Using Groove for analysing RPGame models

Brent van Bladel

University of Antwerp, Belgium

Abstract

Domain-specific visual modelling is a relatively new area in the software engineering. The development of this methodology is accompanied by a need for good visual modelling tools. A case study shows the different features of one such tool named GROOVE.

Keywords:
visual modelling language, graph transformation, Groove

1. Introduction

Since object-oriented programming was introduced in the 1960s, there haven't been many changes to the way we program. Recently, however, the concept of using domain-specific (visual) modelling to generate code is becoming more popular. As this popularity grows, the need for good visual modelling tools grows as well. This paper describes the features offered by one such modelling tool: GROOVE.

GROOVE stands for GRaph-based Object-Oriented VErification. It is centered around the use of simple graphs for modelling object-oriented systems, and graph transformations as a basis for model transformation and operational semantics [1].

This paper will describe the implementation of the RPGame formalism in GROOVE. Section 2 explains the RPGame formalism and section 3 shows its implementation in GROOVE. We analyse different models of the RPGame formalism using GROOVE in section 4. We conclude in section 5.

2. RPGame formalism

2.1. Syntax and Static Semantics

The RPGame formalism describes a simple 'Role Playing Game'. An RPGame consists of a world that is divided into a number of scenes. Each scene has a name and contains a number of connected tiles. These tiles can be connected to each other from the left, right, top or bottom. This way, a map is created for the scene. A logical positioning of the tiles is expected. Therefore, if a tile has a left neighbor, that neighbor should have the tile as its right neighbor, and vice versa. If a tile has a top neighbor, that neighbor should have the tile as its bottom neighbor, and vice versa.

There are two types of characters: a hero and a villain. Each character is always located on exactly one tile. 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. They also have a damage value that depicts how much damage they inflict. Damage is always strictly positive.

There are three types of tiles: an obstacle, a trap and a door. If a tile is an obstacle, no character can stand on it. If a tile is empty, it can contain an item. There are three types of items: a goal, a key and a weapon. An item is located on exactly one tile that is not an obstacle. A trap and a weapon have a strictly positive damage value, depicting how much damage they inflict. A door is a portal to a door on another scene and it can be locked. A key is connected to a door that it unlocks. There must be at least one goal in the game.

2.2. Dynamic Semantics

A character can move from one adjacent tile to another, provided it is not an obstacle and it is not occupied. If the hero moves to a tile containing an item, the item is picked up. Every item can only be picked up once. The hero wins if he picks up all goals. When this happens, the game stops.

The hero can pass through a door to enter another scene. If he goes back through the door, he goes back to the original door at the original scene. A door can be locked, and the hero must pick up the key for that door to be able to enter that door.

The hero can attack villains and vice versa, if they stand on adjacent tiles. Villains do not attack each other. The hero and a villain inflict damage according to their damage value when they attack. A trap hurts the hero, inflicting damage according to its damage value when the hero steps onto it. A hero can pick up a weapon, that gives the hero additional damage, according to the damage of the weapon.

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. Villains in a different scene from the "active" scene (i.e., the scene in which the hero currently resides) do not do anything.

3. RPGame in GROOVE

3.1. Type Graph

GROOVEs implementation of metamodelling is a type graph. The type graph specifies the allowed structure of the graphs, as well as the node type hierarchy, similar to a class diagram. Figure 1 shows the type graph of the RPGame formalism [2].

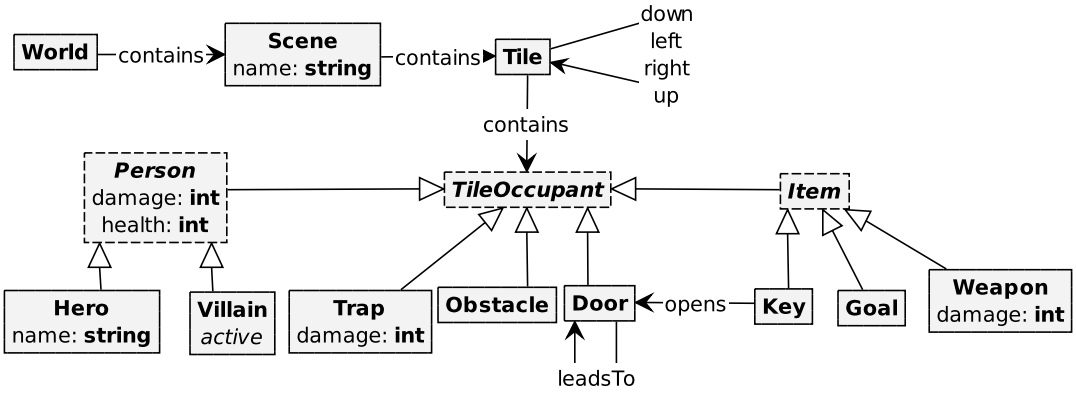

Figure 1: Type graph of RPGame in GROOVE

This implementation simplifies RPGame formalism to a straightforward basic idea: a world contains scenes which contain tiles which can contain something. Therefore we have a World, Scene and Tile type, as well as an abstract TileOccupant type. They have edges with a 'contains'-label between them, indicating the containment relations. The Scene type also has a name attribute and the Tile type has four self-edges to link tiles together: down, left, right and up.

The TileOccupant type is an abstract type, which is a special kind of type provided by GROOVE. Abstract types, together with subtyping, allows for advanced type graphs which are similar to inheritance hierarchies in class diagrams. An abstract type can't have object instantiations. However, when an object with an abstract type is expected, an object with a subtype of that abstract type should be given.

There are three kinds of TileOccupant types in our implementation of the RPGame formalism. A TileOccupant can be a Person, which is an abstract subtype of the TileOccupant type. A Person has a health value and a damage value as its attributes. A Person object can't be initiated because Person is also an abstract type. It has two subtypes: Hero and Villain. A hero has a name attribute and a villain has a boolean attribute which is needed in the operational semantics (see section 3.2). A TileOccupant can also be an Item, which is another abstract subtype. It has three subtypes: Goal, Weapon and Key. A weapon has a damage value attribute. A key has an edge labelled 'opens' connected to the door it can unlock. Finally, a TileOccupant can also define a special kind of tile: an Obstacle, a Trap which has a damage value attribute or a Door which has a self-edge labelled 'leadsTo' to link two doors together.

3.2. Transformation rules

With the type graph defined, we can already model instances of the RPGame formalism. In order to simulate these models, we need transformation rules to transform the models in specified patterns.

In order to simulate an RPGame, the characters must be able to move. Figure 2 shows how we implement this in GROOVE. The tile containing the hero, the hero itself and the tile next to them are matched. The existing 'contains' edge is removed and a new 'contains' edge is created between the hero and the neighboring tile. This rule is not applicable if the neighboring tile contains an obstacle or another character.

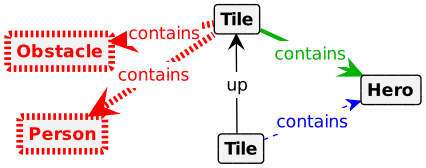

Figure 2: Transformation rule to move the hero

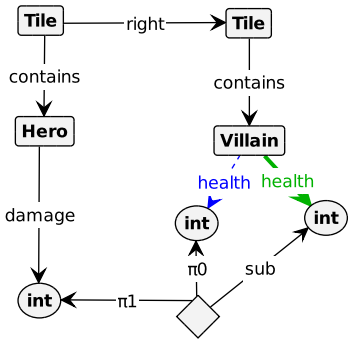

Figure 3: Transformation rule to let the hero attack

The characters must also be able to attack. Its implementation in GROOVE is shown in figure 3. Two adjacent tiles, one containing the hero and one containing a villain, are matched together with that hero and that villain. The damage value of the hero is subtracted from the health value of the villain. This is accomplished with the operation node that takes parameters $\pi 0$ and $\pi 1$ and writes the result to the integer connected with the 'sub' edge. This result becomes the new health of the villain: the old 'health' edge is deleted and a new 'health' edge from the villain to this result is created.

Figures 2 and 3 merely show one example of the transformation rules defining movement and attack. The edge between the two neighboring tiles can be substituted with another direction in order to move or attack in that direction. The movement and attack transformation rules for the villain are the same, except with the Hero and Villain types reversed. One other difference in the transformation rules for villains is that the villain must have its 'active' flag set. There may be multiple villains. In order for GROOVE to keep track of which villain has moved already in a certain time slice, this 'active' flag is needed.

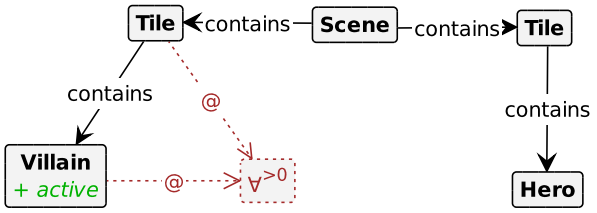

Figure 4: Transformation rule that activates the villains

Figure 4 shows the transformation rule that sets the 'active' flag in all villains in the scene the hero is in. It uses a 'forallx' node to select every villain (at least one) in the same rule and add the 'active' flag. The use of special, auxiliary nodes that stand for universal or existential quantification is a feature of GROOVE that allows advanced nested rules [3].

This rule (figure 4) is applied once every time slice. Then, as long as there are villains that have the 'active' flag set, one villain moves or attacks. During its move or attack, that villain loses its 'active' flag. This keeps happening until there are no villains left with an 'active' flag. This setup allows us to move every villain once in one time slice and is defined in the control program (see section 3.3).

When the hero moves onto a tile that contains an item, a transformation rule to pick up this item is applied. These transformation rules are straightforward: match the hero, the tile containing the hero and the item on that tile. Then the item is removed. If the item was a key, the 'opens' edge to the corresponding door is also removed. This allows us to easily check if a door is locked. All we have to check is if there is an incoming 'opens' edge. If the item was a weapon, the damage value of the weapon and the damage value of the hero are added together, and the result becomes the new damage value of the hero.

There are also transformation rules for when the hero moves onto a special tile. If the hero moves onto a trap, the damage value of the trap is subtracted from the health value of the hero. This is similar to what happens in the attack rules (figure 3). If the hero moves onto an unlocked door, he is automatically moved to the tile with the door it leads to (following the 'leadsTo' edge).

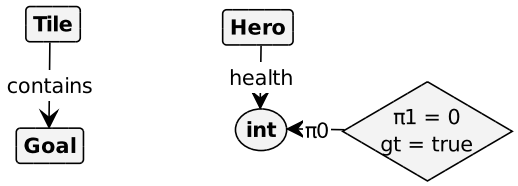

Figure 5: Transformation rule checking the end condition

The simulation continues as long as there are goal items in the world and the hero has a health value greater then zero. This end condition is implemented in a transformation rule shown in figure 5. A goal item on a tile is matched, as well as the hero. The hero's health value is compared with 0, using the 'greater than' operation. The result of this operation is matched as True. This rule doesn't change anything in the model. GROOVE checks if it is applicable. If it is, the end condition is not yet satisfied and the simulation can continue.

3.3. Control program

Figure 6 shows the control program for the RPGame formalism. The outer while loop ensures the simulation keeps running as long as the hero is alive and there are still goals to be collected. This is accomplished by having the 'isNotFinished' rule (see figure 5) as the condition for the while loop. This means that as long as that rule is applicable, the while loop will be executed. One iteration of this while loop represents one time slice.

The first statement in this while loop is a 'choice' statement, which represents a non-deterministic selection between a set of statements. In our case it gives the choice of moving or attacking with the hero. Each choice result in a few rules to be applied. If the hero is moved, we immediately try to pick up an item, activate a trap or use the door. This way, the hero interacts with the tile right after moving on it. If the hero attacks, we immediately check if a villain has died.

After the hero has moved or attacked, the villains in the same scene move or attack. Because there can be multiple villains moving independently of each other, we need a way of identifying which villain has already moved. To accomplish this, we first give every villain in the same scene as the hero an 'active' flag (see figure 4). This is followed by a while loop, which loops as long as there is a villain in the world which has the 'active' flag set. In this loop, another 'choice' statement allows an active villain to move or attack. Upon this move or attack, the villain loses its 'active' flag, identifying that the villain has already moved in this time slice. Unlike when the hero moves or attacks, there is no need for extra statements after the villain moves or attacks. This is because the villain can't pick up an item, can't activate a trap nor use a door. If the hero dies after an attack of a villain, the outer while loop will immediately stop, ending the simulation. Once every villain has moved, the inner while loop ends and a new iteration of the outer while loop, and thus a new time slice, begins.

```
while (isNotFinished){
    choice {Hero.moveDown;
            try Hero.takeGoal;
            try Hero.takeKey;
            try Hero.takeWeapon;
            try Hero.activateTrap;
            try Hero.useDoor;}
        or {Hero.moveLeft;
            try Hero.takeGoal;
            try Hero.takeKey;
            try Hero.takeWeapon;
            try Hero.activateTrap;
            try Hero.useDoor;}
        or {Hero.moveRight;
            try Hero.takeGoal;
            try Hero.takeKey;
            try Hero.takeWeapon;
            try Hero.activateTrap;
            try Hero.useDoor;}
        or {Hero.moveUp;
            try Hero.takeGoal;
            try Hero.takeKey;
            try Hero.takeWeapon;
            try Hero.activateTrap;
            try Hero.useDoor;}
        or {Hero.attackUp;
            try Villain.die;}
        or {Hero.attackDown;
            try Villain.die;}
        or {Hero.attackLeft;
            try Villain.die;}
        or {Hero.attackRight;
            try Villain.die;}

  try Villain.activate;
  while (isActive){
        choice Villain.moveDown;
                or Villain.moveLeft;
                or Villain.moveRight;
                or Villain.moveUp;
                or Villain.attackUp;
                or Villain.attackDown;
                or Villain.attackLeft;
                or Villain.attackRight;
    }
}
```

Figure 6: RPGame control program

4. Analysis

GROOVE has the ability to perform analysis on its models. More specifically, it can perform state space exploration. The simulator will attempt to generate the full state space of a given model. This entails recursively computing and applying all enabled graph transformation rules at each state [4].

Analysing a complete RPGame model and trying to understand its reachability graph is not easy, nor very useful. Therefore we will look at the reachability graph of a small, simple RPGame model. Then we will add different objects to the model and observe the effect it has on the graph. This allows us to gain a deeper understanding of which part of the RPGame influences which part of the reachability graph.

We start with a very simple model: one scene consisting of a three by three grid of tiles (thus 9 tiles in total), with the hero standing on the upper left tile and one goal located on the bottom right tile. The reachability graph generated by GROOVE is shown in figure 7.

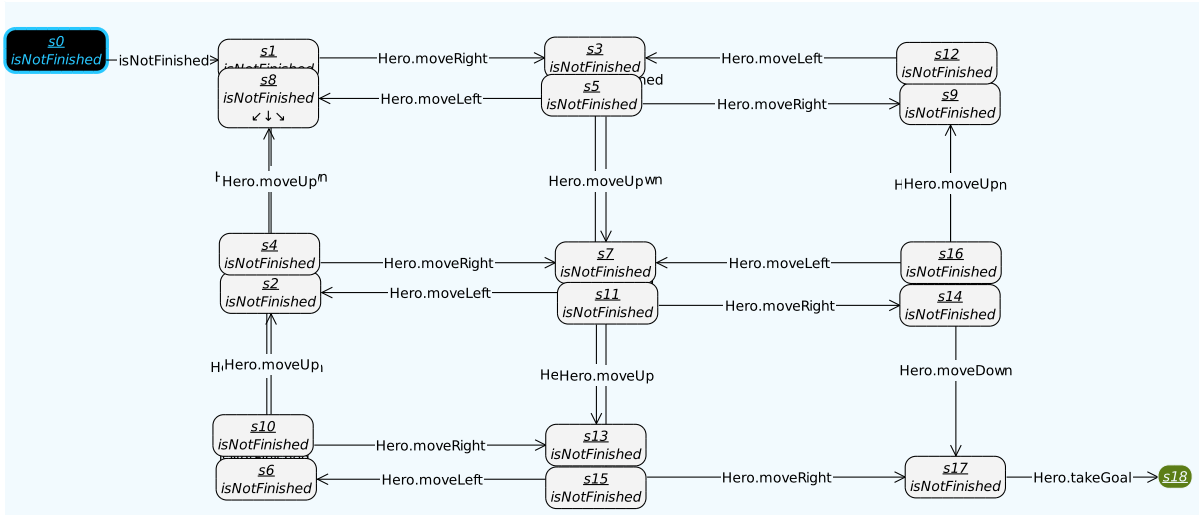

Figure 7: State space of simple RPGame model

The state space generated by GROOVE (figure 7) is not what we would expect. Because there are nine tiles and one hero, we would expect nine states: one for each possible position of the hero. However, GROOVE generates two states for each possible hero position. This is because of the way GROOVE implements state exploration. Starting from the initial state, it will generate a state for the result model of each applicable rule. Following our control program, this means GROOVE will check the end condition, which is the loop-condition, after every iteration of the loop. Even though applying the rule that checks the end condition does not change the model, GROOVE considers it as a new state. If we keep this in mind, the generated state space does conform to what we expected. We also notice that the simulation ends after applying the 'takeGoal' rule in the bottom right corner. This, together with the constant check of the loop-condition, proves that our control program works for this simple example.

If we add an obstacle to one of the tiles and recalculate the reachability graph, the state with the same position as the obstacle is no longer present. This is similar to what would happen if we removed the tile, because no character can stand on a tile with an obstacle. We have already seen that every tile reachable by the hero translate to a state in the reachability graph. Therefore it makes sense that adding an obstacle to a tile is the same as not having the tile, as it is no longer reachable.

Adding an item to one of the tiles has a greater inpact on the state space. Figure 8 shows the generated reachability graph when we add a weapon to the upper right corner of our simple RPGame model. The entire reachability graph from our original game (figure 7) is duplicated and the two grids are connected in the upper right corner with a 'takeWeapon' edge. The reasoning behind this is that for every position of the hero there are two possible states of the RPGame: one where there is a weapon in the upper right corner and one where there isn't. The hero starts in the state where the weapon is present and he can move between the nine tiles resulting in the original grid in the reachability graph. As soon as the hero moves onto the tile containing the weapon, he is forced (by the control program) to pick it up, changing the state to one where the weapon is absent. The hero is free to move between the nine tiles from this position resulting in the second grid in the reachability graph.

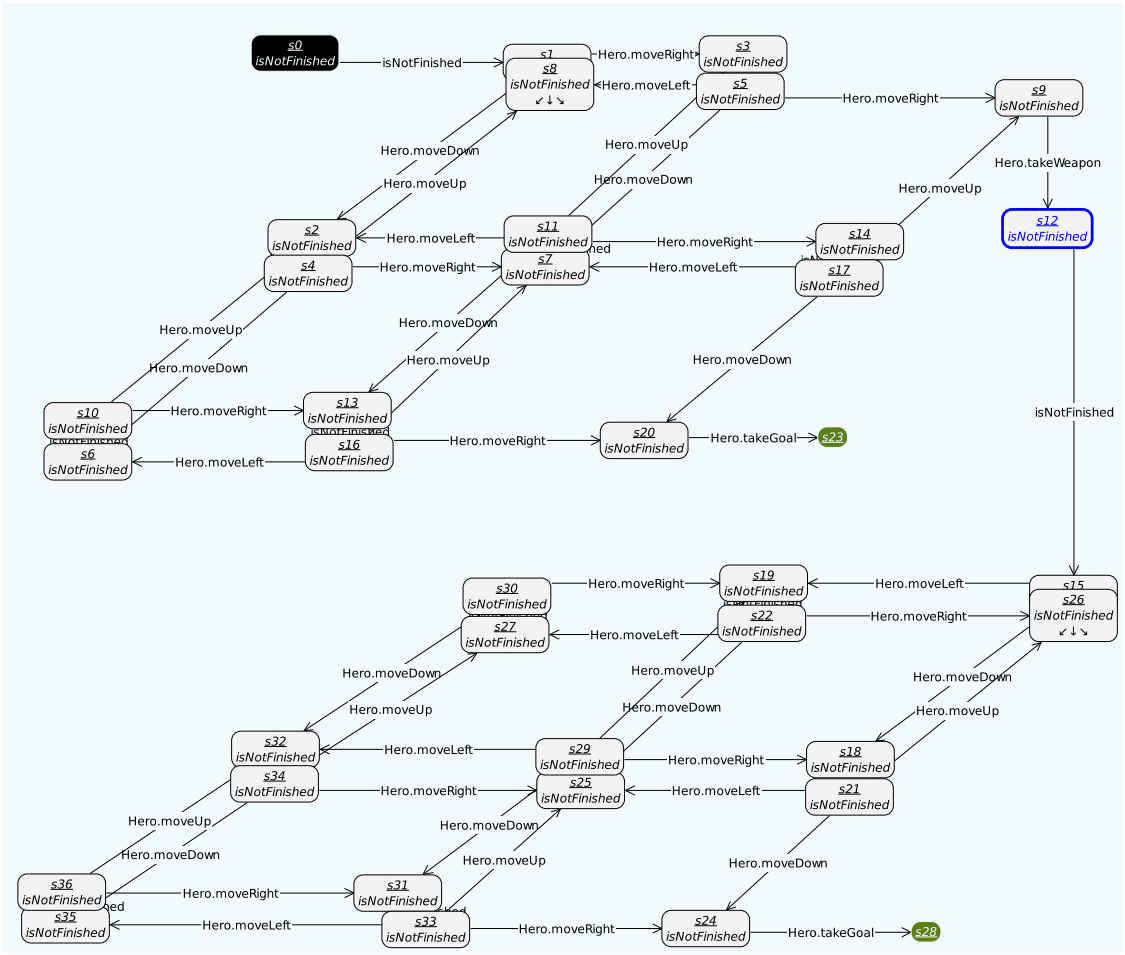

Figure 8: Effect of an item on the state space.

More grids are generated when more items are added. However, adding one item does not add one grid. Adding an item duplicates the current reachability graph. Thus adding another item to the previous example leads to four grids. If we consider that each item is either on a tile or picked up, we get this general rule: 'if there are n items, there are 2^{n-1} grids'. We subtract one from n because there must be at least one goal, if we consider the goals as items. Let t be the amount of reachable tiles in the model, then we can say that the amount of states in the reachability graph is $2^{n-1}t$ (for models containing no traps, no doors and no villains).

If we add a new scene and connect it with the original scene using two doors, it has the same effect on the reachability graph as when we would connect the tiles containing the doors. The amount of reachable tiles simply increases, expanding our original grid with the tiles from the new scene. Adding a key that unlocks the door changes the reachability graph according to the established rules. There is a state for every reachable tile before the key is picked up. When the key is picked up, the hero moves to a new state from which it can move again between tiles. Now, however, the amount of reachable tiles has increased because the door is unlocked. Thus, before the key is picked up, we have the original grid. Then this is duplicated because we added the key (which is an item) and this duplication is extended with all reachable tiles that were previously blocked by the locked door.

Adding a trap has a similar effect to adding an item. Standing on the trap decreases the hero's health value, changing the state and therefore duplicating the grid. The trap is not removed after it is triggered. Therefore the hero can activate it again, duplicating the grid again. It can duplicate the grid as long as the hero has health greater than zero. If n equals the hero's health divided by the trap's damage, then adding the trap causes n duplicates of the original grid. If the hero moves onto the trap in the last duplication of the grid, his health becomes zero and the simulation stops because the end condition is no longer satisfied.

Lastly, we take a look at an RPGame with a villain. Figure 9 shows the first few states of the reachability graph generated when we add a villain to the bottom right corner of our original model. It is clear that this no longer represent the reachable tiles, instead it is a tree-like structure. We can explain this with the following reasoning: in the initial model, which is the root of the tree, the hero can choose between a few different moves. Each of these possible moves results in new model, each represented by a child of the root in the tree. In any of these new models, the villain can choose a move, resulting in a new level of child nodes in the tree. After this, the hero can choose again, and then the villain and so on. We get a decision tree. This decision tree, or search tree, is the same tree which would be used by an artificial intelligent agent. Such agent would generate this tree in order to run a search algorithm like A* or minimax [5].

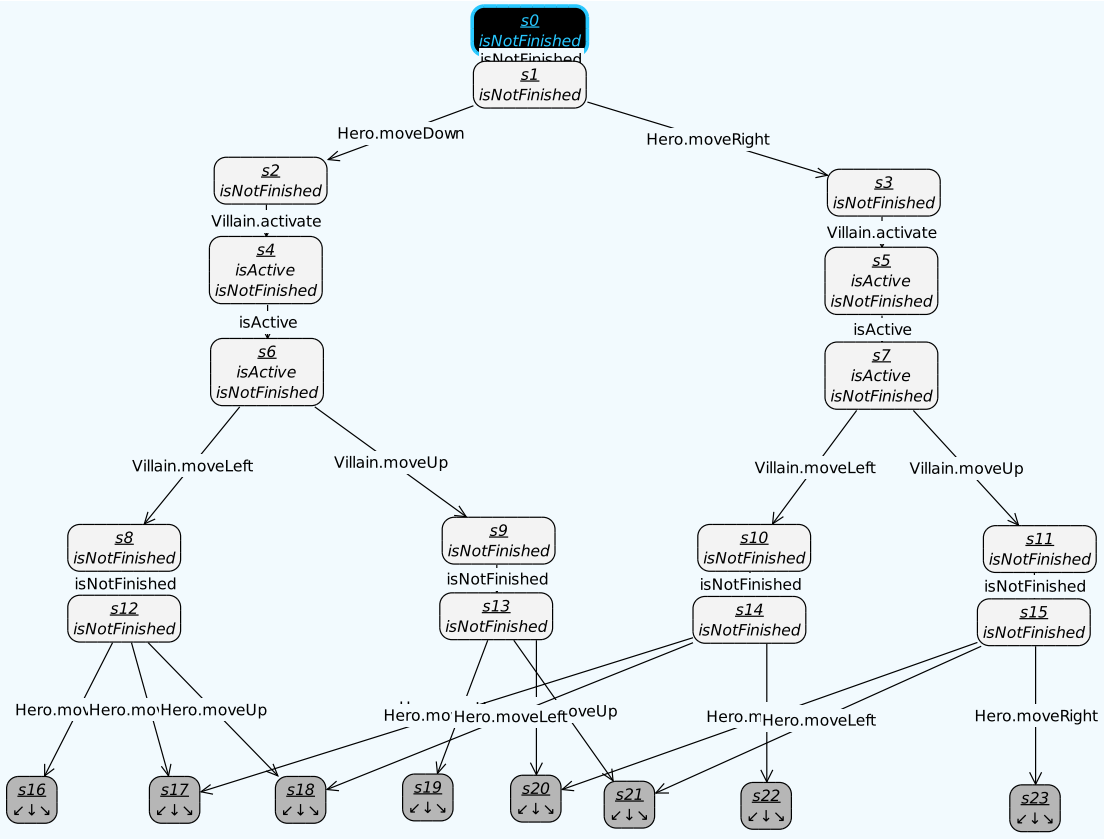

Figure 9: State space of RPGame with one hero and one villain.

Note that it is possible to reach the same state using different paths of the decision tree. For example, moving the hero down and up when the villain moves left results in the same state as moving the hero right and left when the villain moves left (see figure 9).

5. Conclusion

The use-case in this paper shows us that it is easy to implement a relatively complex formalism in GROOVE. We have seen how we can use the many features provided by GROOVE to solve the different problems we encountered while modelling in an elegant way. Above that, GROOVE provides a tool for in-depth analysis of these models which allowed us to gain a better understanding of our RPGame formalism.

References

[1] M. de Mol, A. Rensink, E. Zambon, Groove website, about, http://groove.cs.utwente.nl/about/ (2014).

[2] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, M. Zimakova, Modelling and analysis using groove, International Journal on Software Tools for Technology Transfer (STTT).
URL http://doc.utwente.nl/77423/

[3] H. K. Arend Rensink, Iovka Boneva, T. Staijen, Groove user's manual, groove website (2012).

[4] A. Rensink, The groove simulator: A tool for state space generation, in: J. Pfaltz, M. Nagl, B. Bohlen (Eds.), Applications of Graph Transformations with Industrial Relevance, Vol. 3062 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 479–485. doi:10.1007/978-3-540-25959-6_40.
URL http://dx.doi.org/10.1007/978-3-540-25959-6_40

[5] S. J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 2nd Edition, Pearson Education, 2003.