From Role-Playing Game to Petrinet, the ATL way.

Daan Janssens

daan.janssens@student.uantwerpen.be

University of Antwerp

Abstract

Model driven engineering helps increasing productivity by maximizing compatibility, simplifying the design process and promoting communication. The field of game development is no exception to this. Additionaly, one can use these models and model-to-model transformations to inspect specific properties of their game. In this paper we use ATL (ATL Transformation Language), a model transformation language and toolkit developed on top of the Eclipse platform, to model a role-playing game and transform it stepwise into a petrinet for further analysis and requirements verification. In this way, we tend to explore the capabilities of the ATL language and it's use in game development. We also present a comparison of the discovered advantages and dissadvantages with other transformation languages.

Keywords: ATL, Eclipse Modeling Project, Model-Driven Engineering, RPG, Petrinet

1. Introduction

The field of Model-Driven Engineering (MDE) aims to consider models as first class entities, where model transformation are key for model handling. Domain Specific Languages (DSL) are created in order to cope with the increasing complexity of a specific problem domain. The logic behind Computer games (more specific: role-playing games) is a perfect example of this complexity. By creating a role-playing game Domain Specific Language, we can create basic RPG game models. These models help the design process and promote communication. They can also be used for code generation and verification of various game properties. Without these models, some verifications can only be performed through extensive and costly tests.

In this paper we look at the completability of an RPG game (can the game be succesfully finished?). This can be verified by transforming the RPG model into a petrinet model and exploring the state space of it. We use ATL, a model transformation language and toolkit developed on top of the Eclipse platform, which provides ways to produce a set of target models from a set of source models.

The transformation from RPG to Petrinet happens stepwise. First an RPG model which is conform to the RPG metamodel is transformed into a reduced RPG model, where features that are not helpful for the analysis are removed. This reduced model is conform to a reduced RPG metamodel. Next, the reduced model is transformed into a Petrinet model, which is conform to the Petrinet metamodel. Lastly, we transform our petrinet into a model that is 'readable' by a Petrinet tool called Pipe.

The remaining of this paper is organized as follows: Section 2 discusses the related work. Section 3 gives an introduction to the basics of ATL and the Eclipse modeling framework. Section 4 represents our analysis and introduces the RPG formalism. Section 5 explains our approach in greater detail. Section 6 discusses our design of the metamodels. We present an initial RPG model and the implementation of our transformations in section 7. Section 8 shows our results of performing the transformations on the inital model. Lastly, In section 9 we state our conclusion and future work. We also try to compare ATL with other transformation languages.

2. Related Work

2.1. Games & Petrinets

The idea behind our research is not entirely unique. Game related domain specific languages have been proposed and discussed before by Walter and Masuch (2011), Furtado and de Medeiros Santos (2006), Reyno and Carsí Cubel (2009). However they focused mainly on the code generation part and not the analysis part. Petrinets have been used in the past to verify specific features of games. Brom and Abonyi (2006) and Araújo and Roque (2009) focussed on the interactions between different actors and visualizing the plot of a story between them with petrinets. It however does not use an RPG model or transformation. The work of Marques et al. (2012) comes closest to ours, they use ATL to transform a RPG model into Lua code and transform the RPG into a petrinet to check if the goal can be reached. However, they abstracted their petrinet a lot more than in our work, which makes it less visual and less useful to communicate with.

2.2. ATL

The ATL model transformation language has been studied extensively before. The ATL basics and syntax/semantics are thoroughly discussed in the work of Jouault et al. (2008), Bzivin et al. (2003). An overview of the support in eclipse, the tool which we also used is given by Allilaire et al. (2006).

3. ATL

3.1. Syntax and semantics

ATL follows a specific transformation pattern: a read-only source model Ma gets transformed to a write-only target model Mb. This happens according to a transformation definition MMa2MMb, which is written in ATL. This transformation definition itself can be seen as a model which conforms to the ATL metamodel, while Ma and Mb conform respectively to the metamodels MMa and MMb. All metamodels conform to the MOF metamodel.

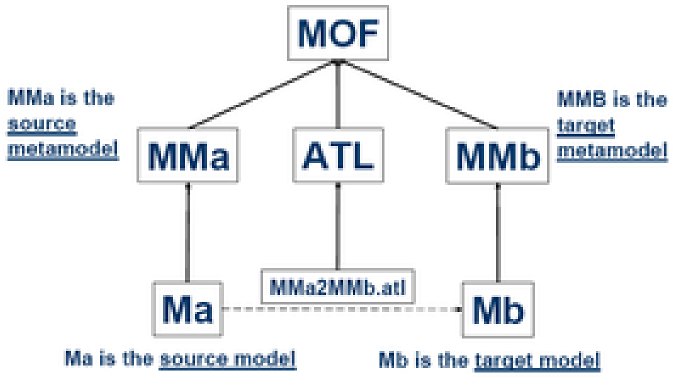


Figure 1: The transformation pattern

A transformation definition bundles one or multiple rules. Those rules are the heart of ATL transformations(Allilaire et al. (2006)) because they describe how output elements are produced from an input element. They can be specified in a declarative or imperative way. The declarative approach is most of the time encouraged (Jouault et al. (2008)) and is also used in our approach.

A matched rule is composed of a source pattern and a target pattern. The source pattern (also called inPattern (Bzivin et al. (2003))) specifies a set of source types (from the source metamodel) and possibly a boolean OCL expression (guard). The ATL engine will try to find a set of matches of this pattern in the source model. The target pattern (outPattern) is composed of a set of elements, where every element specifies a target type (from the target metamodel) and a set of bindings. When a rule containing the target pattern is executed, the target elements of the specified types are created. The binding will specify the value used to initialize the properties.

By executing rules on a match, a dynamic traceability link will be automatically created in the internal strutures of the transformation engine. This link relates three components: the rule, the match and the newly created target elements. These bindings can be queried by the use of the *thisModule.resolveTemp(obj,'nameofelement')*, which allows you to refer to created target elements. This has been extensively used in our approach.

We also used ATL helper functions, which are written in OCL. They can navigate across model elements, though only on the read-only source model side. ATL also offers rule inheritance (the new rule can specify additional elements or restrictions), however according to **?** the support of it is still limited. For that reason we decided not to use it in our approach.

The most important (and confusing at the start) aspect of ATL is that each source element can be used only once as an inPattern for a rule. This caused quite some problems with our inital RPG model, since we designed it in a way that each Tile contained four neighboring Tiles. A solution to this was the creation of a Connector element, that connected two Tiles. We cover this further in this paper.

3.2. The ATL Eclipse modeling environment

Installing and getting acquainted with the modeling environment is not an easy job. There is a lot of outdated information on the internet, which can be misleading. The best way to get acquainted with the plugin is by watching a series of informative videos about it (abidredlove (2009)). Once it is up and running a new ATL project can be made. The basic workflow is very pragmatic; a metamodel is created and defined in the Ecore format, which is basically a sub-set of UML Class diagrams. From an Ecore model, you can create a dynamic instance, which is a file in XSI format conform to the metamodel. Lastly you start creating your ATL rules, for which Eclipse also offers syntax highlighting. For executing an ATL file, one has to specify the paths to the IN and OUT metamodel and model.

4. Analysis

4.1. RPG Formalism

Syntax and Static Semantics

1. An RPGame consists of a world that is divided into a number of scenes.
2. In each scene, there are a number of connected tiles.
3. Tiles can be connected in any direction (this is done by a connector element that connects two Tiles together). This way, a map is created for the scene.
4. In the game, there is one hero. The hero is always on exactly one tile.
5. A tile can be a 'standard' tile, or an obstacle, on which no character can stand.
6. On an "standard" tile, there can be an object.
7. Objects are: goals, keys and doors.
8. A door is a portal to a door on another scene.

Dynamic Semantics

1. A character can move from one tile to another conneted tile (provided it is not an obstacle)
2. A key and goal can be picked up by the hero by walking on its tile. They can only be picked up once.
3. The hero can pass through a door to enter another scene.
4. Door are locked, and the hero must pick up a particular key (for that door) to be able to enter it.
5. The hero wins if he can pick up all goals.
6. The game stops when the hero wins.

We decided to keep the hero as only character, additional villains/NPC's would make this example too complex and are not in our scope of main interest.

4.2. Petrinet Formalism

We used the standard syntax and semantics of a Place/Transition Petrinet. We refer the interested reader to the work of Wang (2007) for a look into the petrinet formalism.

5. Approach

The setup of our experiment is depicted in Figure 2. We started by creating a RPG Language Meta-Model (ecore model) which describes every possible feature of the defined RPG Domain. In our experiment this meta-model is still small, however it could contain a lot more features that are not relevant for analysis purposes (e.g. quest dialogs). Therefore we created a Reduced RPG domain (RRPG Language), an 'intermediate' language which will work as a filter. It will only contain the essential entities needed to check properties related to the petrinet analysis (e.g. it will not contain obstacles and connections to obstacles.) RPG models will be transformed by the *RPG2RRPG* transformation into a RRPG model.

In order to handle petrinet models, we had to create a Petrinet metamodel. A *RRPG2Petrinet* transformation transforms the RRPG model into a petrinet that is conform to the petrinet standards.

We decided to use the tool Pipe (PIPE2) to visualize our petrinets and perform the analysis. The format that Pipe is able to read however, is a bit different from the Petrinet metamodel. Here our last transformation (*Petrinet2Pipe*) is used. We transform a general petrinet model into a tool-specific model. In order to succeed in this, we also had to create a simplified Pipe metamodel.

Since our models are in a xmi format and Pipe is only able to read specific xml files, we had to come up with a small python script. This script only changes a few lines in the xmi file in order to make it work for pipe.

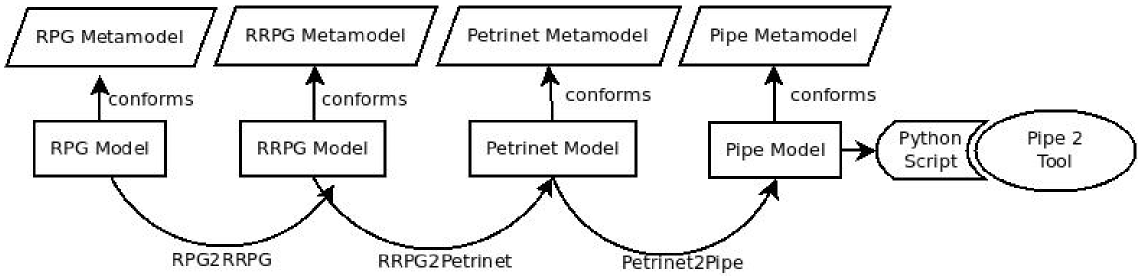


Figure 2: a depiction of our approach

6. Design

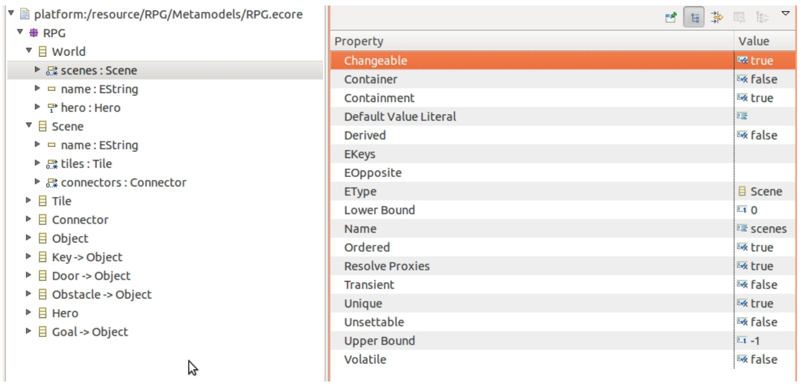
In this section we focus on the design of our metamodels (also known as abstract syntax). We explain how eclipse offers a way to create them and how you can create instance models from them.

6.1. Creating a metamodel

With the ATL plugin and EMF (Eclipse Modeling Framework) installed, one can easily create a new metamodel by selecting *new->other->eclipse modeling framework->ecore model.* Creating these metamodels is done in a text-based way. However the Eclipse modeling framework also offers a UML-alike drag and drop visual editor, which can be accessed if you *initiate an ecore diagram file* from the ecore file. Our metamodels are created this way.

6.2. RPG Metamodel

Figure 3 represents our RPG metamodel as ecore model, while Figure 4 gives a visual representation by showing it's related diagram file.



- platform:/resource/RPG/Metamodels/RPG.ecore
  - RPG
    - World
      - scenes : Scene
      - name : EString
      - hero : Hero
    - Scene
      - name : EString
      - tiles : Tile
      - connectors : Connector
    - Tile
    - Connector
    - Object
    - Key -> Object
    - Door -> Object
    - Obstacle -> Object
    - Hero
    - Goal -> Object

Property	Value
Changeable	true
Container	false
Containment	true
Default Value Literal	
Derived	false
EKeys	
EOpposite	
EType	Scene
Lower Bound	0
Name	scenes
Ordered	true
Resolve Proxies	true
Transient	false
Unique	true
Unsettable	false
Upper Bound	-1
Volatile	false

Figure 3: The Ecore file that represents our RPG metamodel.

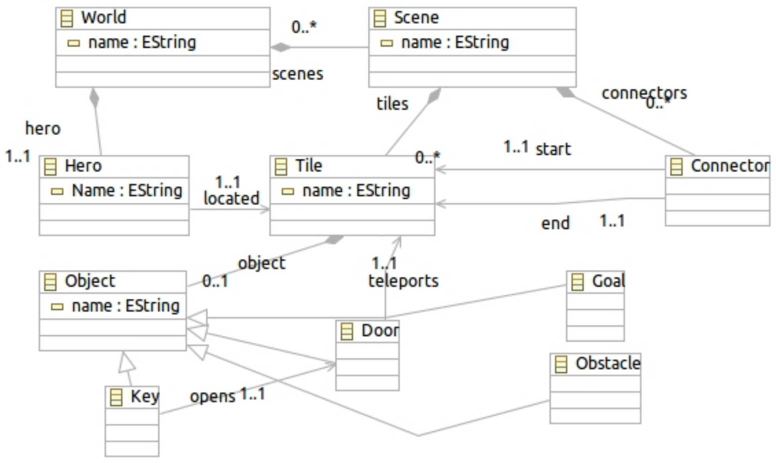


Figure 4: The RPG metamodel as diagram .

As seen from the diagram, a world can contain multiple scenes. Each Scene consists of multiple tiles and multiple connectors, where each connector connects 2 tiles together. A tile can have an object. This Object class is an abstract class, with the Key class, Door class, Goal class and Obstacle class as its subclasses. A world has exactly one Hero which is located at a specific Tile. Lastly, A key opens one specific door and a door teleports to one specific Tile.

It is important to notice that we did not focus on creating a complex full-blown RPG metamodel, we only focused on the part that is relevant to our experiment. However you might notice that a hero, doors and keys can have a name, which is unrelevant for a petrinet analysis. Also the obstacle objects are of no use for this analysis. This is the main reason for a initial transformation to a Reduced RPG model.

6.3. RRPG Metamodel

The Reduced RPG metamodel is comparable to the RPG metamodel, but leaves out a few details that are not relevant for analysis purposes. We present the diagram in Figure 5.

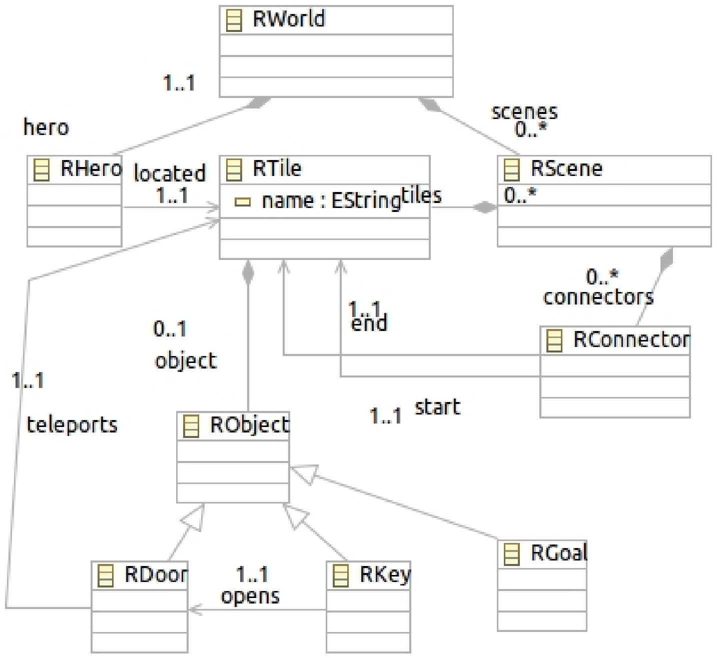


Figure 5: The RRPG metamodel as diagram .

One can notice that the name attributes from the Hero, World, Scene and Object class are removed. Also the Obstacle class has been taken out.

6.4. Petrinet Metamodel

Our petrinet metamodel was based on the metamodel presented by Barbero et al. (2007). It's design is very straight forward and is shown in Figure 6.

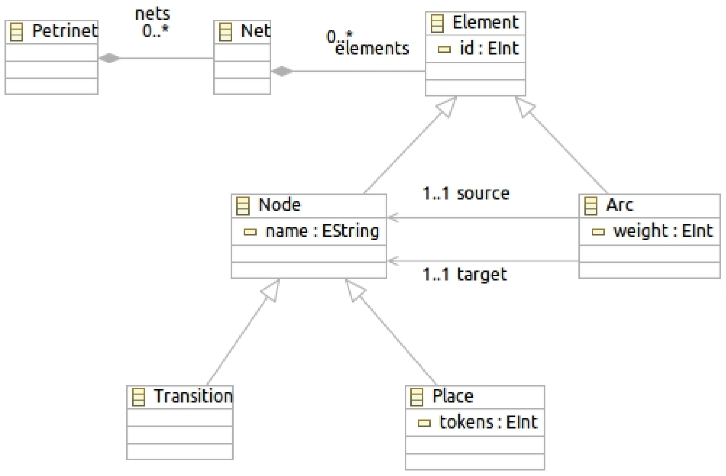


Figure 6: The Petrinet metamodel as diagram.

6.5. Pipe Metamodel

Since the tool Pipe only allows petrinet files of a specific format, we had to transform our petrinet models into a Pipe model. The Pipe metamodel contains a lot more extra information compared to the standard Petrinet metamodel (e.g. The Pipe metamodel also keeps graphics information, like coordinates and orientation.). Figure 7 represent our Pipe metamodel.

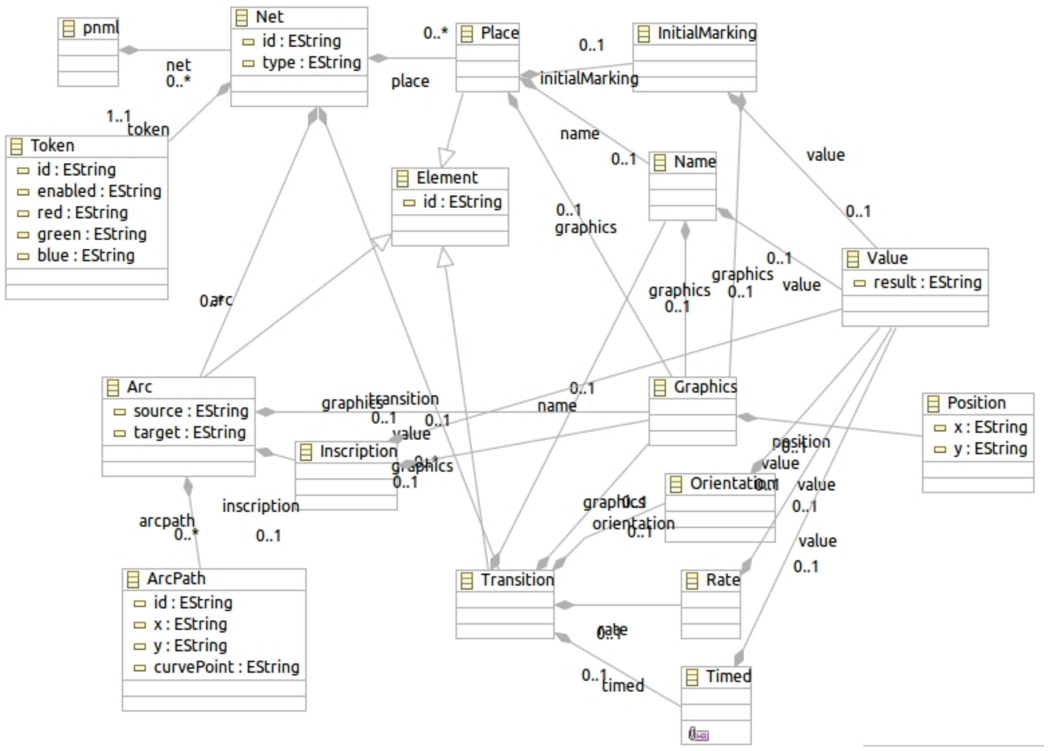
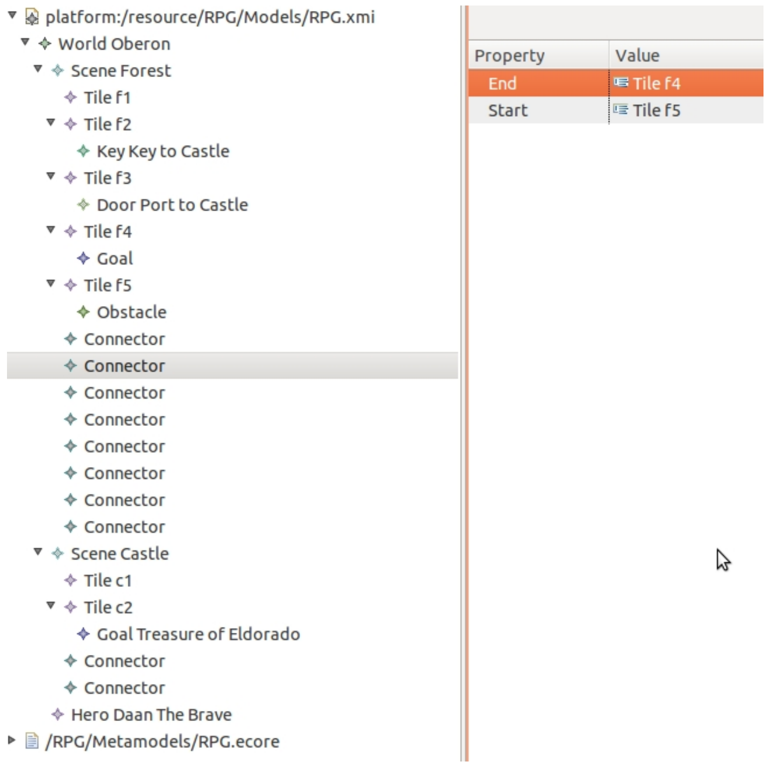


Figure 7: The Pipe metamodel as diagram.

7. Implementation

7.1. The inital RPG model

Eclipse offers a way to create a dynamic instance of a metamodel. By using this functionality we made our initial RPG model. By adding child/Sibling nodes and filling in their properties one can setup the model as they like. Figure 8 represents the model that we used for this experiment. A graphical representation is shown in Figure 9. Notice that the model is conform to the RPG metamodel.



```
▼ platform:/resource/RPG/Models/RPG.xmi
  ▼ World Oberon
    ▼ Scene Forest
        Tile f1
      ▼ Tile f2
          Key Key to Castle
      ▼ Tile f3
          Door Port to Castle
      ▼ Tile f4
          Goal
      ▼ Tile f5
          Obstacle
        Connector
        Connector
        Connector
        Connector
        Connector
        Connector
        Connector
        Connector
    ▼ Scene Castle
        Tile c1
      ▼ Tile c2
          Goal Treasure of Eldorado
        Connector
        Connector
      Hero Daan The Brave
▶ /RPG/Metamodels/RPG.ecore
```

Property	Value
End	Tile f4
Start	Tile f5

Figure 8: The initial RPG model used throughout our experiment.

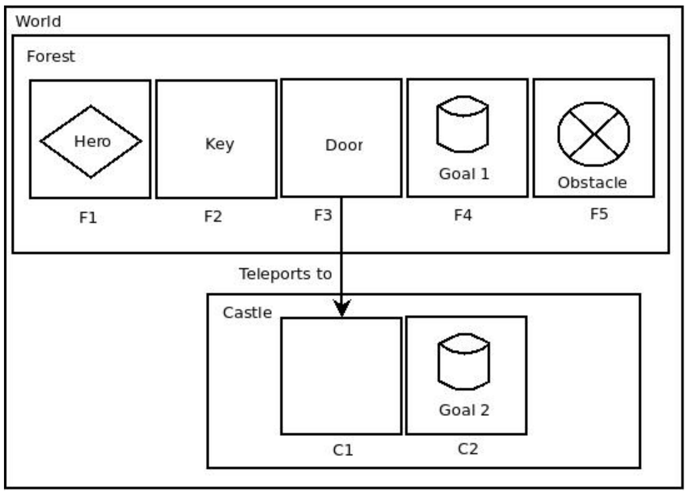


Figure 9: The initial RPG model visual representation.

7.2. Transforming the RPG model into a RRPG model

The transformation from normal RPG model into a Reduced RPG model is performed by our *RPG2RRPG* transformation definition. We will now briefly present parts of our implementation in order to give an idea of the internal working of ATL. We refer the interested reader to our full implementation code.

```
module RPG2RRPG;
create OUT : RRPG from IN : RPG;
```

The code above defines the metamodels for the input model and output model.

```
helper context RPG!Scene def: getValidTiles : Set(RPG!Tile) =
              self.tiles->select(c | not c.object.oclIsKindOf(RPG!Obstacle));
```

Helper functions allow us to navigate over the source model. In the above example, the function getValidTiles, which is a method on the RPG!Scene Object, returns the subset of Tiles of the Scene that do not have an obstacle on it.

Since the RRPG model is very much alike the RPG model, most rules will be a transformation from the RPG model into the comparable variant of the RRPG model. It's important to notice that in this transformation definition each rule only creates one target object. For example:

```
rule Scene2RScene {
       from
              s : RPG!Scene
       to
              rs : RRPG!RScene (
                     tiles <- (s.getValidTiles),
                     connectors <- (s.getValidConnectors)
              )
}
```

Here all Scene objects from the RPG metamodel are transformed into new RScene objects from the RRPG metamodel. Their contained tiles property however, will now be the result of the transformed valid tiles (the tiles that have no obstacle on it.). Since there is no mapping for the name attribute of the Scene, it will be removed in the Reduced RPG model.

```
rule Tile2RTile {
	from
		t : RPG!Tile (not t.object.oclIsKindOf(RPG!Obstacle))
	to
		rt : RRPG!RTile (
			object <- t.object,
			name <- t.name
		)
}
```

The code above represents a rule with a guard expression. It will transform all Tiles in the soure model, except those where the contained object is of the type Obstacle. By removing those from the transformation and by not adding them to the tiles property of RScene objects (see the previous rule), we can easily get rid of the obstacle tiles. The same approach is used to get rid of Connectors that link to and from tiles with obstacles on it.

7.3. Transforming the RRPG model into a Petrinet model

This transformation definition contains rules that create multiple target objects in the same rule. This greatly increases complexity. We again present some important aspects of this transformation definition:

```
helper def : id: Integer = 1;
```

The above helper, represent an integer variable, which is used to give unique id values to each petrinet element. We made a distinction between RTiles with and RTiles without a hero by using guard expressions. Depending on the presence of a hero, a RTile would be transformed into a Place with or without a token. For this we made 2 rules, namely: *Tile2Place* and *TileWithHero2Place*. This all is possible, because the guard expression make sure that a RTile can only be target of only one of these two rules (remember that each source element can only be once the target of a transformation). We present the *TileWithHero2Place* rule below:

```
rule TileWithHero2Place {
	from
		r : MM!RTile (r = thisModule.getHero.located)
	to
		p1 : MM1!Place
		(
			id <- thisModule.id,
			tokens <- 1,
			name <- r.name
		)
	do {
```

```
                thisModule.id <- thisModule.id+1;
                thisModule.elements <- thisModule.elements->including(p1);
        }
}
```

The above example shows how each element receives an unique id by increasing the id variable after each rule.

We present some more complex structures of our transformation in section 7.3.1 and 7.3.2.

7.3.1. Doors and keys

Modeling teleportation and the relationship of doors and keys required some more advanced ATL knowledge. A RDoor object will be transformed into an arc from the RTile that contains the door to a new transition and an arc from that transition to the RTile to where the door should teleport the hero to. This is shown in the following rule:

```
rule Door2DoorPlace {
        from
                r : MM!RDoor
        using{
                Tile : MM!RTile = thisModule.getTileFromObject(r);
        }
        to
                a1 : MM1!Arc(
                        source <- (Tile) ,
                        target <- (t1),
                        id <- thisModule.id
                ),

                t1 : MM1!Transition(
                        id <- thisModule.id+1
                ),

                a2 : MM1!Arc(
                        source <- (t1),
                        target <- (r.teleports),
                        id<- thisModule.id+2
                )
        do {
                thisModule.id <- thisModule.id+3;
                thisModule.elements <- thisModule.elements->including(a1);
                thisModule.elements <- thisModule.elements->including(t1);
                thisModule.elements <- thisModule.elements->including(a2);
        }
}
```

A RKey object on its turn is transformed into a *KeyNotYetTaken* and *KeyTaken* place. A newly created arc connects the *KeyTaken* place with the transition which was created by the RDoor object transformation. Here one of the more complex features of ATL comes in the picture, namely: the resolveTemp operation. This specific operation makes it possible to point, from an ATL rule, to any of the target model elements (including non-default ones) that will be generated from a given source model element by an ATL matched rule. We present the Key2KeyPlace rule below:

```
rule Key2KeyPlace {
       from
              r : MM!RKey
       to
           p1 : MM1!Place
              (
                     id <- thisModule.id,
                     tokens <- 0,
                     name <- 'KeyTaken'
              ),
              a1 : MM1!Arc(
                     source <- p1,
                     target <- thisModule.resolveTemp(r.opens,'t1'),
                     id <- thisModule.id+1
              ),
              p2 : MM1!Place
              (
                     id <- thisModule.id+2,
                     tokens <- 1,
                     name <- 'KeyNotYetTaken'
              )
       do {
              thisModule.id <- thisModule.id+3;
              thisModule.elements <- thisModule.elements->including(p1);
              thisModule.elements <- thisModule.elements->including(a1);
              thisModule.elements <- thisModule.elements->including(p2);
       }
}
```

The last part that is needed to make doors and keys work as petrinet model is done by the rules: *Connection2Move* and *Connection2MoveToKey*. These transform the RRPG RConnectors into a petrinet connection between it's connecting places. A *Connection2Move* will simply create a transition and 2 arcs that connect it's places with the transition. It is used for RConnectors that go towards RTiles that have no object or a RDoor object on it. A *Connection2MoveToKey* on the other hand is used for RConnectors that have as target a RTile that contains a RKey. It will create arcs and transitions and connects them with the *KeyTaken* and *KeyNotYetTaken* place accordingly.

The idea is to end up with something like Figure 10. In case the key is taken it will allow the hero to teleport from place f3 (which represents the tile with a door) towards place c1.

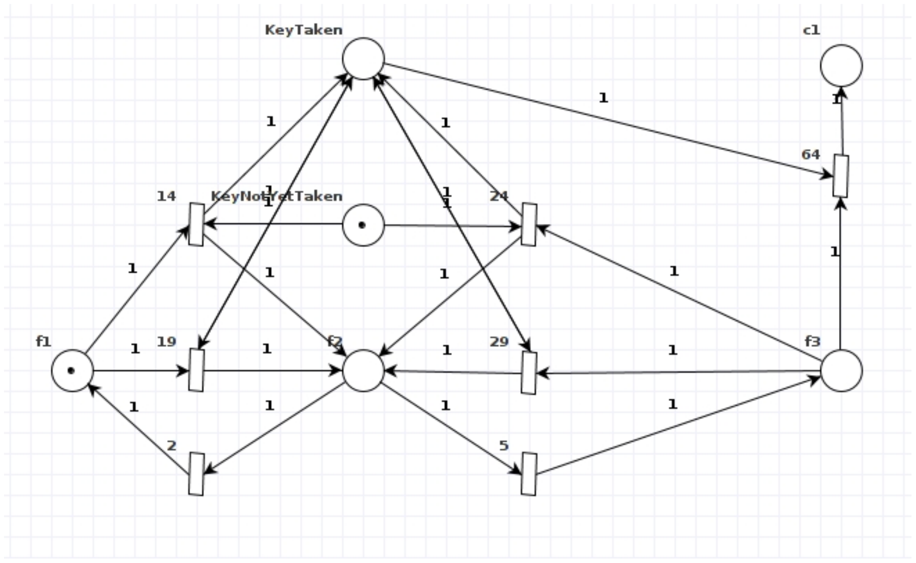


Figure 10: A petrinet concept of the use of keys and doors.

7.3.2. Goals

Since we can only win when all goals are taken, we should also model this into our petrinet. A single place should indicate if the hero has won or not. We used the RHero object transformation for this, since there can be only one RHero object in the RWorld. We show the rule below:

```
rule Hero2Setup{
       from
              h : MM!RHero
       to
              w : MM1!Place(
                     id <- thisModule.id,
                     tokens <- 0,
                     name <- 'Won'
              ),
              gf : MM1!Place(
                     id <- thisModule.id+1,
                     tokens <- 0,
                     name <- 'GoalsFound'
              ),
              gl : MM1!Place(
                     id <- thisModule.id+2,
                tokens <- thisModule.amountOfGoals,
                name <- 'GoalsLeft'
        ),
        a1 : MM1!Arc(
                source <- (gf) ,
                target <- (t1),
                weight <- thisModule.amountOfGoals,
                id <- thisModule.id+3
        ),
        t1 : MM1!Transition(
                id <- thisModule.id+4
        ),
        a2 : MM1!Arc(
                source <- (t1),
                target <- (w),
                id<- thisModule.id+5
        )
    do {
        thisModule.id <- thisModule.id+6;
        thisModule.elements <- thisModule.elements->including(w);
        thisModule.elements <- thisModule.elements->including(gf);
        thisModule.elements <- thisModule.elements->including(gl);
        thisModule.elements <- thisModule.elements->including(a1);
        thisModule.elements <- thisModule.elements->including(t1);
        thisModule.elements <- thisModule.elements->including(a2);
    }
}
```

A *Won*, *GoalsFound* and *GoalsLeft* place will be created by the above rule. Once all goals are found a transition will be enabled that goes from *GoalsFound* towards *Won*.

Each RGoal will be transformed in a *GoalTaken* place and *GoalNotYetTaken* place, which is comparable to the transformation of RKeys. They however also generate a *CheckMoreGoals* place.

Each time a goal is picked up, a token of *GoalsFound* will be consumed, the token that represents the hero's location will be moved to the *CheckMoreGoals* place, where the petrinet will check if there are still goals left. A transition will go from the *CheckMoreGoals* place to the place that resembles the tile where the goal is located if there are still goals left. If there are none left, only the transition that goes to Won will be enabled. We try to visualize this complex process in Figure 11

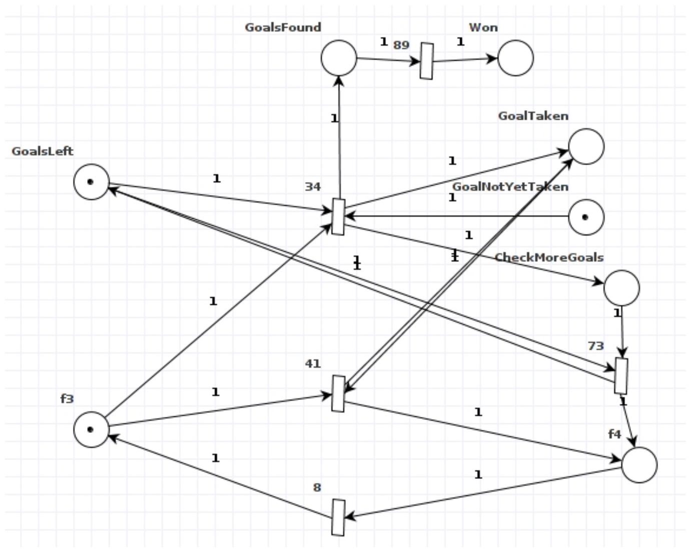


Figure 11: A visual representation of how the goal system is modeled.

7.4. Transforming the Petrinet model into a Pipe model

The process of transforming a Petrinet into a Pipe model is very straight forward since it is a 1-to-1 mapping of the Arcs, Transitions and Places. Each rule creates a few extra elements that are required by the Pipe framework. For example:

```
rule Place2Place{
      from
             p : MM!Place
      to
             pn : MM1!Place(
                    id <- p.id.toString(),
                    graphics <- g1,
                    name<-n,
                    initialMarking<-im
             ),
             g1 : MM1!Graphics(
                    position <- pos
             ),
             pos : MM1!Position(
                    x <- '10',
                    y <- '10'
             ),
             v : MM1!Value(
                    result<-p.name
             ),
             n : MM1!Name(
                    value <- v,
                    graphics <- g2
```

```
                ),
                g2 : MM1!Graphics(
                ),
                im : MM1!InitialMarking(
                        value<-v2,
                        graphics<-g3
                ),
                v2 : MM1!Value(
                        result<-p.tokens.toString()
                ),
                g3 : MM1!Graphics(
                )
}
```

7.5. From XMI to XML by using a Python Script

Since Pipe only allows specific XML files, we had to change a few lines in the XMI file to make it compatable. By using the *PipeXSI2PipeXML.py* script we end up with an analysable Petrinet. We included the code of our python script as appendix to this paper.

8. Resuls

By performing the 3 ATL transformations and the python script on the initial RPG model, which we introduced at the start of section 7, we end up with the petrinet depicted in Figure 12

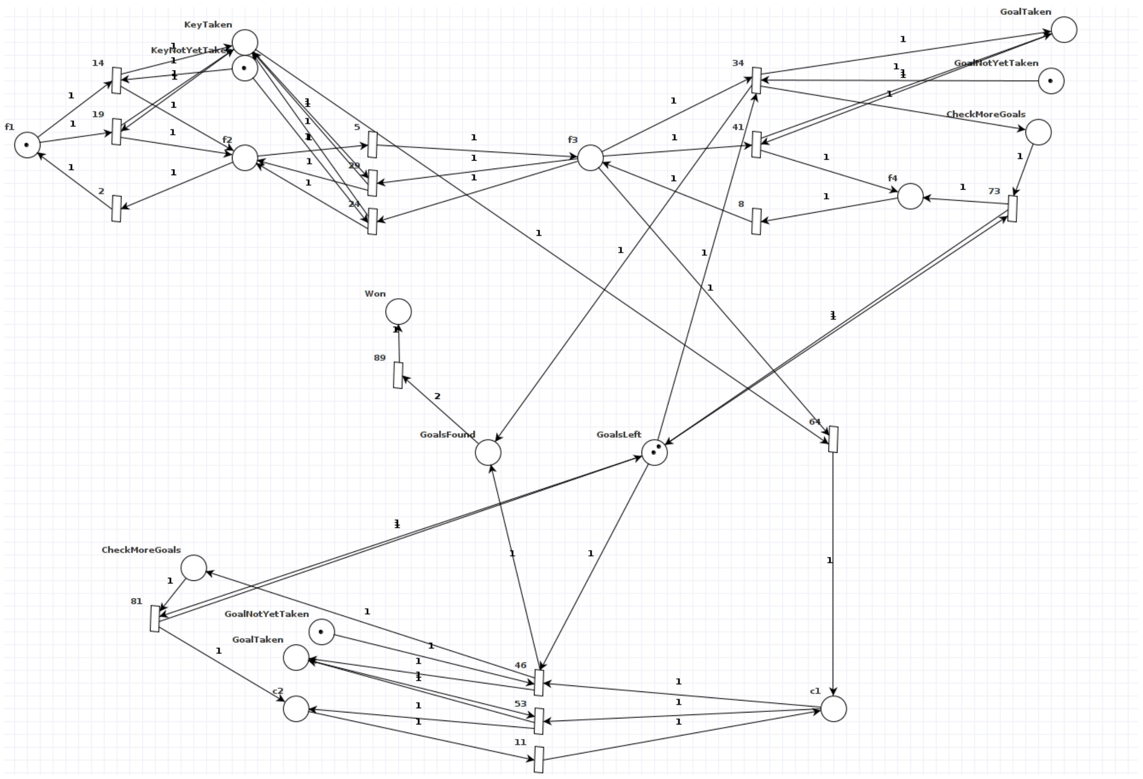
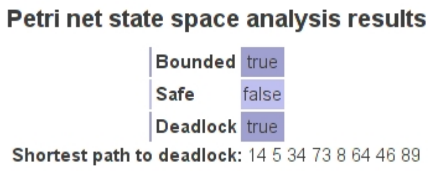


Figure 12: The resulting petrinet.

By performing a state space analysis in Pipe we discovered the shortest path, which is shown in Figure 13. One can also perform an invariant analysis and more.



Petri net state space analysis results

Bounded	true
Safe	false
Deadlock	true

Shortest path to deadlock: 14 5 34 73 8 64 46 89

Figure 13: The state space analysis of the petrinet.

9. Conclusion and future work

9.1. Conclusion and comparison

In this paper we presented an ATL experiment where we transformed a RPG model into a working and analyzable petrinet. We were sceptic at the start, but the success of our experiment made us believe in the power of ATL. Moreover, we believe that ATL is the perfect tool for performing small/medium sized 1-to-1 mapped transformations. Our RPG to Petrinet and more specifically the *RRPG2Petrinet* transformation definition is in the gray area, it starts to show why ATL is not the perfect language for this kind of transformations, namely: the amount of rules and their size starts to become large, the creation of multiple target elements tends to make it complex and the lack of visual nature. We also tried to find a way to add constraints to our metamodel, however there does not seem to be a proper way to do this.
If we compare ATL to other model transformation languages and tools, like for example AtomPM (Syriani (2009)), we can cleary state that a visual representation of a rule can make it a lot more comprehendable and readable. The creation of the metamodel (abstract syntax) in ATL can be however simular to how it is done in AtomPM, by using the EMF diagram editor. Another major difference between AtomPM and ATL is the fact that AtomPM allows to execute rules separately by write an execution scheme of the rules. While ATL simply executes all rules on their matched elements at the same time. If we compare their environments we could conclude that both are not easy to setup. ATL has a lot of outdated information, while AtomPM is a rather new tool that misses a large active community behind it. We think however that a tool like AtomPM could clear the job in a significant less amount of time and still end up being more readable, reuseable and understandable, not only towards the creators/developers but also non-technical people.

9.2. Future work

Our RPG to petrinet experiment can be extended in many ways. First of all the RPG DSL could be enlarged, this allows the creation of complexer models. The game can be turned into a turn based version, where villains try to kill the hero. Traps can be included and much more. These new features require more complex structures in the *RRPG2Petrinet* transformation. E.g. Health and damage can be represented as tokens in the petrinet. Another extension could be code generation from our RPG DSL.

Appendix A. PipeXSI2PipeXML.py

```
 #!/usr/bin/python

import sys

#USAGE! :
# arg1 is XMI file that has to be adapted in order to be able to open in Pipe
# arg2 is to-be-made XML file that can be opened in pipe

arg1 = str(sys.argv[1])
arg2 = str(sys.argv[2])

f2 = open(arg2,'w')

#open eclipsecommand file
with open(arg1) as f:


       #go through each line
       for line in f:

              #change line in case it's needed
              if (line.find("<Pipe:pnml") != -1):
                     line = "<pnml>"
              elif (line.find("</Pipe:pnml") != -1):
                     line = "</pnml>"
              elif (line.find("<value result=") != -1):
                     splitted = line.split('"')
                     value = splitted[1]
                     line = "<value>"+value+"</value>"
              elif (line.find("</value>") != -1):
                     line = "<value>0</value>"

              #write line to output
              f2.write(line)

f2.close()
```

References

abidredlove, 2009. abidredlove youtube channel @ONLINE. URL: `http://www.youtube.com/user/abidredlove?feature=watch`.

Allilaire, F., Bzivin, J., Jouault, F., Kurtev, I., 2006. Atl – eclipse support for model transformation, in: IN: PROC. OF THE ECLIPSE TECHNOLOGY EXCHANGE WORKSHOP (ETX) AT ECOOP.

Araújo, M., Roque, L., 2009. Modeling games with petri nets. Breaking New Ground: Innovation in Games, Play, Practice and Theory. DIGRA2009. Londres, Royaume Uni .

Barbero, M., Jouault, F., Gray, J., Bézivin, J., 2007. A practical approach to model extension, in: Model Driven Architecture-Foundations and Applications, Springer. pp. 32–42.

Brom, C., Abonyi, A., 2006. Petri-nets for game plot, in: Proceedings of AISB artificial intelligence and simulation behaviour convention, Bristol, pp. 6–13.

Bzivin, J., Dup, G., Jouault, F., Pitette, G., Rougui, J.E., 2003. First experiments with the atl model transformation language: Transforming xslt into xquery, in: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture.

Furtado, A.W.B., de Medeiros Santos, A., 2006. Sharpludus: improving game development experience through software factories and domain-specific languages. Universidade Federal de Pernambuco (UFPE) Mestrado em Ciência da Computação centro de Informática (CIN) .

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., 2008. Atl: A model transformation tool. Sci. Comput. Program. 72, 31–39. URL: `http://dx.doi.org/10.1016/j.scico.2007.08.002`, doi:10.1016/j.scico.2007.08.002.

Marques, E., Balegas, V., Barroca, B.F., Barisic, A., Amaral, V., 2012. The rpg dsl: A case study of language engineering using mdd for generating rpg games for mobile phones, in: Proceedings of the 2012 Workshop on Domain-specific Modeling, ACM, New York, NY, USA. pp. 13–18. URL: `http://doi.acm.org/10.1145/2420918.2420923`, doi:10.1145/2420918.2420923.

PIPE2, . Pipe2 website @ONLINE. URL: `http://pipe2.sourceforge.net/`.

Reyno, E.M., Carsí Cubel, J.A., 2009. Automatic prototyping in model-driven game development. Comput. Entertain. 7, 29:1–29:9. URL: `http://doi.acm.org/10.1145/1541895.1541909`, doi:10.1145/1541895.1541909.

Syriani, E., 2009. Atompm @ONLINE. URL: `http://syriani.cs.ua.edu/atompm/atompm.htm`.

Walter, R., Masuch, M., 2011. How to integrate domain-specific languages into the game development process, in: Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology, ACM. p. 42.

Wang, J., 2007. Petri nets for dynamic event-driven system modeling. Handbook of Dynamic System Modeling , 1–17.