Executable meta-modeling in Kermeta with a rpg formalism

Ward Loos,
University of Antwerp,
ward.loos@student.ua.ac.be

Abstract

In this paper I report on my experience in implementing a role-playing game formalism using the executable meta-language Kermeta and converting AToM^3 models into Kermeta models. I also reflect on the differences and similarities between kermeta and the tools used in the Model driven engineering course for domain-specific modeling.

Keywords: modeling, meta-modeling, kermeta, atom3, ecore, domain-specific modeling

1. Introduction

After implementing a role-playing game formalism with different domain-specific modeling techniques and tools, I report on my findings using the executable meta-modeling language kermeta and its workbench. These previous experiences include metaDepth and AToM^3 with coded and rule-based operational semantics.

Using different tools also implicates different output formats and files. I implemented a conversion tool to convert models generated by AToM^3 into valid models for Kermeta.

All the code mentioned in this report is available so the reader can try out some things himself.

In section 2 we take a look at the Kermeta language and framework, section 3 details how a role-playing game formalism can be implemented in kermeta, section 4 compares some other tools/frameworks with kermeta, section 5 explains how we can convert atom3 models to kermeta models and finally section 6 summarizes my findings on kermeta.

2. Kermeta

2.1. Kermeta language

Kermeta Muller et al. (2005) aims to be a common denominator of modeling languages which allows it to describe both the structure and behavior of models (as seen in Figure 1). In other words, it's a meta-modeling language and an action language. Kermeta extends the Essential Meta-Object Facilities (EMOF) OMG (2006) language from the OMG[1] and is compliant with the Object Constraint Language (OCL) OMG (2012).

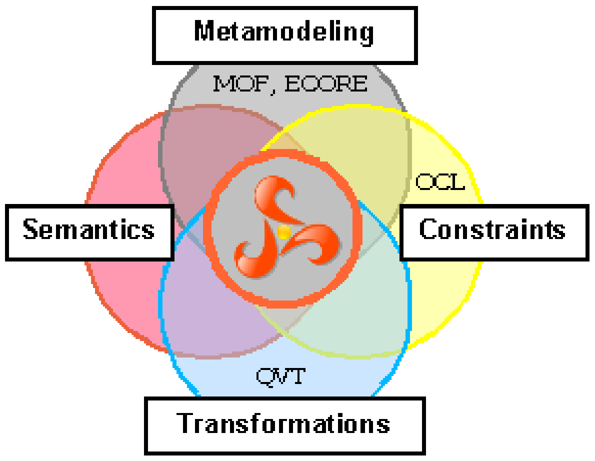


Figure 1: Kermeta positioning

Kermeta follows primarily the object-oriented paradigm. It has support for (abstract) classes and methods, properties, multiple inheritance, exceptions, generics (templates), namespaces (packages), ... However, kermeta also supports aspect-oriented programming and design by contract (invariants and pre- and postconditions). The aspect-oriented features allow you to weave in code for implementing something like, for example, a tracer or a logger. It's also statically typed. Something is does lack are constructors. This choice was made to remain compliant with the MOF specification.[2] If you really need a constructor, you could always use a method init(...) or something of that form. Some features I found kermeta lacked is an elsif

[1]Object Management Group - `http://www.omg.org`
[2]`http://www.kermeta.org/documents/faq/constructors`

keyword and no return, break or continue keywords. These omissions sometimes really limit your options in controlling the flow through your program or vastly worse the cyclomatic complexity.

Kermeta supports model navigation expressions including, but not limited to, expressions defined in the OCL which are implemented as lambda expressions. Associations with multiplicities between objects are defined in the form of attributes and references. An attribute is a link with containment (composition in UML) whereas a reference is just a link (one way association in UML). Opposite properties allow you to create a two-way link between objects (e.g. A.a#b and B.b#a).

Kermeta is implemented on top of the Java language. Internally it uses Java types and functions to implement the language. Kermeta gives you the possibility to import Java code in Kermeta or to use models in Java code. It is interpreted by Java code when you run a kermeta program.

2.2. Models

The meta-model you create is saved as a kermeta specific file (.kmt). However, to be able to create instance models you need a meta-model formatted in Ecore (.ecore) which is an UML dialect specified by Eclipse. The created meta-models are stored in the XML Metadata Interchange (XMI) OMG (2011) format. Kermeta relies on the Eclipse Modeling Framework (EMF) Eclipse (2013) to nicely display these models. This is a standard for exchanging metadata information via Extensible Markup Language (XML). This can be used for metadata whose meta-model can be expressed in MOF, and by extension, in kermeta. The EMF is also used to create instance models which are also XMI files and stored as an .xmi file. Ecore needs a root element in your meta-model to be able to generate a model. This implicates that an element only can be defined as a child of another element (an attribute) or as the root element.

Note that EMF is not really necessary because all the files are plain XML. If you want, you can create the XML files manually but this would only be viable for very small (meta-)models.

2.3. Platform

Kermeta can be downloaded as a plugin for the Eclipse IDE (available from the marketplace). The plugin provides syntax highlighting, type checking, an interpreter and a visual debugger. The plugin also provides some additional menu options: see dependent files, see file dependencies, generate

Ecore and Generate Km (Kermeta Model). The first 2 options can be pretty helpful when dealing with complex models. Especially see dependent files; it allows you think about what a change in your model could cause. Generate Ecore allows you to convert your kmt meta-model to an ecore file. This file is needed when working in EMF and when loading a resource (model) in your code.

The EMF is also required so I will describe it here as a part of the kermeta platform. It allows you to open ecore files and view its contents in a more visual way than just displaying XML code (see section 3.3) EMF lets you generate an ecore diagram (.ecorediag) from an ecore file. This looks like an UML diagram and gives a better view on the structure you generated. Models can be created in EMF without writing any code. You create a dynamic instance of your root element and add children to it. If opposite properties are set, you will only have to set the association in one element and EMF will fill in the other one. EMF also provides a list of elements suitable for the association you are trying to fill in.

Note that it is possible to use different modeling tools than EMF. As long as they support XMI (and ecore) you should be fine. I didn't test this myself because I had no reason to do so. I imagine this could be very helpful when working with kermeta having prior experience in another tool than EMF.

The combination of kermeta and EMF basically enables you to take 2 approaches for coming to a solution. A coding approach in which you implement the meta-model in a kermeta source file (.kmt) and implement the action code as well. The ecore file is then only used to create and verify models. Second approach provides a more high-level way and would probably be preferable over the first. Design your solution first in an ecore diagram, generate an ecore file and finally generate a kmt file. Then you can fill in the necessary behavior code. In the first approach you would probably make an UML diagram anyway.

I went with the first approach because the kermeta documentation is pretty sparse on information about EMF. So at first, it wasn't very clear to me what role it played. In addition to that, I could rely on the code from the assignments we had to make for the course (see section 4) and just translate the code.

A kermeta program can be be run directly from eclipse. There is also a standalone package available which lets you run kermeta programs from the command line as follows:

```
java -jar kermeta_standalone.jar -U filename -C package::Class -O operation
```

2.4. Kermeta 2

Version 2 was released somewhere in 2012. It introduces a central configuration file and forces some minor differences in the code. [3] There are probably also changes in the core framework. The main difference is under the hood. Kermeta can now be compiled to Scala byte code which introduces a massive speed improvement over the interpreter used in kermeta 1.

I chose to stick with version 1 (after starting with version 2) because most of the documentation is targeted at version 1. This doesn't matter too much though because the differences are small. The eclipse plugin however regressed and lacks some features. Generating ecore files from kmt for example is missing. This removes a lot of flexibility. Probably not a problem for a experienced kermeta developer but it presented me with a problem. The role-playing game (RPGame) formalism is fairly simple and the models are not too big so speed isn't a factor for me (checking invariants however is already slow).

3. Implementing the role-playing game formalism in Kermeta

3.1. Role-playing game formalism

Short overview of the rpg formalism used in the implementation. This formalism is covered in greater detail in section 3.2

A game can contain one or multiple scenes which contain one or multiple tiles themselves. There is one hero character and multiple villains. All the characters are placed on a tile. Each turn the hero can move or attack (whatever is possible) and villains in the same scene can do the same. A hero can also pickup items on tiles like keys for doors, weapons and goals. Tiles can also The game ends when the hero collects all goals or dies. Moving between scenes can be achieved by going through a door.

3.2. Meta-model

3.2.1. Files

You can find the code discussed in section 3.2 in metamodel/RPGame.kmt, metamodel/RPGame.ecore and metamodel/RPGame.ecorediag

[3] `http://www.kermeta.org/documents/user_doc/migrationguide`

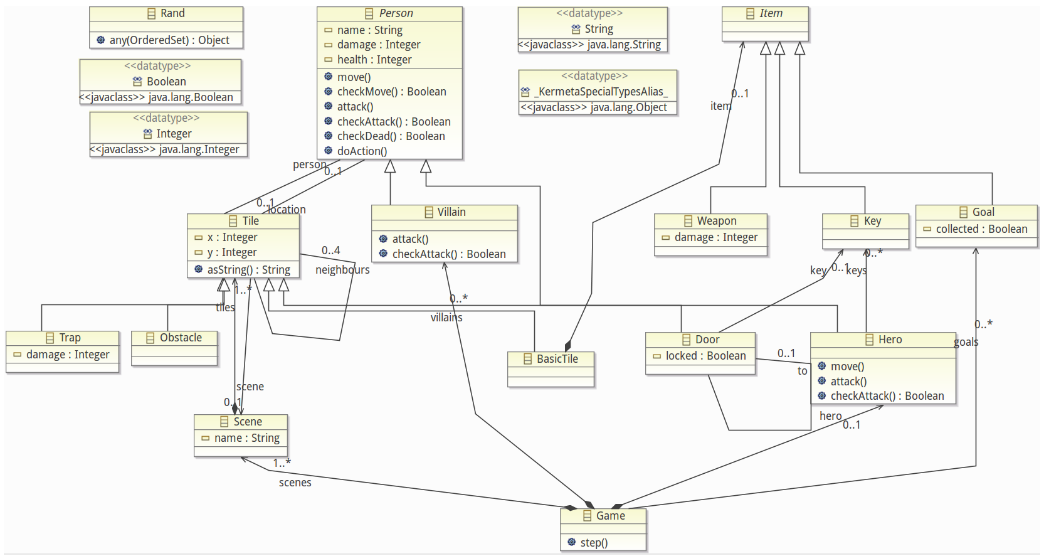


Figure 2: RPGame Ecore diagram

3.2.2. Game

Game is the root element of our formalism. It holds the scenes, the hero and the villains. It also keeps a references to the goals. This is necessary to be able to determine if the hero has won. We need at least one scene, one goal and exactly one hero. Villains are optional. The operation step() implements one tick in the game in which the hero and villains in the same scene as the hero get do to an action (move or attack). This will typically be called in a loop which keeps running until the game finishes.

```
class Game {

attribute scenes : oset Scene[1..*]
attribute hero : Hero
attribute villains : oset Villain[0..*]
reference goals : oset Goal[1..*]

operation step() is do ... end
}
```

3.2.3. Scene

A scene defines the setting in our game (e.g. Castle, Forest, City, ...). It holds one or more tiles which define the size of the scene. Scene has also a name which can be used for pretty printing (this attribute is also accessible in tile objects so they can print their location).

```
class Scene {

attribute name : String
attribute tiles : oset Tile[1..*]
}
```

3.2.4. Tile

Tile is the most basic of tiles. It has attributes x and y for a notion of position. It also holds a reference to its scene, neighbours and possibly a person standing on it. The neighbours reference is mostly useful for the person standing on the tile so it can determine what actions it can do. The operation asString() returns the tile represented as a string in the form: SceneName$< x, y >$

```
class Tile {

attribute x : Integer
attribute y : Integer
reference scene : Scene
reference neighbours : oset Tile[0..4]#neighbours
reference person : Person#location

operation asString() : String is do ... end
}
```

BasicTile inherits from Tile and adds to possibility to place an item on a tile which then can be picked up by the hero.

```
class BasicTile inherits Tile {

attribute item : Item
}
```

Door inherits from Tile and allows the hero to move between scenes. A door can be locked which prevents the hero from going through it. However, the hero can unlock the door if he has picked up the key to which the door references. The attribute locked and the reference key are not linked to eachother so there aren't any conditions. Every combination is possible. For example, a door can be locked but not have a key set. This means the hero will never be able to go through the door. This could useful in forcing the hero through the scenes. The invariant doorCheck enforces that a door only links to a door in another scene.

```
class Door inherits Tile {

attribute locked : Boolean
reference key : Key
reference to : Door#to

inv doorCheck is do self.scene != self.to.scene end
}
```

Trap inherits from Tile and can deal damage to the hero when he moves onto the trap.

```
class Trap inherits Tile {

attribute damage : Integer
}
```

Obstacle inherits from Tile and doesn't have any attributes or methods defined. A person cannot move on an obstacle tile. Despite of not implementing anything, it is still useful to be able to check for obstacle tiles by using

```
tile.isKindOf(Obstacle)
```

. The invariant obstacleCheck allows us to enforce that a person cannot be on an obstacle tile.

```
class Obstacle inherits Tile {

inv obstacleCheck is do self.person == void end
}
```

3.2.5. Person

Person is an abstract class that represents a character in the game. A person has a name, health and damage he can deal. It also holds a reference to the tile it's standing on. The class is abstract because of the abstract operations attack() and checkAttack(). The reason they are abstract is because you don't want to attack someone of your own type: hero → hero (in practice not possible because multiple heroes would violate our meta-model) or villain → villain in our formalism. Operation checkAttack() should return true if there is a valid target on one of the neighbouring tiles. Attack() should execute an attack move on one of the viable targets. Operations checkMove() and move() basically do the same but for moving, i.e., check for valid move locations and execute a move between tiles. checkDead() should be called every time a person takes damage. They are not abstract however. It takes care of printing some info to the console. Operation doAction() simply chooses at random between attack or move based on the results of checkAttack() and checkMove().

```
abstract class Person {

attribute name : String
attribute damage : Integer
attribute health : Integer
reference location : Tile#person

operation move() is do ... end
operation checkMove() : Boolean is do ... end

operation attack() is abstract
operation checkAttack() : Boolean is abstract

operation checkDead() : Boolean is ... do end

operation doAction() is do ... end
}
```

Hero inherits from Person and adds a reference to the keys he picked up during the game. The operation move from Hero is overloaded by using the method keyword. First, the operation move from Person is called by executing super. Further, some code is added to handle visiting different kind of tiles. In case of a door, if it's locked we try it to unlock with the hero's collected keys. If the door is unlocked the hero moves through to the tile in the .to reference. If it's a trap, the trap's damage is subtracted from the hero's health. With a basictile we check for an item. A key is picked up and added to the set of keys, a weapon's damage is added to the hero's damage and a goal is collected.

```
class Hero inherits Person {

reference keys : oset Key[0..*]

method move() from Person is
do
super
if isKindOf(Door) then do ... end
if isKindOf(Trap) then do ... end
if isKindOf(BasicTile) then do ... end
end

method attack() is do ... end
method checkAttack() : Boolean is do ... end
}
```

Villain inherits from Person and just implements the necessary code in attack() en checkAttack(). The differences between the code in villain and hero are minimal. Only the selectors are adjusted. A villain can't do anything with items so the we don't need to overload the move() operation.

```
class Villain inherits Person {

method attack() is do ... end
method checkAttack() : Boolean is do ... end
}
```

3.2.6. Item

Item serves as an abstract base class for all other items to inherent from.

```
abstract class Item {

}
```

Key inherits from item. It doesn't need any attributes or references because doors themselves keep the information which key they need.

```
class Key inherits Item {

}
```

Weapon inherits from item and has one attribute damage. This damage is added to the hero's damage when he finds the weapon. The weapon is then removed from the game.

```
class Weapon inherits Item {

attribute damage : Integer
}
```

Goal inherits from item and has one attribute collected. When the hero finds the goal the attribute is set to true. The game element has a reference to each goal so it can check after every step() if the hero has won the game by collecting every goal.

```
class Goal inherits Item {

attribute collected : Boolean
}
```

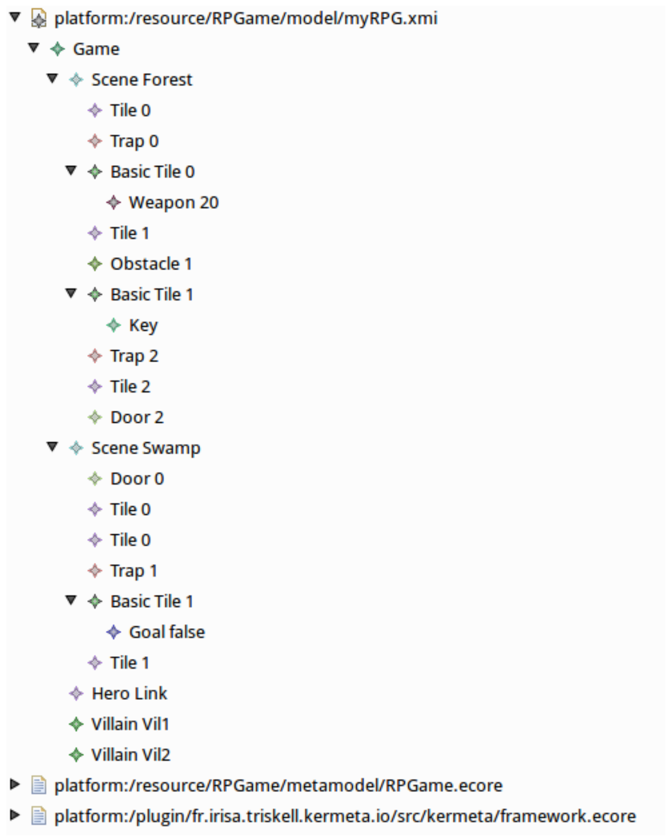
3.3. Example model

Below is a sample model consisting of 2 scenes Forest and Swamp. There are 2 villains and there is one goal.
Legend: H hero, T trap, w weapon, OOO obstacle, V villain, k key, D door

```
Forest:

  0   1   2
|---|---|---|
| H | T | w | 0
|---|---|---|
|   |OOO|V,k| 1
|---|---|---|
| T |   | D | 2
```

```
|---|---|---|

Swamp:

  0   1   2
|---|---|---|
| D | V |   | 0
|---|---|---|
| T | g |   | 1
|---|---|---|
```

Figure 3 shows how the model looks like when opened in eclipse.



```
▼ platform:/resource/RPGame/model/myRPG.xmi
  ▼ Game
    ▼ Scene Forest
        Tile 0
        Trap 0
      ▼ Basic Tile 0
          Weapon 20
        Tile 1
        Obstacle 1
      ▼ Basic Tile 1
          Key
        Trap 2
        Tile 2
        Door 2
    ▼ Scene Swamp
        Door 0
        Tile 0
        Tile 0
        Trap 1
      ▼ Basic Tile 1
          Goal false
        Tile 1
      Hero Link
      Villain Vil1
      Villain Vil2
▶ platform:/resource/RPGame/metamodel/RPGame.ecore
▶ platform:/plugin/fr.irisa.triskell.kermeta.io/src/kermeta/framework.ecore
```

Figure 3: RPGame model with 2 scenes

Opening this model with a text editor shows the xml. The xml is very readable and easy to understand.

```
<?xml version="1.0" encoding="ASCII"?>
<RPGameMeta:Game
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:RPGameMeta="platform:/resource/RPGame/metamodel/RPGame.ecore#/"
    xsi:schemaLocation="platform:/resource/RPGame/metamodel/RPGame.ecore#/
    ../metamodel/RPGame.ecore#/1"
    goals="//@scenes.1/@tiles.4/@item">
  <scenes name="Forest">
    <tiles x="0"
        y="0"
        scene="//@scenes.0"
        neighbours="//@scenes.0/@tiles.1 //@scenes.0/@tiles.3"
        person="//@hero"/>
    <tiles xsi:type="RPGameMeta:Trap"
        x="0"
        y="1"
        scene="//@scenes.0"
        neighbours="//@scenes.0/@tiles.0 //@scenes.0/@tiles.2
        //@scenes.0/@tiles.4"
        damage="10"/>
    <tiles xsi:type="RPGameMeta:BasicTile"
        x="0"
        y="2"
        scene="//@scenes.0"
        neighbours="//@scenes.0/@tiles.1 //@scenes.0/@tiles.5">
      <item
          xsi:type="RPGameMeta:Weapon"
          damage="20"/>
    </tiles>
    <tiles x="1"
        y="0"
        scene="//@scenes.0"
        neighbours="//@scenes.0/@tiles.0 //@scenes.0/@tiles.4
        //@scenes.0/@tiles.6"/>
    <tiles xsi:type="RPGameMeta:Obstacle"
        x="1"
        y="1"
        scene="//@scenes.0"
        neighbours="//@scenes.0/@tiles.1 //@scenes.0/@tiles.3
        //@scenes.0/@tiles.5 //@scenes.0/@tiles.7"/>
    <tiles xsi:type="RPGameMeta:BasicTile"
        x="1"
        y="2"
        scene="//@scenes.0"
        neighbours="//@scenes.0/@tiles.2 //@scenes.0/@tiles.4
        //@scenes.0/@tiles.8"
        person="//@villains.0">
      <item
          xsi:type="RPGameMeta:Key"/>
    </tiles>
    <tiles xsi:type="RPGameMeta:Trap"
        x="2"
        y="1"
        scene="//@scenes.0"
        neighbours="//@scenes.0/@tiles.3 //@scenes.0/@tiles.7"
        damage="10"/>
    <tiles x="2"
        y="1"
        scene="//@scenes.0"
        neighbours="//@scenes.0/@tiles.6 //@scenes.0/@tiles.4
        //@scenes.0/@tiles.8"/>
    <tiles xsi:type="RPGameMeta:Door"
        x="2"
        y="2"
        scene="//@scenes.0"
        neighbours="//@scenes.0/@tiles.7 //@scenes.0/@tiles.5"
        locked="true"
        key="//@scenes.0/@tiles.5/@item"
        to="//@scenes.1/@tiles.0"/>
  </scenes>
  <scenes name="Swamp">
    <tiles xsi:type="RPGameMeta:Door"
        x="0"
        y="0"
        scene="//@scenes.1"
        neighbours="//@scenes.1/@tiles.1 //@scenes.1/@tiles.3"
        locked="true"
        to="//@scenes.0/@tiles.8"/>
    <tiles x="0"
        y="1"
        scene="//@scenes.1"
        neighbours="//@scenes.1/@tiles.0 //@scenes.1/@tiles.2
        //@scenes.1/@tiles.4"
        person="//@villains.1"/>
    <tiles x="0"
        y="2"
        scene="//@scenes.1"
        neighbours="//@scenes.1/@tiles.1 //@scenes.1/@tiles.5"/>
    <tiles xsi:type="RPGameMeta:Trap"
        x="1"
        y="0"
        scene="//@scenes.1"
        neighbours="//@scenes.1/@tiles.0 //@scenes.1/@tiles.4"
        damage="10"/>
    <tiles xsi:type="RPGameMeta:BasicTile"
        x="1"
        y="1"
        scene="//@scenes.1"
        neighbours="//@scenes.1/@tiles.3 //@scenes.1/@tiles.1
        //@scenes.1/@tiles.5">
      <item
          xsi:type="RPGameMeta:Goal"
          collected="false"/>
    </tiles>
    <tiles x="1"
        y="2"
        scene="//@scenes.1"
        neighbours="//@scenes.1/@tiles.4 //@scenes.1/@tiles.2"/>
  </scenes>
```

```
  <hero name="Link"
      damage="30"
      health="900"
      location="//@scenes.0/@tiles.0"/>
  <villains
      name="Vil1"
      damage="20"
      health="100"
      location="//@scenes.0/@tiles.5"/>
  <villains
      name="Vil2"
      damage="25"
      health="100"
      location="//@scenes.1/@tiles.1"/>
</RPGameMeta:Game>
```

3.4. Running the example

You need the Eclipse IDE with the Eclipse Modeling Tools and Kermeta Workbench (version 1).
Easiest way is to follow these steps:

1. Download the Eclipse Modeling Tools version of Eclipse from `http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/junosr1`
2. Install Kermeta Workbench from the Eclipse Marketplace (Help → Eclipse Marketplace)
3. Create a new Kermeta project called RPGame
4. Copy the contents of the RPGame folder delivered with this report into your kermeta project folder
5. Right click on run.kmt → Run As → Run Configurations → goto tab Java Classpath
6. Left click on User Entries → Add External JARs
7. Browse to your eclipse installation → plugins → org.kermeta.language.mdk_1.4.0.jar
8. Now click Run

Bug: If you regenerate the .ecore meta-model file from RPGame.kmt, one of the types is not correct. You should search for the following line:

```
<eClassifiers xsi:type="ecore:EDataType" name="Integer"
instanceClassName="java.lang.Object">
```

And replace it with:

```
<eClassifiers xsi:type="ecore:EDataType" name="Integer"
instanceClassName="java.lang.Integer">
```

4. Comparison with other domain-specific modeling techniques

4.1. metaDepth

Metadepth de Lara and Guerra (2010) describes itself as a framework for multi-level meta-modeling. It takes more of a procedural approach. It uses nodes (similar to structs in C) and operations. The structural part of meta-models and models themselves are defined in metadepth. Metadepth depends on the Eclipse Object Language (EOL) for adding behavior to meta-models. Similar to kermeta, metadepth (or rather EOL) makes use of the OCL for model navigation. Kermeta and metadepth are actually surprisingly similar despite the difference in paradigm. If the metadepth nodes changed in name to classes and the operations would become methods you would almost have kermeta code. They are also both interpreted by Java code. The syntax for associations is exactly the same and invariants can be defined and checked in a similar way. One advantage metadepth is that you can all instances anywhere in the action code. In kermeta you can only access the root element and go from there through your model. Kermeta however has a clear advantage is tool support. Having a (visual) debugger can be of great value and save you a lot of time. This even becomes more important if your (meta-)models become more complex. Because kermeta uses xmi to store its models, you can also use a variety of other modeling tools.

4.2. AToM^3

AToM^3 is a very flexible tool which supports a vast number modeling techniques. In case of coded operational semantics in atom3, you can take a very similar approach for implementing the rpgame formalism in kermeta. Draw an ecore diagram in EMF with the correct associations, methods and constraints. From that ecore diagram you can generate an ecore meta-model from which you can finally generate a kermeta meta-model. One of the key differences and strengths of AToM^3 is that you can generate of nice visual representation of your models. AToM^3 also gives you the possibility to move further away from a coding approach by rule-based semantics. The stability of the tool however remains a problem, as well a lacking some basic features like undo. It has some basic debugging tools built-in but because it runs directly in python, any python debugger could be used.

5. Converting AToM^3 models into Kermeta models

5.1. Implementation

To be able to reuse models in kermeta which where created in AToM^3, I created some python code to write out AToM^3 models to xmi. I made use of the built-in python library xml.etree.ElementTree to achieve this.

For starters, I implemented the rpgame formalism in python which closely follows the kermeta implementation.[4] Naturally, no operational semantics are implemented because we only want to mimic the abstract syntax. Each class has at least a method getXml(rootElement) which returns an xml element. The parameter rootElement allows to attach the new element to a root element which allows us to create the necessary nesting. For example, Scene.getXml():

```
def getXml(self, rootElement):
  e = SubElement(rootElement, "scenes")
  e.set("name", self.name)
  return e
```

Classes which extend another class don't have to re implement getXml() completely. They first call getXml() of their parent through super and use the returned element to set some extra attributes. For example, Trap which extends Tile:

```
def getXml(self, rootElement):
  e = super(Trap, self).getXml(rootElement)
  e.set("xsi:type", "RPGameMeta:Trap")
  e.set("damage", self.damage)
  return e
```

Another useful method which most classes have is getModelPath(). This method returns the path of an element in a model in the form of a string, e.g. for a tile: //@scenes.0/@tiles.2. This provides an easy way for objects

[4]This code can be found in dataTypes.py

to get and store these paths without having to do anything themselves. For example, tile keeps its neighbours in an array of strings. When getXml() is called it can simply do

```
" ".join(self.neighbours)
```

.

The basic idea is also that each object keeps track of the objects that need to be nested in the xmi document. This way we can generate the document by nesting for-loops as we'll see later.

The actual conversion is done in the class XmiGenerator.[5] First, we parse all the non-link elements from ASGroot by calling the method genObj(). We store each element in a list and create a corresponding dataTypes object which is also stored in a list. For example:

```
  def getTraps(self):
    traps = self.asgroot.listNodes["Trap"]
    self.tiles.extend(traps)
    for tile in traps:
      self.tilesObj.append(Trap(tile, len(self.tilesObj)))
```

Storing the ASGroot element serves no other purpose than to be able to get the correct object when parsing links. This works because they have the same index in both lists. Example:

```
  def getSceneObj(self, scene):
    return self.scenesObj[self.scenes.index(scene)]
```

Parsing all the different links in done by calling solveLinks(). Again, there is a basic idea that is applied for each type. A link is always between 2 elements (in our formalism anyway). We use each element to get the corresponding object. Then we can execute the necessary actions to enable the same link in the kermeta model. Example:

```
  def personOnTileLinks(self):
    pOTLs = self.asgroot.listNodes["PersonOnTileLink"]
```

[5]This code can be found in genXMI.py

```
  for pOTL in pOTLs:
    person = pOTL.in_connections_[0]
    tile = pOTL.out_connections_[0]
    personObj = self.getPersonObj(person)
    tileObj = self.getTileObj(tile)
    tileObj.person = personObj.getModelPath()
    personObj.location = tileObj.getModelPath()
```

Finally we can call genXMI(). The actions described above are all called in the constructor so this is only method that a user of the class should call. An object keeps track of its direct subelements so we can create most of the xmi document by nesting for loops.

```
def genXMI(self):
  gameXml = self.gameObj.getXml()
  for sceneObj in self.scenesObj:
    sceneXml = sceneObj.getXml(gameXml)
    for tileObj in sceneObj.tiles:
      tileXml = tileObj.getXml(sceneXml)
      try:
        itemObj = tileObj.item
        itemXml = itemObj.getXml(tileXml)
      except Exception:
        pass
  heroXml = self.heroObj.getXml(gameXml)
  for villainObj in self.villainsObj:
    villainXml = villainObj.getXml(gameXml)

  return gameXml
```

Only BasicTile objects have an item attribute but the rest of the objects in the tiles' list not. In addition to that, we don't know if the item is actually set. Therefor, this code is placed in a try - except block.

5.2. Running AToM^3

To test our implementation you should use the included AToM^3 version. We encountered a bug (for our implementation anyway) in AToM^3 which is fixed in the included version. Testing can be done with the following model:

`atom3/User Models/RPGame_Models/ForestSwampCastleWithAssociations_RPGame_MDL.py`

Use the following GraphGrammar:

`atom3/User Formalisms/RPGame/RPGTransformations_exec/RPG_GG_exec.py`

6. Conclusion

The kermeta framework is very stable with its future cut out. Version 2 is already released and promises improved performance over version 1 by allowing compilation. Using standardized techniques by the OMG, it automatically opens its framework up for other modeling tools to be used to manipulate both models and meta-models.

Combining the abstract syntax and behavioral code feels very natural for programmers coming from a non modeling background.

Converting models from AToM^3 to Kermeta proved to be fairly easy and painless. However this could be made even simpler if AToM^3 would provide a default export of its models to xmi. This would allow the use of modeling tools, such as kermeta, to execute model transformations instead of relying on normal python code to do the task.

References

de Lara, J., Guerra, E., 2010. Deep meta-modelling with metadepth.
URL `http://astreo.ii.uam.es/~jlara/metaDepth/papers/TOOLS.pdf`

Eclipse, 2013. Eclipse modeling framework project.
URL `http://www.eclipse.org/modeling/emf/`

Muller, P., Fleurey, F., Vojtisek, D., Drey, Z., Pollet, D., Fondement, F., Studer, P., Jzquel, J., 2005. On executable meta-languages applied to model transformations.
URL `http://www.irisa.fr/triskell/publis/2005/Muller05c.pdf`

OMG, 2006. Mof 2.0 core specification.
URL `http://www.omg.org/spec/MOF/2.0/`

OMG, 2011. Mof/xmi mapping, version 2.4.1.
URL `http://www.omg.org/spec/XMI/2.4.1/`

OMG, 2012. Object constraint language, version 2.3.1.
URL `http://www.omg.org/spec/OCL/2.3.1/`