Role-Playing Game Modeling in VMTS

Dylan Kiss

University of Antwerp, Department of Mathematics and Computer Science, Middelheimlaan 1, 2020 Antwerp, Belgium

Abstract

VMTS [1] is a graph-based, domain-specific (meta)modeling and model processing framework. The system provides a graphical interface for defining, customizing and utilizing languages. This report explores the capabilities of VMTS by means of modeling a role-playing game. This consists in creating an abstract syntax for the RPG formalism and defining operational semantics in order to simulate a model. This will be compared to the capabilities of AToMPM to state the advantages and disadvantages of VMTS.

Keywords: VMTS, metamodeling, metamodel-based transformations, role-playing game

1. Introduction

By means of a running example, a role-playing game, I will explore the modeling and transformation capabilities of VMTS and compare them to my experiences in AToMPM.

Section 2 shortly explains what VMTS is. Section 3 elaborates the RPG formalism used throughout this report. Section 4 explains how the abstract syntax (metamodel) and constraints for the RPG formalism are created. Section 5 explains how operational semantics can be defined for our RPG formalism. Section 6 compares the capabilities and user-friendliness of VMTS with AToMPM. At last, section 7 concludes my experiences with VMTS.

2. VMTS

The Visual Modeling and Transformation System (VMTS) is a graph-based, domain-specific (meta)modeling and model processing framework developed at the Budapest University of Technology and Economics. The system provides a graphical interface for defining, customizing, and utilizing

Email address: dylan.kiss@student.uantwerpen.be (Dylan Kiss)

languages [2]. Besides that, VMTS provides the capability of defining and using transformations for your metamodel. This can serve for model optimizations, simulations and even translation to another formalism.

3. RPG formalism

3.1. Syntax and static semantics

1. An RPG consists of a number of scenes that have a name.
2. In each scene, there are a number of connected tiles. Tiles can be connected to each other from the left, right, top or bottom. This way, a map is created for the scene.
3. If a tile has a left neighbor, that neighbor should have the tile as its right neighbor. If a tile has a top neighbor, that neighbor should have the tile as its bottom neighbor.
4. In the game there are two types of characters: a hero and a villain. There can only be one hero, but there can be zero or more villains. A character is always exactly on one tile.
5. A tile can be an empty tile, an obstacle on which no character can stand, a trap or a door. A door is a portal to a door on another scene.
6. On an empty tile there can be an item.
7. There are three types of items: a goal, a key and a weapon. A weapon has a stricly positive damage value.
8. The hero and a villain have a health value that depicts how much damage they can take. The health always has a strictly positive value.
9. The hero, a villain, and a trap have a damage value that depicts how much damage they inflict. Damage is always strictly positive.

3.2. Dynamic semantics

1. A character can move from one adjacent tile to another (provided it is not an obstacle or it is not occupied).
2. An item can be picked up by the hero by walking on its tile. Every item can only be picked up once.
3. A hero can pick up a goal. The hero wins if he can pick up all goals. There must be at least one goal in the game.
4. The hero can pass through a door to enter another scene. If the hero goes back through the door, he goes back to the original door at the original scene.
5. The hero can attack villains and vice versa, if they stand on adjacent tiles.
6. Villains do not attack each other.

7. A trap hurts the hero, but not a villain.
8. A door can be locked, and the hero must pick up a particular key (for that door) to be able to enter that door.
9. A hero can pick up a weapon that can give the hero additional damage, according to the damage of the weapon.
10. The hero and a villain inflict damage (according to their damage value) if they choose to attack (when on a tile adjacent to their adversary's tile). A trap inflicts damage (according to their damage value) on the hero if the hero steps onto it.
11. The game is simulated in time slices: first, the hero gets one chance to move or attack. Then, all the villains in the same scene, each get their chance to move or attack. The order in which the villains get their chance is not determined.
12. Villains in a different scene from the "active" scene (i.e., the scene in which the hero currently resides) do not do anything.
13. When the hero achieves the goals, or dies, the game is over.

4. Abstract syntax

4.1. Metamodeling formalism

The first step is creating the metamodel, or abstract syntax, of the RPG language. VMTS provides a simplified class diagram formalism for this (Figure 1). A class is here called an *atom*. If you create an atom, you have to specify its name in the metamodel as well as its *instance name*, the type name of the atom when it is instantiated in a model. Additionally you can create attributes for an atom, which can be of the simple types *bool*, *int*, *float*, *string* or *Reference*, or of a user defined complex type. VMTS provides three types of connections between atoms: a *relationship*, *containment* or *inheritance*. For a connection you have to assign multiplicities and role names to both its ends. These role names are important when you want to write constraints for your metamodel.

4.2. The RPG metamodel

I will explain the metamodel for the RPG formalism (Figure 2). The first atom is *SceneMeta*. This atom represents a scene in the game and has an attribute *name*. It has a containment relationship (*containsTile*) with the atom *TileMeta*, that represents a tile in a scene. TileMeta has four relationships with itself: *leftNeighbor*, *rightNeighbor*, *topNeighbor* and *bottomNeighbor*. It also has a containment relationship (*containsCharacter*) with the atom *CharacterMeta*, which represents a character on a tile. CharacterMeta has three

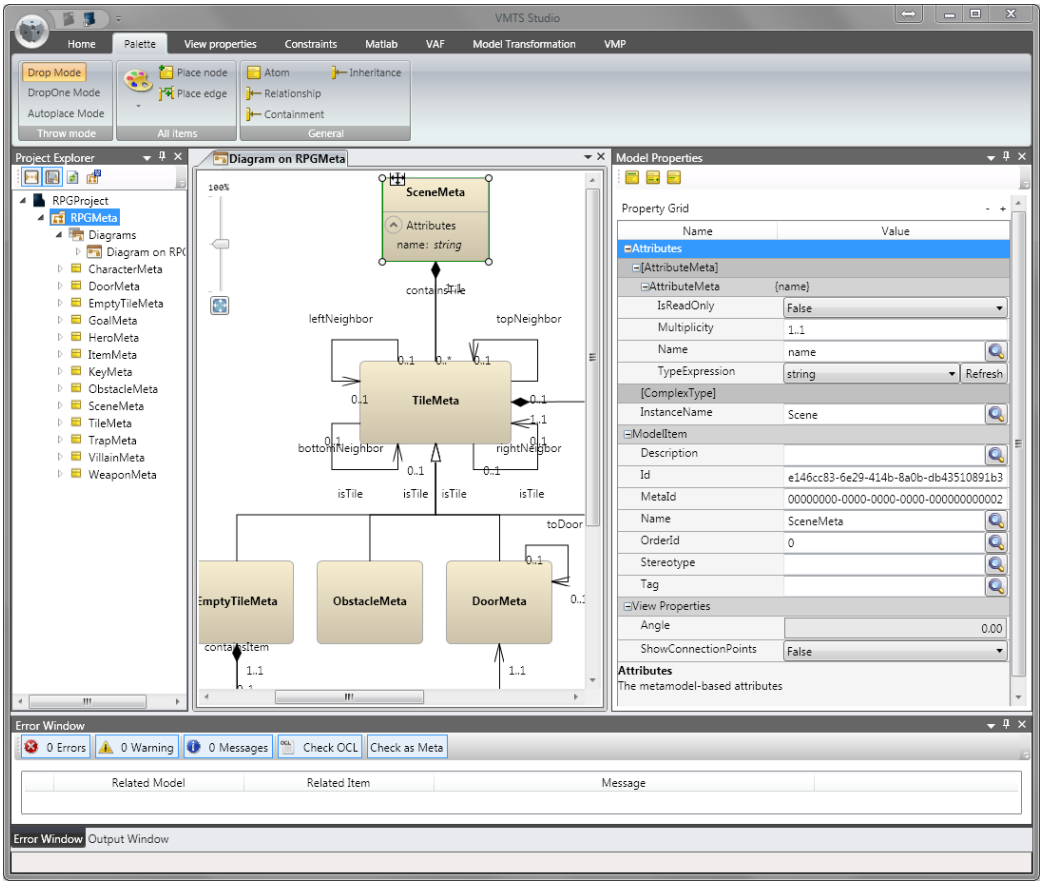

Figure 1: Creating the RPG metamodel in VMTS

attributes: *name*, *health* and *damage*, which represent these properties of a character. CharacterMeta also has two subtypes: *HeroMeta* and *VillainMeta*, which represent a hero and a villain, respectively. TileMeta has four subtypes: *EmptyTileMeta*, *ObstacleMeta*, *DoorMeta* and *TrapMeta*. The latter has an attribute *damage*, which represents the damage inflicted when a character steps on a trap. DoorMeta has a relationship *toDoor* with itself. This represents where a door leads to on another scene. EmptyTileMeta has a containment relationship (*containsItem*) with *ItemMeta*, which represents an item on a regular tile. ItemMeta has three subtypes: *GoalMeta*, *KeyMeta* and *WeaponMeta*. The latter has an attribute *damage*, which represents the increase in damage a hero gets when he picks up the weapon. KeyMeta has a relationship (*unlocksDoor*) with DoorMeta, which represents that a key can unlock a specific door.

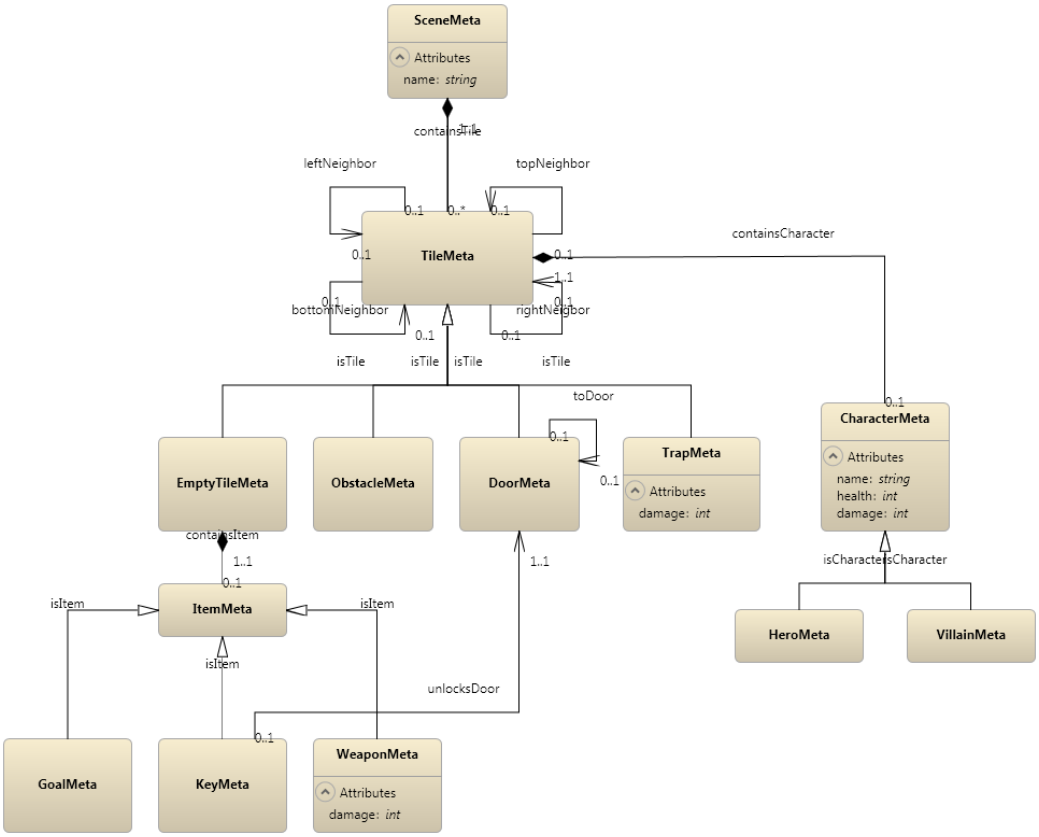

Figure 2: RPG metamodel

4.3. Constraints

In VMTS it is possible to define constraints either in OCL (Object Constraint Language) or in C#. When defining a constraint, you first have to choose a *context* for your constraint. This can be any atom or relationship in your metamodel or even the metamodel itself. The context you choose will be the *self* object in your constraint.

I added seven constraints to my metamodel (Figure 3). The first four constraints, *(right/left/top/bottom)NBConsistent*, check that, when a tile has a connection to another tile, i.o.w. is a neighbor of that tile, that other tile has the right connection back to the original tile. The *rightNBConsistent* constraint (Listing 1) e.g. checks whether, if a tile has a right neighbor, that neighbor has the original tile as its left neighbor.

The fifth constraint, *nothingOnObstacle* (Listing 2), makes sure that no character can stand on an obstacle. It uses the role name *Character* of the relationship *containsCharacter* and the function *oclIsUndefined* to check whether there is no character on an obstacle.

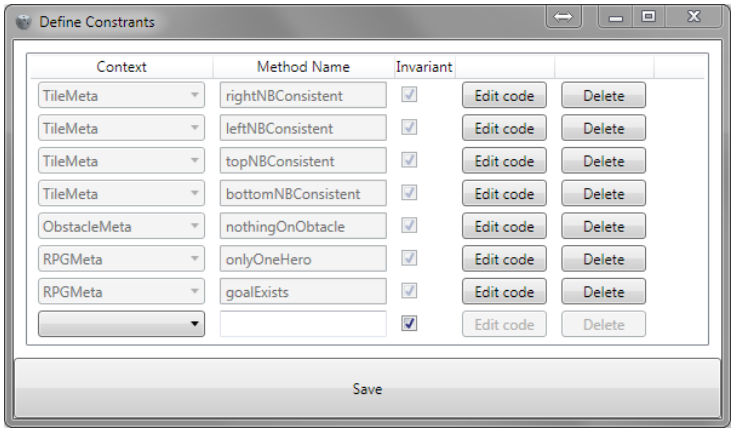

Figure 3: RPG metamodel constraints

```
package Constraints
  context TileMeta
    inv rightNBConsistent:
      (not self.RightTileEnd.oclIsUndefined())
      implies (self.RightTileEnd.LeftTileEnd = self)
endpackage
```

Listing 1: Constraint *rightNBConsistent*

```
package Constraints
  context ObstacleMeta
    inv nothingOnObstacle:
      self.Character.oclIsUndefined()
endpackage
```

Listing 2: Constraint *nothingOnObstacle*

The sixth constraint, *onlyOneHero* (Listing 3), specifies that there should be exactly one hero in the game. The context used here is *RPGMeta*,

```
package Constraints
  context RPGMeta
    inv onlyOneHero:
      self.HeroMeta->size() = 1
endpackage
```

Listing 3: Constraint *onlyOneHero*

the metamodel itself. Since VMTS does not support the OCL function *allInstances()*, the workaround is to use the atom name as role name in

the metamodel context.

The seventh constraint, *goalExists* (Listing 4), checks whether there is at least one goal in the game. It is specified in the same way as the previous

```
package Constraints
  context RPGMeta
    inv goalExists:
      self.GoalMeta->size() > 0
endpackage
```

Listing 4: Constraint *goalExists*

constraint.

With the constraints added, the abstract syntax of the RPG formalism is now complete. Figure 4 shows an example model that conforms the RPG metamodel created. It contains one scene with nine tiles (seven empty tiles, one obstacle and one trap), a hero, a villain and a goal. As such it also meets our constraints.

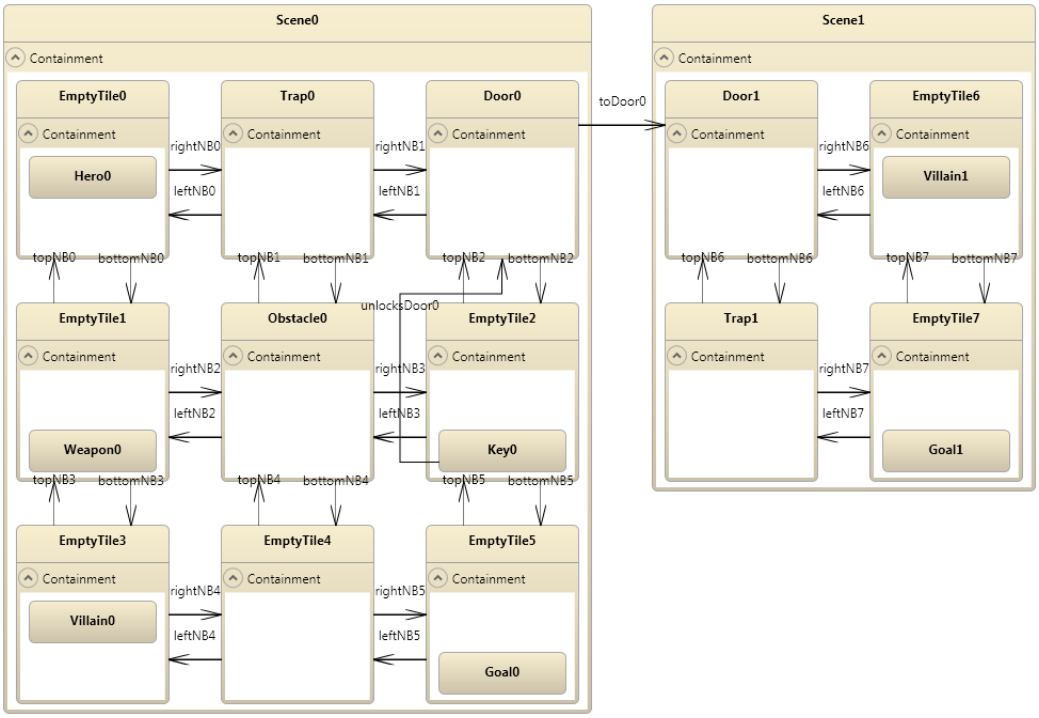

Figure 4: RPG model example

5. Operational semantics

VMTS makes use of graph rewriting to implement transformations. The first step is creating the rewriting rules for the RPG formalism. For each rule, we create a new model in VMTS of the type MTRMETA (Model Transformation Rule). The first thing to do in a new rule is specify the metamodel(s) that you want to use (Figure 5). You can specify multiple metamodels, which

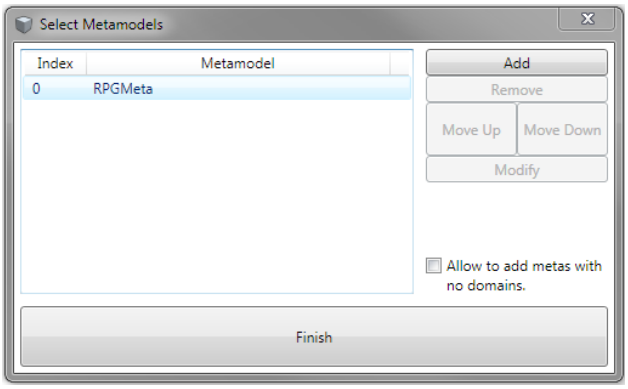

Figure 5: Specifying the metamodel(s) to use in a rewriting rule in VMTS

is useful if you want to implement denotational semantics and translate one formalism into another. The next step in constructing a rewriting rule is specifying the nodes and relations to be matched. For each node or relation you place on the diagram, you can choose which element(s) in your metamodel it should conform to, and whether it should be created, modified or deleted if necessary (Figure 6). All these options are visually represented:

Figure 6: Node settings in a rewriting rule in VMTS

deleted elements are red, modified elements are grey, created elements are blue ... Figure 7 shows an example rewriting rule that moves a hero to a

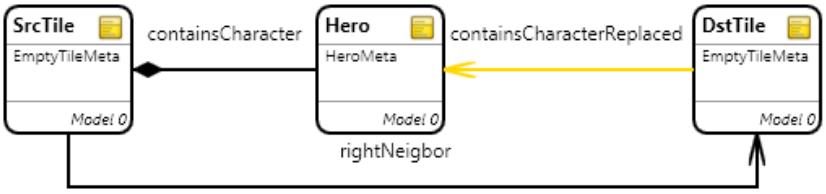

Figure 7: Example rewriting rule in VMTS

neighbor tile on the right. The yellow arrow represents replacing a relation, in this case *containsCharacter*.

It is also possible to specify constraints and/or actions for every element in a transformation rule. Constraints and actions are written in C#. When creating rules for the RPG formalism, I was unable to get rules with constraints or actions compiled. As such, I have limited the rules with the following assumptions:

1. There are no villains in the game
2. There are only two types of tiles in the game: EmptyTile and Obstacle
3. There is only one type of item in the game: Goal

With these assumptions on an RPG model, I created six rewriting rules for the operational semantics (Figure 8):

1. Rule_MoveHeroLeft
   Move the hero from an empty tile to its left neighbor, if that is an empty tile too.
2. Rule_MoveHeroRight
   Move the hero from an empty tile to its right neighbor, if that is an empty tile too.
3. Rule_MoveHeroTop
   Move the hero from an empty tile to its top neighbor, if that is an empty tile too.
4. Rule_MoveHeroBottom
   Move the hero from an empty tile to its bottom neighbor, if that is an empty tile too.
5. Rule_PickUpGoal
   If there is a goal on the empty tile the hero stands on, remove that goal.
6. Rule_GoalsNotCollected
   Check whether there are still one or more goals in the game.

The last step is to create a *Transformation Control Flow* that specifies in which order the rules are executed. We again create a new model, but this

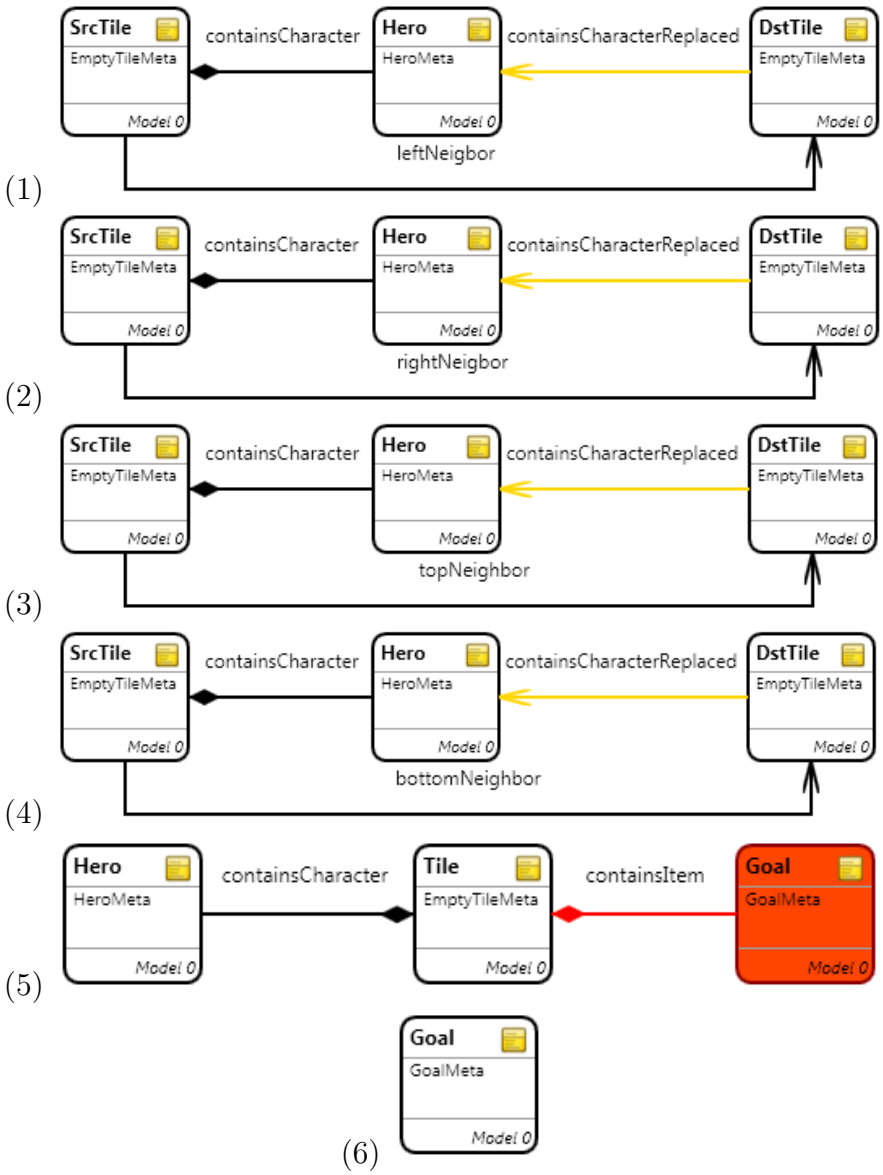

Figure 8: Rewriting rules for the RPG formalism in VMTS

time of the type TCFMETA (Transformation Control Flow). Like with the rewriting rules, we also have to specify the metamodel(s) we are using first. After that, we can add a start node, an end node, rule containers and flow edges. These four elements will determine the order in which the rules are executed.

Since VMTS does not support elements that make a random choice between rules, we have to specify a deterministic order in which the rules will

be executed. Figure 9 shows the TCF diagram for the RPG formalism. The

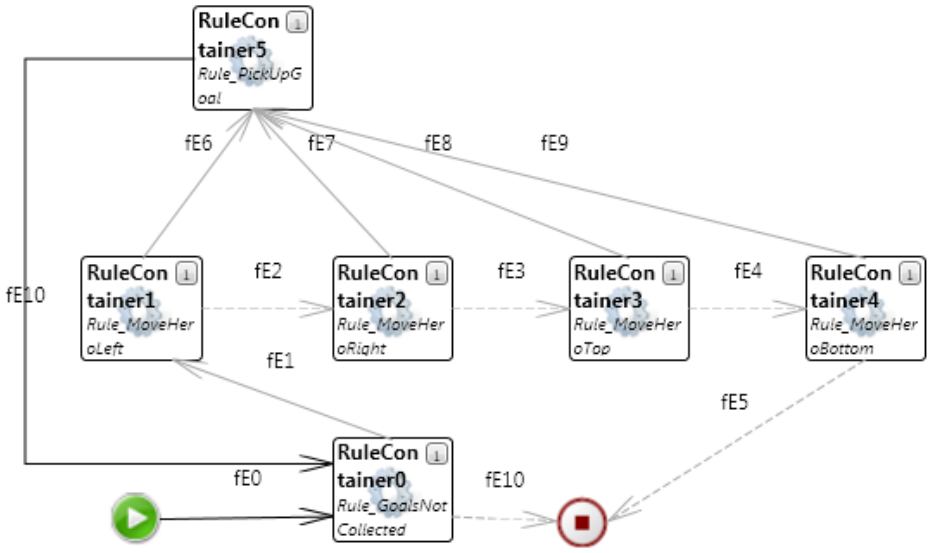

Figure 9: Transformation Control Flow diagram for the RPG formalism

first rule that gets executed is *GoalsNotCollected*, which checks whether there are still goals. If that rule fails (dashed gray line), we stop the transformation. If that rule succeeds (full gray line), we go to the "move" rules. These rules are tried in the order: left, right, top, bottom. If a rule fails, it tries the next rule. If even the last rule *MoveHeroBottom* fails, the transformation stops, because the hero is stuck. If one of the "move" rules succeeds, the *PickUpGoal* rule gets executed, which removes a goal from its tile if the hero is on that tile. After that rule we go straight back to the first rule, *GoalsNotCollected*, and start all over again.

As you might notice, in most cases this transformation control flow will not be able to collect the goals. Very soon it will just move the hero back and forth, since it always tries the rules in the same order. This restriction makes that we can not model a random behavior of the RPG.

6. Comparison with AToMPM

To conclude this report, I will compare the capabilities and user-friendliness of VMTS and AToMPM in order to state the advantages and disadvantages of both tools.

6.1. Abstract and concrete syntax

Creating an abstract syntax or metamodel in VMTS is very similar to the way it is done in AToMPM. They both use a simplified class diagram. The

major difference and advantage that VMTS has, is that it supports n-layer metamodeling. When creating elements in VMTS, you have to define a name as wel as an instance name.

Creating a concrete visual syntax is different in VMTS than in AToMPM. AToMPM has a visual syntax plugin built in, which enables you to easily create a visual syntax for your formalism. VMTS does not have such a plugin built in. There is no information available about this, although they put it as one of their features. After mailing with a developer at the VMTS team, I found out that you should use WPF (Windows Presentation Foundation) to create a specific plugin that implements a visualization. AToMPM is clearly more user-friendly for this purpose.

6.2. Operational and denotational semantics

Both VMTS and AToMPM allow you to create operational and denotational semantics. They also both use graph rewriting for this, although their visual representation differs. In AToMPM you have a clearly divided left-hand side (LHS), right-hand side (RHS) and negative application condition (NAC) for every rewriting rule. VMTS puts everything in one graph, which makes it easier to see what gets created, modified or deleted, but it does not support negative application conditions.

6.3. Modeling environment

Although VMTS has a professional looking interface, it did not live up to my expectations. When writing constraints or actions in rules and executing them, the tool always crashed. Another disadvantage is that the tool is only available for the Windows platform, because it is written in C# using the .NET framework. AToMPM is a web-based tool and as such usable on all platforms.

7. Conclusion

For this project, I created a role-playing game formalism in VMTS. My intentional plan was to create an abstract and concrete visual syntax, as well as operational en denotational semantics for the RPG formalism. In the end I have only been able to create an abstract syntax and define restricted operational semantics. Creating a concrete visual syntax is not supported out-of-the box in VMTS and requires developing a plugin with WPF. When defining semantics, I was unable to get constraints and actions to work. As such, I could only create basic rewriting rules that didn't require constraints or actions. Also, because of this, there was no way to create denotational semantics.

In the end I have compared VMTS to AToMPM. From my experience, I find AToMPM more user-friendly than VMTS. Although VMTS has the advantage of being an n-layer metamodeling tool. VMTS has been completely rewritten last summer, according to the developers, but it still needs finishing up. Probably most bugs will be gone in the next version.

References

[1] T. Levendovszky, L. Lengyel, G. Mezei, H. Charaf, A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, Electronic Notes in Theoretical Computer Science 127 (1) (2005) 65–75.

[2] Budapest University of Technology and Economics, Visual Modeling and Transformation System.
URL `https://www.aut.bme.hu/Pages/Research/VMTS/Introduction`

[3] Budapest University of Technology and Economics, VMTS Modeling Tutorial.
URL `https://www.aut.bme.hu/Upload/Pages/Research/VMTS/VMTS_Modeling_Tutorial.pdf`

[4] Budapest University of Technology and Economics, VMTS Model Transformation Tutorial.
URL `https://www.aut.bme.hu/Upload/Pages/Research/VMTS/VMTS_ModelTransformation_Tutorial.pdf`