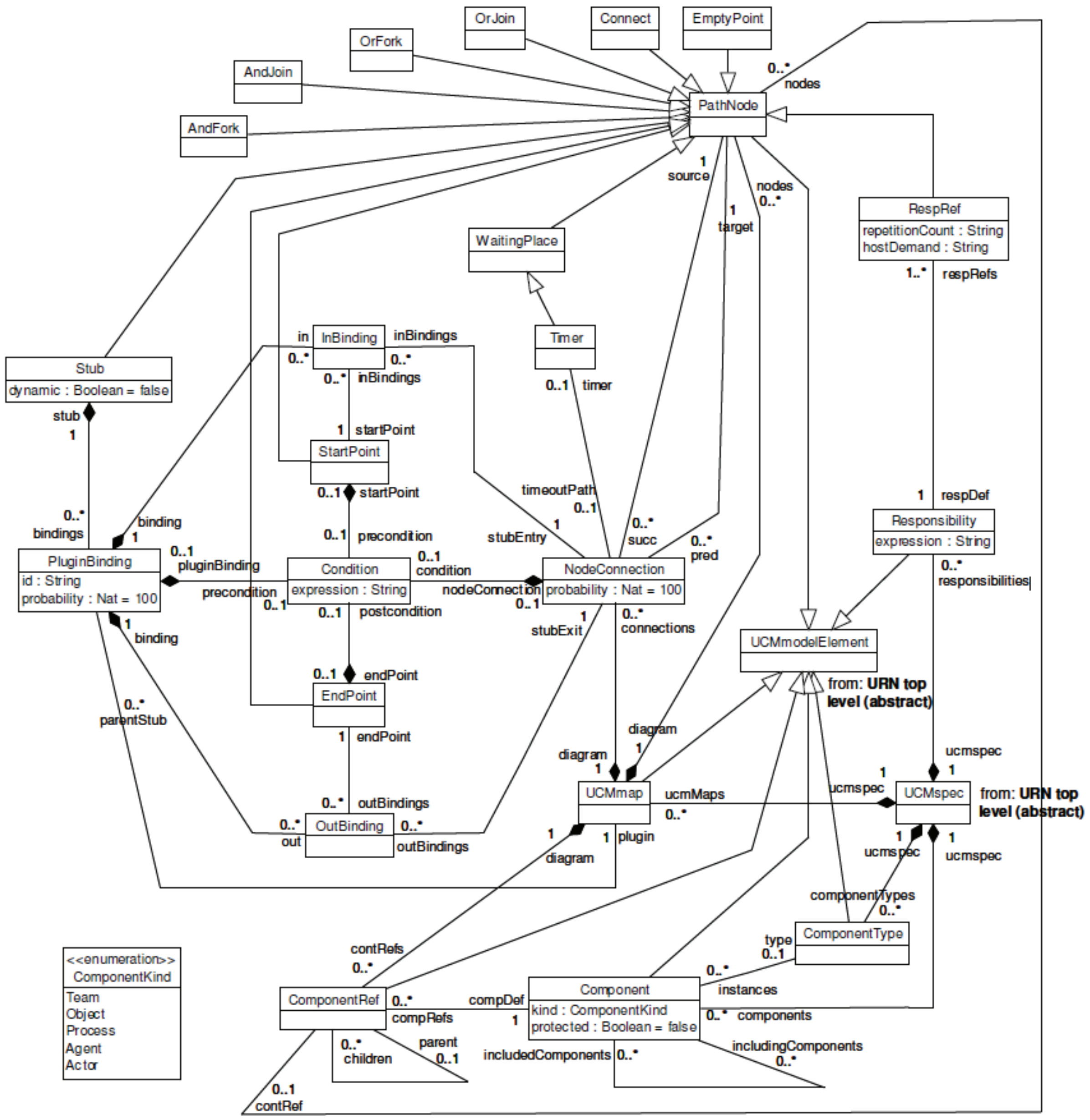
Releasing a New Version of AoUrnToRam

1. **Rebuild all projects**
   Project - Clean - Clean all projects
2. **Export aoUrnToRamPlugin**
   Right click on seg.jUCMNav.aoUrnToRamPlugin - Export - Plug-ins and fragments - Finish
3. **Copy the aoUrnToRam folder to a temporary location**
4. **Remove the tests**
   a. Delete all .test folders
   b. Delete the testUtil folder
   c. Delete TestRunner.kmt
5. **Package the aoUrnToRamPlugin and aoUrnToRam**
   Zip the exported plugin (seg.jUCMNav.aoUrnToRamPlugin_$version.jar) and the aoUrnToRam temporary folder to aoUrnToRam_release_$version.zip.
   Note: Make sure that both seg.jUCMNav.aoUrnToRamPlugin_$version.jar and the aoUrnToRam folders are at the root level of the zip. That is, when you extract the zip to the eclipse/dropins folder the extracted files should be:
   - eclipse/dropins/seg.jUCMNav.aoUrnToRamPlugin_$version.jar
   - eclipse/dropins/aoUrnToRam/
6. **Publish on the Wiki**
   Attach the zip file to http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/AoUrnToRamRelease

AoURN Metamodel - High Level View

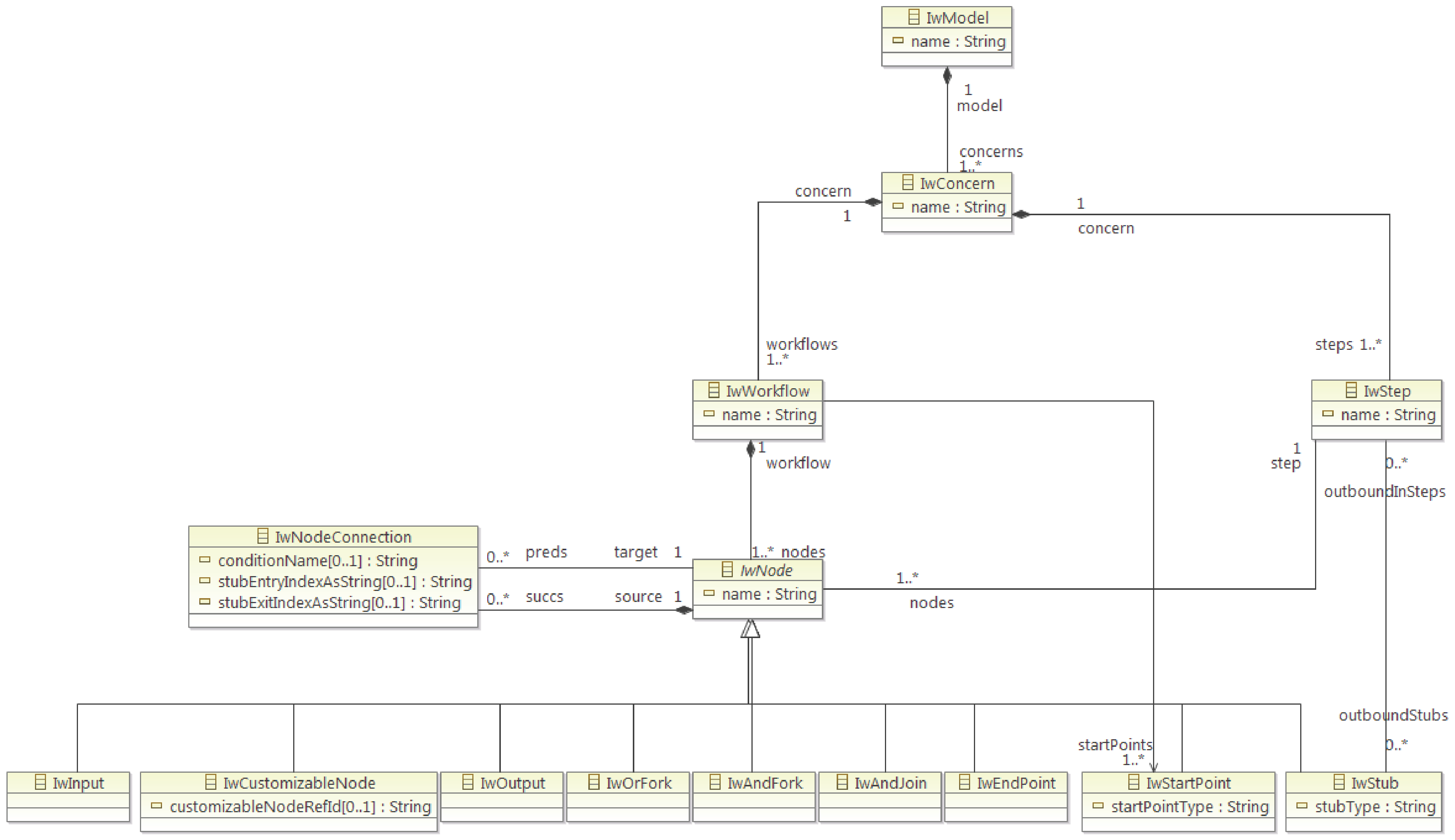


AoURN Metamodel - UCM Map Level View

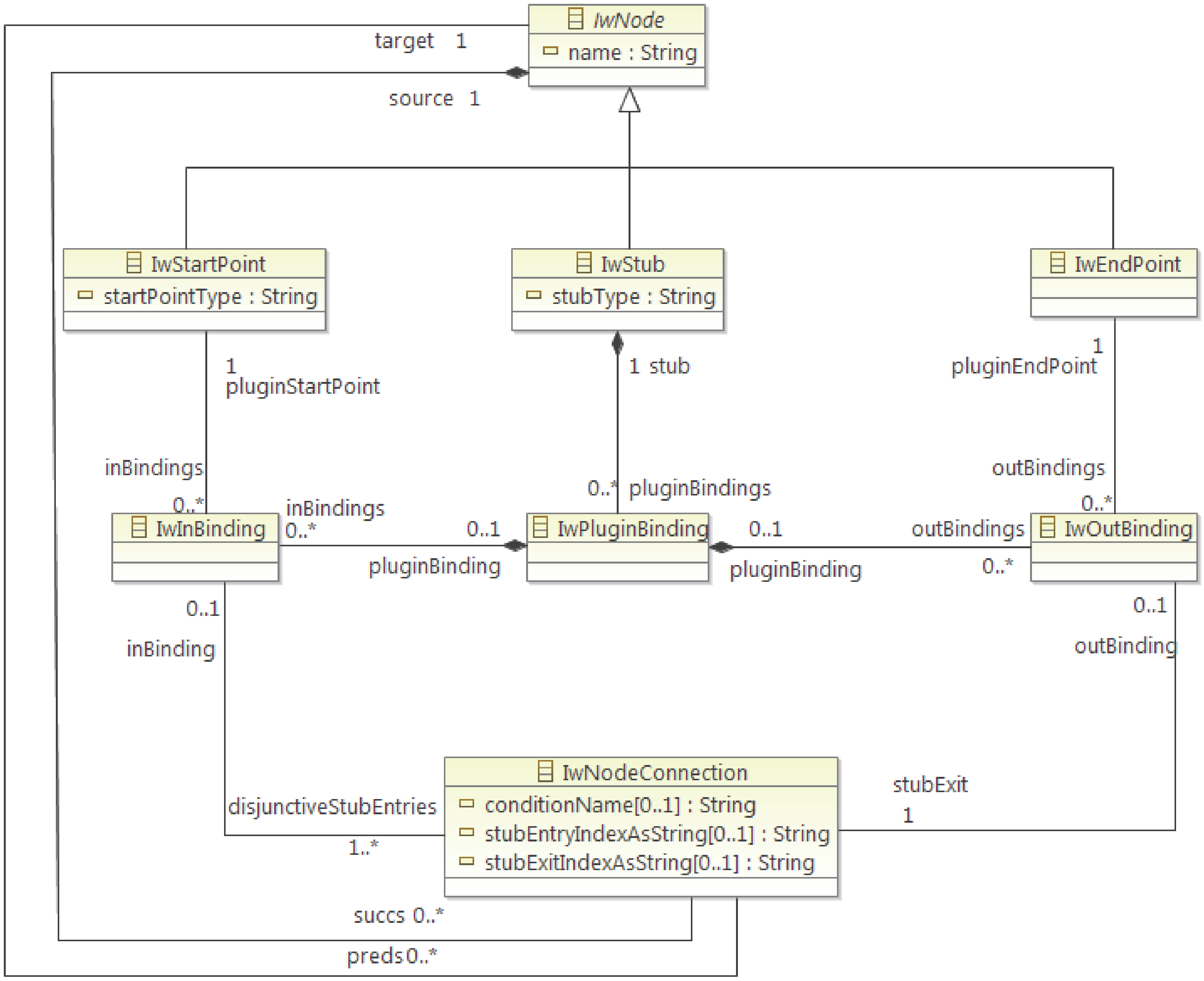


Source: Gunter Mussbacher, Aspect-Oriented User Requirements Notation, p. 305

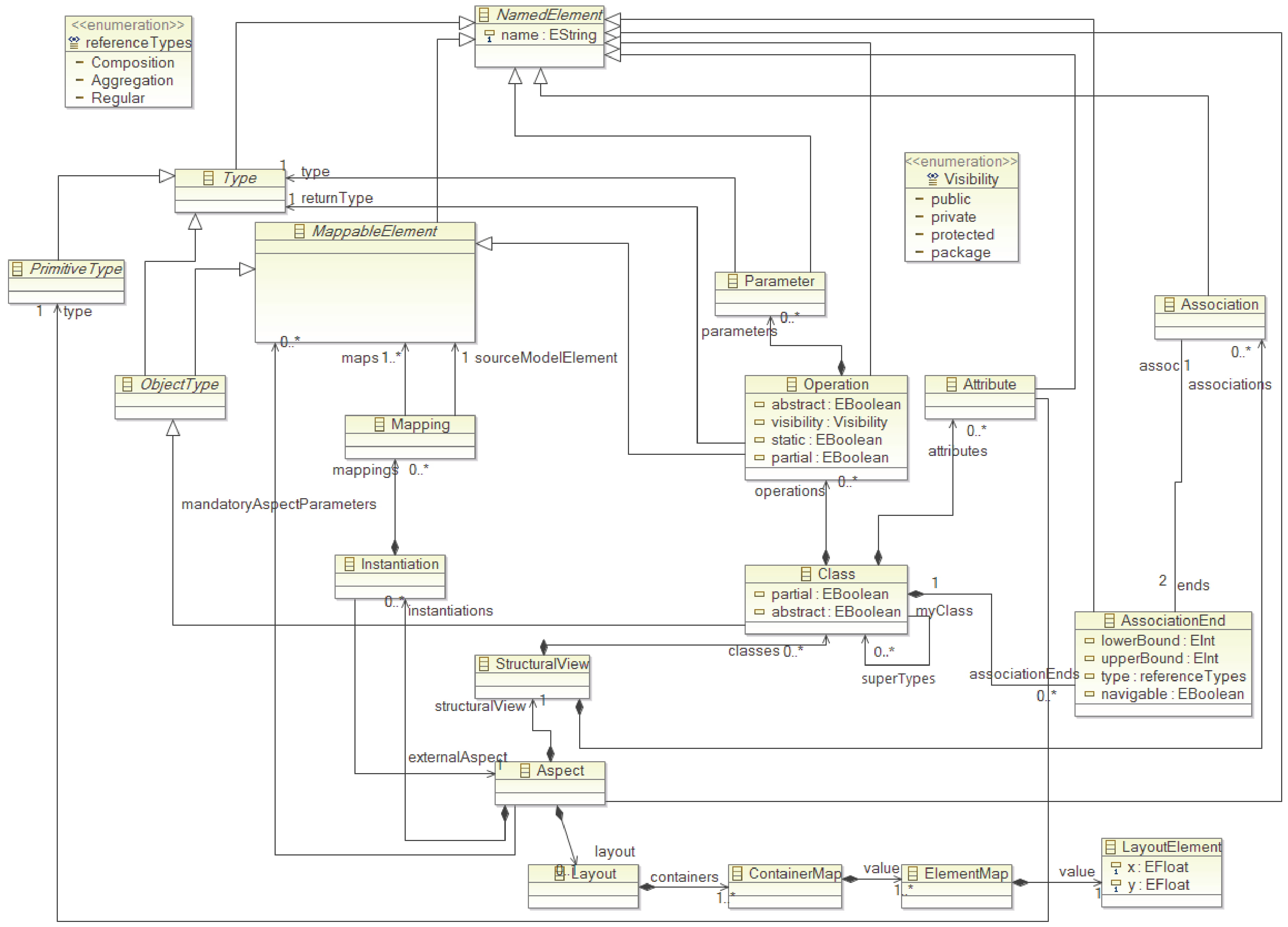
Intermediate Workflow Metamodel - Global View



Intermediate Workflow Metamodel - Stub View



RAM Metamodel



<<enumeration>>
referenceTypes
Composition
Aggregation
Regular
NamedElement
name : EString
Type
PrimitiveType
ObjectType
MappableElement
Parameter
Operation
abstract : EBoolean
visibility : Visibility
static : EBoolean
partial : EBoolean
<<enumeration>>
Visibility
public
private
protected
package
Association
Attribute
AssociationEnd
lowerBound : EInt
upperBound : EInt
type : referenceTypes
navigable : EBoolean
Class
partial : EBoolean
abstract : EBoolean
Mapping
Instantiation
StructuralView
Aspect
Layout
ContainerMap
ElementMap
LayoutElement
x : EFloat
y : EFloat
1 type
1 returnType
1 type
0..*
maps 1..*
1 sourceModelElement
parameters 0..*
mappings 0..*
mandatoryAspectParameters
operations 0..*
attributes 0..*
assoc 1
associations 0..*
2 ends
1 myClass
associationEnds 0..*
classes 0..*
0..* superTypes
0..* instantiations
structuralView 1
externalAspect 1
layout
containers 1..*
value 1..*
value 1