Combination of Domain-Specific Languages

Practical Research

Rafael Ugaz

1 Introduction

Domain Specific Languages (DSLs) are compact and focused languages for specifying systems at a high-level of abstraction while restricting the solution to a specific domain. DSLs are an essential part of Domain Specific Modelling (DSM), a branch of Model-Driven Engineering (MDE). A DSL can be used to describe a view of a system, however, the use of small DSLs means that they have to be combined together in order to form a complete system. In previous work [7], we have performed a literature review on the current techniques for DSL combination where we also identified the areas that require further research. The next step in this research is to implement these techniques and experiment DSL combination ourselves. But for this to be possible, preliminary enabling technology is needed that consists in the creation of the DSLs to be combined and the complete implementation of a DSM solution to transform them into running systems in a target environment.

The running example of this work is a role-playing game (RPG) that can be run in the Android platform. The first and main DSL designed is used for modelling this RPG in a visual environment. The rest of the DSLs are designed as extensions for the RPG DSL. One for creating graphs and calculating the shortest paths between two of its nodes, and the last one is for reading user input and processing it.

This paper is organized as follows: Section 2 provides the necessary background on Domain Specific Modelling. Section 3 describes the DSLs that were designed for this project. Section 4 presents the details of the code generation process. Section 5 discusses the future work that the work of this paper is going to enable in more detail.

2 Background

The work explained in this paper centres around the creation of the enabling technology for transforming a model created in a high level modelling tool into an application that can be run on an Android platform. The process of performing this transformation is called a DSM solution.

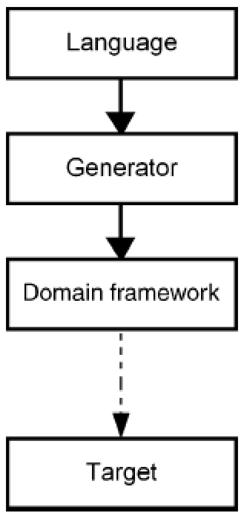


Figure 1: Basic architecture of DSM [2]

The main background necessary for this paper consists of Domain Specific Modelling (DSM) solutions and its components.

The main focus of this work is the creation of a DSM solution for creating RPG games for the Android platform. A DSM solution allows for the automated generation of running systems in a target environment from high level models, that are modelled using the formalism. This provides benefits such as improved productivity, quality and complexity hiding [2]. DSM solutions have a three-level architecture, as illustrated in Figure 1, on top of the target environment:

- A DSL is the most, or sometimes only, visible part for the users (e.g. developers or designers). It is the equivalent of source code in traditional programming with the addition that it provides an abstraction specific to a particular problem domain. Thus, it allows the designers to perceive themselves as working directly with domain concepts. In the case of the RPG DSL, it provides a specific visual modelling environment for the sole purpose of designing role-playing games.
- The generator handles the extraction of information from the models in order to transform it into code that can be run in the target environment of the DSM solution. In the ideal case, the generated code does not needed to be manually modified any further. The underlying target environment usually provides a framework on top of it that is joined with the generated code to form the final product (e.g. an executable). The generation process can also be comprised of multiple languages and transformations, as in the case of this work.
- A domain framework provides the interface between the generated code

and the underlying target platform. It is usually formed by utility code or elements that simplify the generated code (e.g. by extracting duplicated code). This code can sometimes be reused from earlier development efforts, before a DSM approach was used. For the RPG DSL, the domain framework is primarily composed of Java code for displaying the Android application (concrete syntax) and meta-model constructs that are reused along different generated artefacts.

The generated code and the domain framework are not executed by themselves but in some target environment. This environment contains platform code and is used regardless of how the implementation is done, manually or using generators (DSM approach). In this work the Android platform is used as the target environment.

3 DSLs

In this section we explain the test case DSLs that have been created for this project. Currently, three formalisms have been designed: one main DSL for modelling role-playing games (RPGs) and two extension DSLs that add new features to it. The first extension DSL allows the creation of a directed graph and the calculation of the shortest path from one node to another node of the graph. The intent is that this DSL adds pathfinding capabilities to the RPG DSL. The purpose of the second extension DSL is to detect and process user input. This can provide user input handling to the RPG DSL, allowing the user to have influence on the game (which is otherwise a simulation since it has no user intervention).

These extension DSLs can also be used independently but they can be much more valuable when combined with the RPG DSL to produce a final DSL for modelling RPGs that has pathfinding and user input handling capabilities. What interests us in this research is the process of their combination.

Each of the DSLs explained in this section are part of a Domain-Specific Modelling (DSM) solution.

3.1 RPG

A role-playing game (RPG) is a computer game where players assume the roles of characters in a fictional setting. The games created with this DSL fall more into the sub-category of RPGs called *dungeon crawlers*. As the name indicates, the player has to explore a dungeon taking the role of a character. In the dungeon the player encounters enemies that it has to fight, items to collect and quests to complete.

As mentioned before, the RPG DSL has a central role with respect to the other DSLs of this work, namely, they are designed to extend it. In the following sections, each language component of the RPG DSL is described.

3.1.1 Syntax

The abstract syntax of the RPG consists of all the structural elements that take part in the game. At the top most level, the game contains one or more *scenes* or "levels" where all the other structural elements are contained. In each scene, there are a number of *tiles* that can be connected to each other from the left, right, top or bottom. This way, a two-dimensional map is created for each scene. The game also contains characters, a hero and villains. There can be exactly one hero but multiple villains. All characters can stand on one tile at the same time and a tile can hold at most one character.

There are four different types of tiles: an *obstacle* tile on which no character can stand, a *door* tile, which is connected to another door tile and can be used by a character to move form one scene to the other, a *trap* tile, where a character loses life points whenever it stands on it and a regular tile, with no added effects.

On a regular tile, there can be an *item*. There are three types of items: *goals*, can be picked up by the hero (its purpose is explained in the semantics), *weapons*, that when picked up give a bonus in attack to the character and *keys*, which are used to unlock doors (and enable a door for use).

The abstract syntax (meta-model) of the RPG DSL has been defined using the class diagram formalism of the AToMPM meta-modelling tool. This is a pretty straightforward translation from the domain concepts of a role-playing game into class diagram constructs (classes and associations). The RPG meta-model, defined using class diagrams, can be seen in Figure 2. Note that only structural concepts are modelled here, there are no *operations* defined in any class construct since the behaviour of the system is modelled using a different formalism (model transformations). This way, the structure and behaviour of the game (abstract syntax and operational semantics respectively) are cleanly separated and modular.

The *concrete syntax* of the DSL gives each abstract syntax element a visual representation. It is a simple one-to-one mapping between each metamodel class or association and an svg graphic or bitmap image. An instantiated RPG model, using the visual concrete syntax, is illustrated in Figure 3.

3.1.2 Semantics

Since a role-playing game has an execution, its semantics are *operational*. They specify the behaviour of the abstract syntax elements explained in the previous section and describe explicitly how the game can be executed or *simulated*.

The operational semantics can be described with the following rules:

- The game is turn-based, first the hero gets a turn to either move or attack, then all the villains get a turn to do the same, then it goes back to the hero and the cycle repeats itself until the end game conditions are met.
- A character can move from one adjacent tile to another, as long as the tile is not an obstacle tile and is not occupied by another character.
- A character can attack an enemy that stands in any adjacent tile.

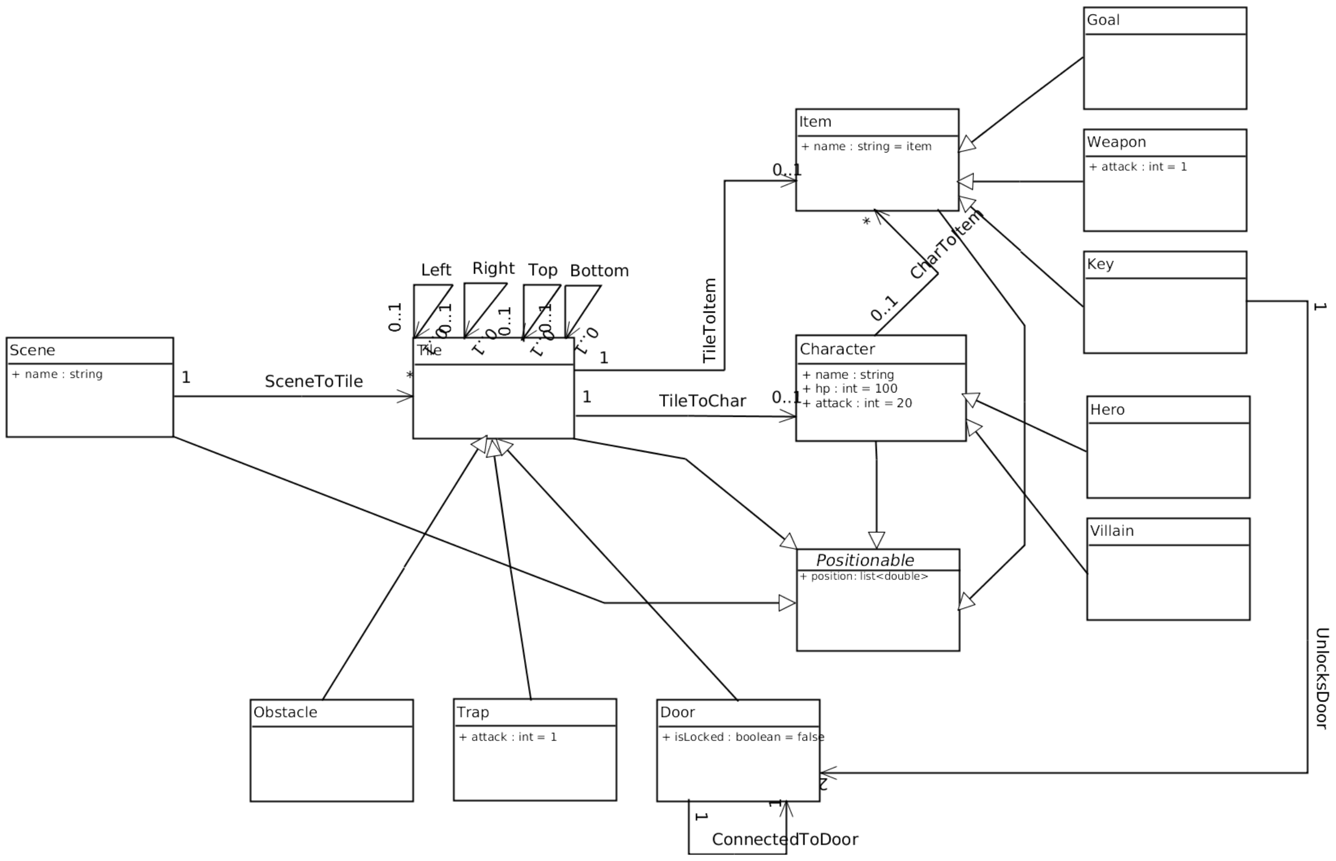


Figure 2: Abstract syntax of the RPG DSL, specified using a class diagram

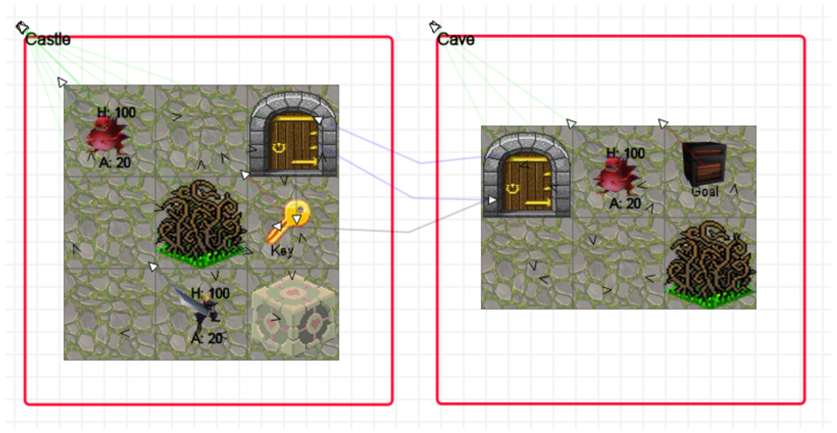


Figure 3: An instance of the RPG DSL, with concrete syntax

- An item can be picked up by the hero by walking on its tile. Every item can only be picked up once.
- If the hero picks up a weapon, the damage of the weapon is added to the hero's attack.
- If the hero picks up a key, he can use a door that is unlocked by that key.
- The player wins the game if he picks up all the goals. There must be at least one goal at the beginning of the game.
- The player loses the game if the hero is killed.

These operational semantics rules have to be modelled in some way into the RPG DSL. For this purpose we have chosen *rule-based model transformations*, a type of model transformation. Model transformation is a sub-field of Model-Driven Engineering (MDE) that allows the manipulation of models. This technique can be used for transforming a model that represents a state of the game into a new one which is what effectively happens in the game just described.

Rule-based model transformations use graph-rewriting rules to specify how a source model is to be transformed. They are composed of a Left-Hand Side (LHS) and a Right-Hand Side (RHS) pattern, an optional Negative Application Condition (NAC) pattern. The LHS and NAC patterns respectively describe what sub-graphs should and should not be present in the source model for the rule to be applicable. The RHS pattern describes how the matched LHS pattern should be transformed by its application. Rule-based transformations by themselves are not of much use, they require a transformation *schedule* to determine the order in which they are applied to the *host* model (the subject of the transformations). For the case of the RPG, the schedule models the turn-based loop between the hero and the villains and every action (moving,

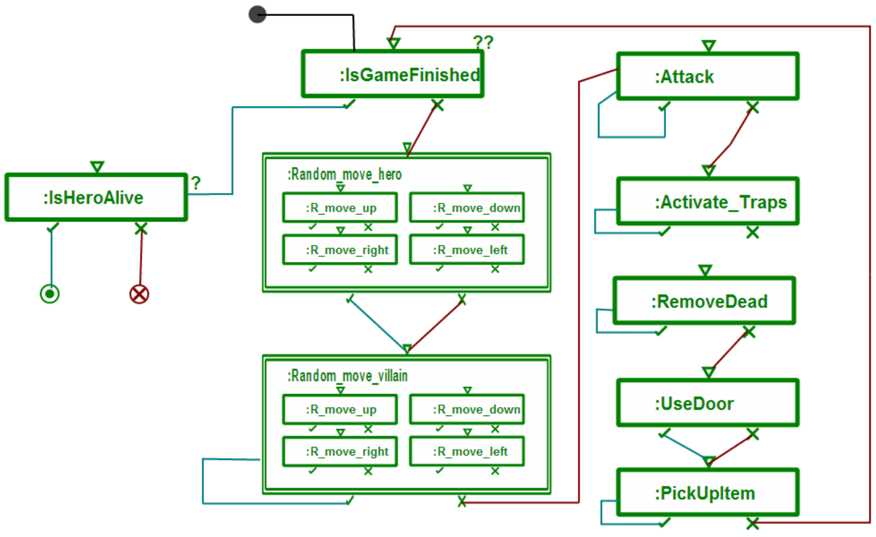


Figure 4: Operational semantics of the RPG DSL, defined using a transformation schedule

attacking, picking up items) that can happen in them. The schedule for the RPG DSL can be seen in Figure 4.

3.2 Pathfinding

The pathfinding DSL allows the creation of a weighted graph with a single source and destination nodes. After creating such a graph, the operational semantics of the DSL can be executed in order to find the *shortest route* between two nodes. The calculation of the shortest path is based on Dijkstra's algorithm, which has been completely modelled using rule-based model transformations.

The purpose of this DSL is to extend the RPG DSL with pathfinding capabilities. Pathfinding is at the core of any kind of artificial intelligence (AI) in games. It allows to find paths between any two coordinates of the game world. An example of a feature possible because of pathfinding is to have a villain move towards a Hero to attack it and *chase* it. This is a big improvement compared to the default random movement behaviour of the RPG DSL.

3.2.1 Syntax

The abstract syntax of the pathfinding formalism consists of the following elements:

- A *Node* that represents the entity of the same name in the search graph. For our concrete case this is a location in the game map, e.g. a Tile. A node can be a regular node or two special types: source and destination. There must be exactly one source and one destination in the graph for the algorithm to work correctly. The Node also contains attributes required for the Dijkstra's algorithm like the *distance* to the destination node and a *isVisited* flag.

- An *Adjacent* edge denotes that two nodes are connected in the search graph.
- A *Previous* edge which is used after the application of the algorithm to point to the shortest path found. The path can be obtained by navigating from the destination towards the source following the Previous edges.

The abstract syntax of the Pathfinding formalism, like the RPG formalism, has been defined using class diagrams in AToMPM.

The concrete syntax of the Pathfinding formalism is similar to the concrete syntax of a undirected graph with the addition of annotations for specific attributes of Dijkstra's algorithm, e.g. distance, edge weights, and whether a node is visited.

3.2.2 Semantics

The operational semantics of the Pathfinding formalism model Dijkstra's algorithm completely, including its calculation of the node with the minimum distance.

The semantics of Dijkstra's algorithm can be divided in the following steps:

1. Assign to every node an initial distance value: *zero* for the initial node and *infinity* for all other nodes.
2. Mark all nodes as *unvisited*. Set the initial node as *current*. Create a set of the unvisited nodes called the unvisited set consisting of all the nodes.
3. For the current node, consider all of its unvisited neighbours and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbour B has length 2, then the distance to B (through A) will be $6 + 2 = 8$. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked *visited* (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.

6. Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3.

These steps have been modelled using the MoTif transformation formalism in conjunction with the rule-based transformations formalism to produce a main schedule containing two sub-schedules (denoted by double edged rectangles) and around 15 rule models. The main schedule model can be seen in Figure 5.

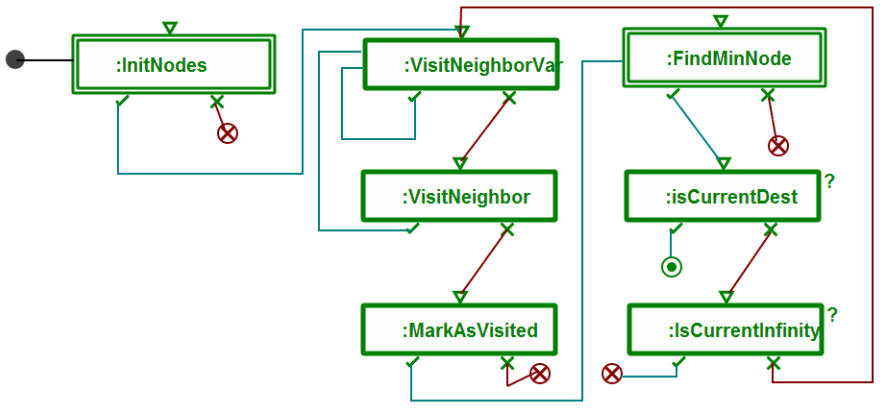


Figure 5: Schedule model of the Pathfinding DSL defined using the MoTif Transformation formalism

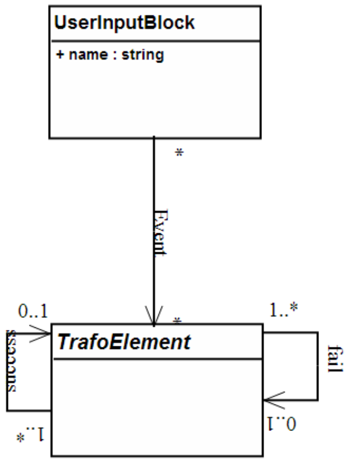
3.3 User Input

The purpose of the User Input DSL is to allow the creation of models that handle the input of the user and produce events that can be passed to rules in a transformation schedule.

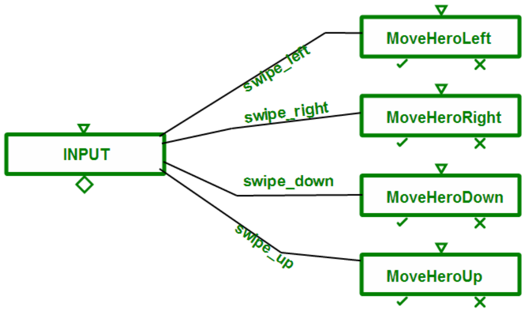
With the current version of MoTif (the transformation language being used in this work), this is not possible and therefore most of the choices in the operational semantics of the RPG are being done randomly.

The abstract syntax of the User Input DSL can be seen in Figure 6a and is based on the MoTif [5] DSL, since the idea is for them to be merged together at a later stage of this project. It consists of a *UserInputBlock* class, that takes care of processing the input of the user and an *Event* association from this class to a *TrafoElement* class. TrafoElement is the class that all other rule blocks inherit from in the MoTif metamodel, e.g. AtomicRules, BranchRules and Success or Fail states. The Event association contains the information of the input received, e.g. what key was pressed, by who and when.

Most of its design has to be implemented with code, as opposed to the previous formalisms (RPG and Pathfinding) that were designed using models



(a) UserInput metamodel



(b) Instance model of the UserInput DSL

almost exclusively, because AToMPM does not provide any support for user input handling. For this reason also, the User Input formalism can only be executed and tested in Java and not in AToMPM. In Figure 6b, an example can be seen of how the UserInput would seem, after being integrated with the MoTif formalism, in a transformation schedule for the RPG DSL.

4 Code Generation

In this section we describe the process of transformation that models of the explained DSLs (section 3) undergo in order to become runnable applications for the Android platform. This process is composed of two different types of elements: formalisms and transformations. The term formalism is used here in a general way so that it includes modelling and programming languages as well as executable code. Transformations convert instances of one formalism into instances of another. An instance is a single occurrence of a model defined in a formalism. This terminology is the same as that of FTGPM diagrams [4], that have been used to illustrate the code generation process explained in this section.

A FTGPM illustrating the whole code generation process can be seen in Figure 7. On the left part of the diagram we can see the Formalism Transformation Graph (FTG). The FTG contains the set of languages involved in the generation process (depicted by a rectangle) as well as the available transformations between those languages (labelled as small circles). On the Process Model (PM), at the right side of Figure 7, the control flow between the different tasks that are necessary to produce the Android platform executable is shown. The labelled round edges are actions, i.e. executions or transformations declared in

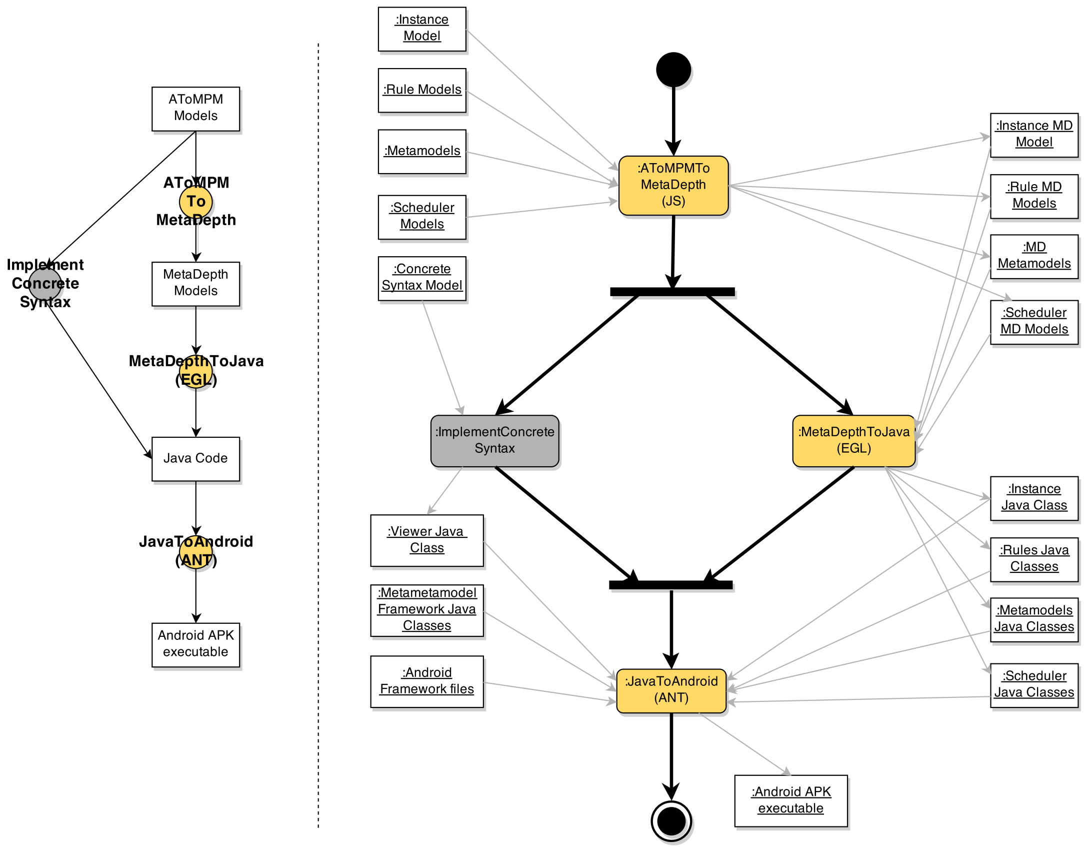


Figure 7: Code generation process, from AToMPM to Android, using a FTG+PM diagram

the FTG and the labelled square edged rectangles correspond to models (instances) that are consumed and produced by the actions. On the PM side, a thin arrow indicates data flow and thick arrows indicate control flow. Manual transformations are shown greyed out in the FTG and PM diagrams.

In the next sections, we explain each language and transformation that takes part in the code generation process.

4.1 AToMPM

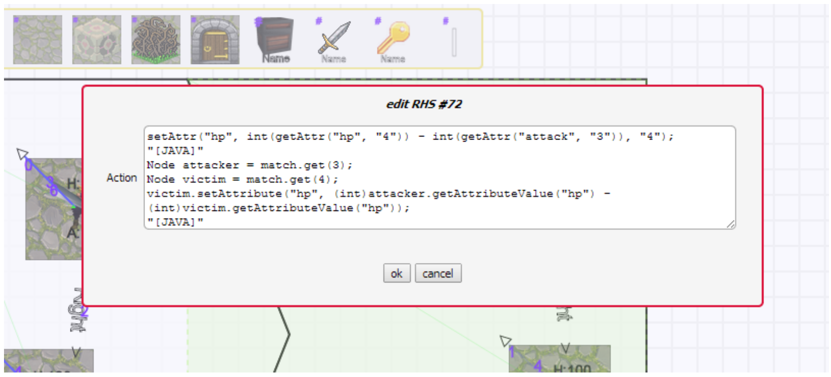
AToMPM [6] stands for *A Tool for Multi-Paradigm Modelling* and is an open-source framework for designing and engineering DSLs. All the DSLs of this work have been designed using this tool. AToMPM allows for the creation of all the components of a DSL, abstract, concrete syntax and semantics. In AToMPM the metamodels used for defining these three components are modelled as well, this is done by using a metametamodel (that is also modelled). Therefore, they can all be inspected and modified with this tool. AToMPM features extensibility by supporting the addition of third party scripts or *plugins* for added functionality. The transformation explained in subsection 4.3 is implemented mainly using plugins. All models created using this tool are stored in files that use the JavaScript syntax and it is in this form that they are processed by the corresponding transformation.

AToMPM contains a built-in engine for performing rule-based model transformations, which consists of a rule matcher and a transformation scheduler. Any DSL whose operational semantics are defined using rule-based transformations can thus be simulated inside the tool.

This engine also allows for the execution of action and condition code inside transformation rules, although it only supports Python or JavaScript code. Since the target environment of our work is Android, primarily developed in Java, it is not possible to use any action/condition code of existing models that use either Python or JS. To solve this issue, we added Java code to the action and conditions of rules alongside the existing Python/JS code. The Java code has to be surrounded by the "[JAVA]" tags in order to be parsed correctly by the following transformations. In Figure 8, an example of the action code of a rule can be seen where the first line contains the Python code and the lines surrounded by the "[JAVA]" tags contain the equivalent Java code.

A feature that would have been useful during the course of this work is association inheritance for language syntax formalisms like Class Diagrams. This is currently not possible because AToMPM only supports edges that are between two non-edge entities, i.e. nodes. For the case of Class Diagrams, this would allow the formalism to be extended with an inheritance link between it's associations, effectively allowing association inheritance for any instance models defined in it.

AToMPM provides a concrete syntax metamodel for defining the visual representation of formalisms. Unfortunately, this only allows for a partial modelling



```
setAttr("hp", int(getAttr("hp", "4")) - int(getAttr("attack", "3")), "4");
"[JAVA]"
Node attacker = match.get(3);
Node victim = match.get(4);
victim.setAttribute("hp", (int)attacker.getAttributeValue("hp") -
(int)victim.getAttributeValue("hp"));
"[JAVA]"
```

Figure 8: Action code in AToMPM using Java syntax

of the concrete syntax of a formalism since it is not explicit enough. It supports a mapping between each abstract syntax element and a visual construct (in svg format or image). What it does not support are more complex properties of a language that belong to the concrete syntax but that a designer is forced to design using the abstract syntax language (e.g. class diagrams). For example, in the RPG DSL, it is desirable that the Tiles of a Scene are arranged in the two-dimensional next to their neighbours, so if Tile *a* has a *top* link going to Tile *b*, then Tile *b* is automatically positioned above *a*. This had to be implemented within the RPG abstract syntax model while it clearly belongs to the concrete syntax. For this reason, the translation of the concrete syntax is currently done via an ad hoc manual transformation (represented by the gray color in FTGPM diagrams) and not automated like the other models. Since the DSLs in this work do not have a complex concrete syntax model, this is not considered as a major issue.

4.2 MetaDepth

MetaDepth [1] is a textual meta-modelling environment which has as main advantage over AToMPM that it supports the execution of EGL templates.

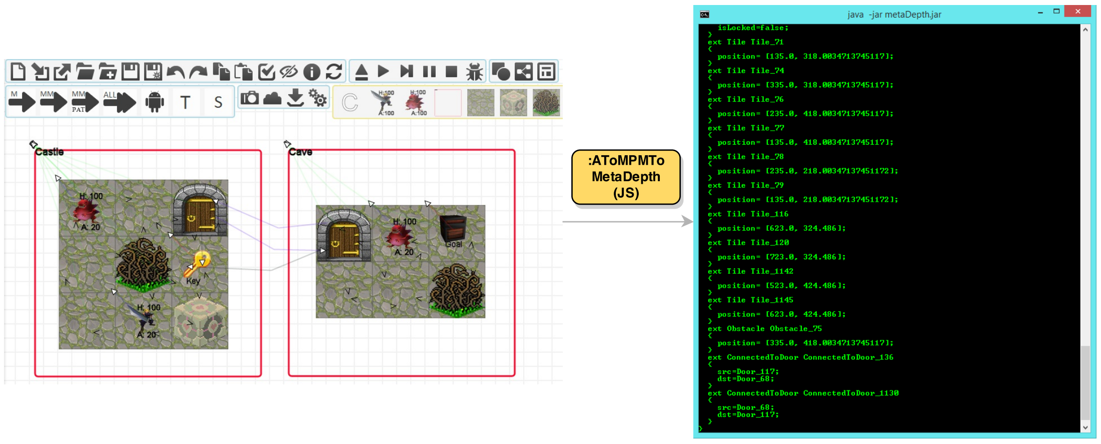
MetaDepth also gives the option for validating models by checking any constraints included by the meta-modeller. Constraints are currently not supported since they also have to be converted into the right format before MetaDepth can process them.

Adding MetaDepth to the code generation process allows us to use EGL but also makes it more convoluted and more prone to error. Before an EGL template can be executed on a MetaDepth model, the model has to be loaded correctly into the program. This adds a new layer of complexity since it requires the exporter of AToMPM to be complete and to cover all possible errors that the transformed model could originate. This transformation, implemented in JavaScript is explained in detail in subsection 4.3.

4.3 AToMPM to MetaDepth: Javascript

After a model has been correctly loaded into MetaDepth, the appropriate EGL template can be executed to produce a text file. The first transformation converts models defined in AToMPM (in JS format) to models in the MetaDepth format. This transformation is performed using a modified version of an existing plugin included in the latest version of AToMPM. The downside of the original plugin is that it can only process one model at a time, i.e. the one that is currently loaded in the model editor of AToMPM. Thus, a user has to manually load every model it wishes to export and then click on the export button of the plugin. This is very inconvenient, specially for systems that span through a large amount of models, e.g. the RPG DSL requires an instance model, metamodels and (rule) transformation models to be exported.

For this reason, the plugin has been modified in order to support automatic export of all the required models with a single click from the user. This plu-



```
   isLocked=false;
  }
  ext Tile Tile_71
  {
   position= [135.0, 318.0034713745117];
  }
  ext Tile Tile_74
  {
   position= [335.0, 318.0034713745117];
  }
  ext Tile Tile_76
  {
   position= [235.0, 418.0034713745117];
  }
  ext Tile Tile_77
  {
   position= [135.0, 418.0034713745117];
  }
  ext Tile Tile_78
  {
   position= [235.0, 218.00347137451172];
  }
  ext Tile Tile_79
  {
   position= [135.0, 218.00347137451172];
  }
  ext Tile Tile_116
  {
   position= [623.0, 324.486];
  }
  ext Tile Tile_120
  {
   position= [723.0, 324.486];
  }
  ext Tile Tile_1142
  {
   position= [523.0, 424.486];
  }
  ext Tile Tile_1145
  {
   position= [623.0, 424.486];
  }
  ext Obstacle Obstacle_75
  {
   position= [335.0, 418.0034713745117];
  }
  ext ConnectedToDoor ConnectedToDoor_136
  {
   src=Door_117;
   dst=Door_68;
  }
  ext ConnectedToDoor ConnectedToDoor_1130
  {
   src=Door_68;
   dst=Door_117;
  }
}
```

Figure 9: AToMPM-to-MD

gin first prompts the user to select two models: the host and transformation (i.e. schedule) models and then it exports them along with all other necessary models for their correct functioning. The automatically exported models are obtained by navigating through the dependencies of these two selected models (e.g. metamodel toolbars and imported rules). For the case of a transformation model, these are all the rule models (and their pattern metamodels) that are contained in it as well as any sub-transformations. For an instance model, the plug in exports the metamodel(s) that are loaded into the editor at the time of export.

Other minor changes made to the original plugin are: escaping of non-alphanumeric characters in any variable names of the model (permitted in AToMPM but not in MetaDepth), as well as modifying any variable names that use MetaDepth's reserved words (e.g. Node, Edge, Model) so that the models can be properly loaded into MetaDepth. Finally, special characters like slash, backslash and quotations are properly escaped.

An additional script that was implemented is one for performing the RAMification [3] of a regular metamodel in order to generate a pattern metamodel (in MetaDepth). This script performs the basic actions from the RAMification process. It removes the abstract classes of the model (relaxation), adds label attributes to be used by the transformation engine (augmentation) and lastly it modifies the data type of the attributes of all model elements so they can express conditions or actions (modification).

4.4 Java

Java is a popular general-purpose programming language that can be used to write Android applications ("apps"). It has a much lower level of abstraction compared to the previous meta-modelling languages.

4.4.1 Multiple Inheritance

A difference between Java and the previous languages is that Java does not support multiple implementation inheritance. It only supports multiple interface inheritance which only allows the classes to inherit method definitions (methods with an empty body). This poses a problem since the previous languages allow full implementation inheritance and a model that has to be transformed into Java (using EGL) could contain this feature. The choice that was made was to keep the multiple inheritance and recreate or simulate it in Java. The alternative was to stop supporting this feature in the previous languages but this would limit the modelling capabilities of the designers by depriving them from this feature.

For recreating multiple inheritance in Java, we had to recreate inheritance altogether. This was done by adding a subclass tree as a hash map data structure to each metamodel. Then, at the moment of creation of the metamodel object, all the types that are defined in it are stored as *keys* and all their subtypes as *values*. Using the subclass tree as well as extra framework functions, it is possible to determine any inheritance relationship between two objects of the model. Since Java's regular inheritance is not used in the generated classes, no attributes or methods are being inherited by subclasses. Therefore, this had to be done manually in the generation step (subsection 4.5) by effectively duplicating all attributes and methods of a superclass in all its subclasses. This clearly goes against object oriented design but is the only choice since Java's regular inheritance is not being used.

The drawbacks of this approach are that all the functionality that Java provides for inheritance is lost for the generated models. No polymorphism is possible since, for Java, no inheritance relationship exists between any classes. This means that objects have to be *casted* to a class, given that the name of such class is known beforehand. If the name of the class of an object is not known beforehand it is not possible to cast it and therefore impossible to access the inherited attributes of the object.

For example, using the RPG DSL from subsection 3.1, one could desire to read the attribute *Position* of an object of kind *Tile* (i.e. of type Tile or a subtype thereof). It would not be possible to simply cast the object to Tile since it could have the type *Trap* or *Door* (both subtypes of Tile) and this would throw an exception and the program would fail.

To bypass this problem, we use Java Reflection which makes it possible to inspect classes, fields and methods at runtime. Using Reflection one is able to read an attribute of an object by passing its name as a string to a Java Reflection method. In the same example as before, for the object of kind Tile, its Position can be obtained with at runtime without having to perform any casts. Reflection has the downside that it could perform operations that would be illegal in non-reflective code, such as accessing private fields and methods and it can also produce other unexpected side-effects. For the concrete case of this project, it made debugging a bit more difficult since it moves the functionality of accessing and modifying fields from compile-time to runtime and instead of

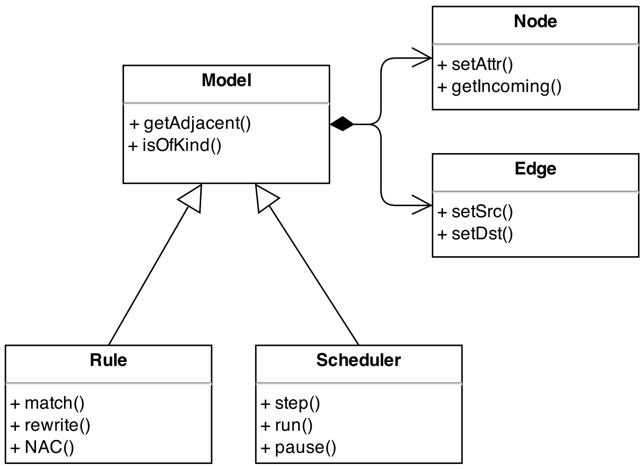


Figure 10: Java framework design

receiving errors with the compiler, an exception is thrown.

4.4.2 Framework

Being Java the last programmable language of the transformation process (the last being Android's APK package format), it contains all the framework needed for the DSM solutions discussed before. The main functionality of the framework is the metametamodel for all languages that are implemented. The structure of the metametamodel can be seen in Figure 10. It consists of the following classes:

Model The *Model* class that contains helper methods for querying and manipulating a generated model. All generated model classes are subclasses of Model. Some examples of the functionality added by this class is:

- $getNodesOfKind$ for obtaining a list of nodes of the same type and subtype of a given node.
- *getAdjacent* for obtaining all the nodes that are connected to a given nodes via an edge.
- *add* and *remove* for manipulating the model's elements.

Node The *Node* class represents an element of a model. It is equivalent to the Class construct of class diagrams. Every node object of a model defined in AToMPM, is transformed into a MetaDepth Node clabject and finally it is converted into a subclass of a Java Node. This framework class contains helper methods such as the following:

- *getAttribute* and *setAttribute* for manipulating a node's attributes (using Java's Reflection). This is needed as part of the solution for the lack of multiple inheritance in Java.
- *getIncoming* and *getOutgoing* returns the incoming and outgoing edges of a node, respectively.

Edge The *Edge* class contains the information of an edge or link of the previous languages of the transformation. The equivalent in class diagrams of AToMPM are associations and in MetaDepth the clabjects of the type *Link* (a manually added class to replace the built-in and problematic Edge construct of MetaDepth). This class contains methods for accessing and modifying the source and destination nodes of an edge.

RuleModel The *RuleModel* class is a template for all transformation rules, defined using the formalism of the same name in AToMPM. Having this all generated rule models inherit from this class gives us the advantages of polymorphism and also avoids having duplicated code for each generated rule.

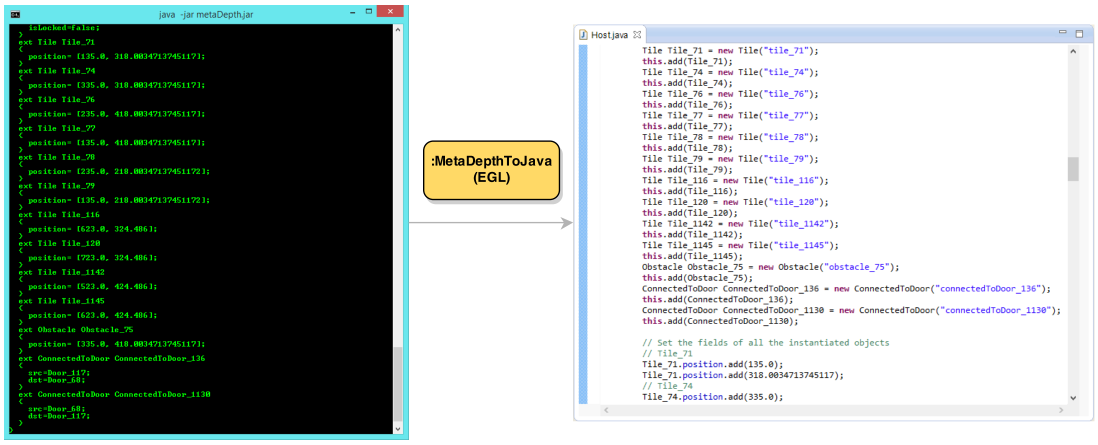
ScheduleModel The *ScheduleModel* class serves as a template as well, but for transformation schedules. Furthermore, it provides the same benefits as the RuleModel framework class.

4.5 MetaDepth to Java: EGL

As mentioned in subsection 4.1, AToMPM models are stored using the JavaScript syntax and can be queried and modified with JS scripts as well. In theory, this is enough for creating a code generator using JavaScript. However, JavaScript is not a very appropriate language for code generation compared to the available alternative: the Epsilon Generation Language (EGL). EGL (subsection 4.5) is a model-to-text template-based language for generating code that can be applied to any language that conforms to the model interface of the Epsilon Family of Languages. Since AToMPM currently does not implement this interface, it cannot be used directly with EGL templates. For this reason we added the MetaDepth tool to the code generation process, which supports the execution of EGL templates. Furthermore, the latest version of AToMPM includes a plugin for exporting AToMPM models into MetaDepth models.

Once a model has been translated into MetaDepth's syntax, it can be loaded on the tool and next, an EGL template can be executed. There are four kinds of models that are currently supported for transformation to Java, and each of them requires a different EGL template:

- Metamodels, defined using the Class Diagram formalism.
- Instance models, defined using any metamodel that is defined using the Class Diagram formalism.



```
  isLocked=false;
 }
 ext Tile Tile_71
 {
  position= [135.0, 318.0034713745117];
 }
 ext Tile Tile_74
 {
  position= [335.0, 318.0034713745117];
 }
 ext Tile Tile_76
 {
  position= [235.0, 418.0034713745117];
 }
 ext Tile Tile_77
 {
  position= [135.0, 418.0034713745117];
 }
 ext Tile Tile_78
 {
  position= [235.0, 218.00347137451172];
 }
 ext Tile Tile_79
 {
  position= [135.0, 218.00347137451172];
 }
 ext Tile Tile_116
 {
  position= [623.0, 324.486];
 }
 ext Tile Tile_120
 {
  position= [723.0, 324.486];
 }
 ext Tile Tile_1142
 {
  position= [523.0, 424.486];
 }
 ext Tile Tile_1145
 {
  position= [623.0, 424.486];
 }
 ext Obstacle Obstacle_75
 {
  position= [335.0, 418.0034713745117];
 }
 ext ConnectedToDoor ConnectedToDoor_136
 {
  src=Door_117;
  dst=Door_68;
 }
 ext ConnectedToDoor ConnectedToDoor_1130
 {
  src=Door_68;
  dst=Door_117;
 }
}
```

```java
Tile Tile_71 = new Tile("tile_71");
this.add(Tile_71);
Tile Tile_74 = new Tile("tile_74");
this.add(Tile_74);
Tile Tile_76 = new Tile("tile_76");
this.add(Tile_76);
Tile Tile_77 = new Tile("tile_77");
this.add(Tile_77);
Tile Tile_78 = new Tile("tile_78");
this.add(Tile_78);
Tile Tile_79 = new Tile("tile_79");
this.add(Tile_79);
Tile Tile_116 = new Tile("tile_116");
this.add(Tile_116);
Tile Tile_120 = new Tile("tile_120");
this.add(Tile_120);
Tile Tile_1142 = new Tile("tile_1142");
this.add(Tile_1142);
Tile Tile_1145 = new Tile("tile_1145");
this.add(Tile_1145);
Obstacle Obstacle_75 = new Obstacle("obstacle_75");
this.add(Obstacle_75);
ConnectedToDoor ConnectedToDoor_136 = new ConnectedToDoor("connectedToDoor_136");
this.add(ConnectedToDoor_136);
ConnectedToDoor ConnectedToDoor_1130 = new ConnectedToDoor("connectedToDoor_1130");
this.add(ConnectedToDoor_1130);

// Set the fields of all the instantiated objects
// Tile_71
Tile_71.position.add(135.0);
Tile_71.position.add(318.0034713745117);
// Tile_74
Tile_74.position.add(335.0);
```

Figure 11: MD-to-Java

- Transformation rule models, defined with the TransformationRule metamodel (subsubsection 4.5.1).

- Transformation rule schedule models, defined with the MoTif [3] metamodel (subsubsection 4.5.2).

Note that for this approach only one template is required for the metamodels of all the different formalisms involved. This is because all of their metamodels have been defined using the same metamodel, i.e. class diagrams.

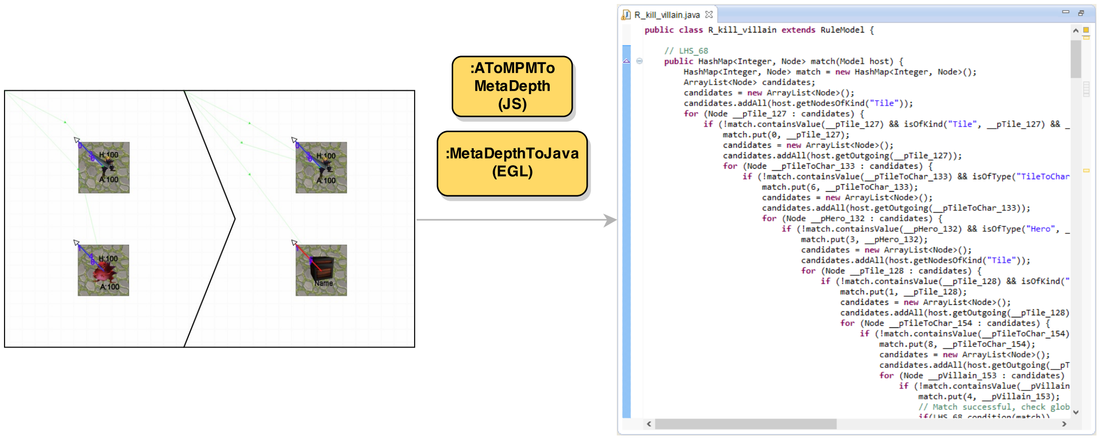
4.5.1 Rule

Generating the rule and schedule models is only the first part of the work. What remains is being able to apply these rules to host models, using a schedule model to determine the order. In order to apply a transformation rule to a host model and produce a new host model, a technique similar to pattern matching has to be performed. Applying a transformation rule involves the following three steps:

1. Try to *match* **all** the elements of the left-hand side (LHS) pattern to the host model.

2. If the matching is successful (no single element of the LHS was left out of the match) then try to match the negative condition (NAC) pattern to the host model.

3. If the NAC was **not** matched, then the match was successful and the right-hand side (RHS) pattern can be applied to the matched elements. The RHS performs the actual transformation of the model.



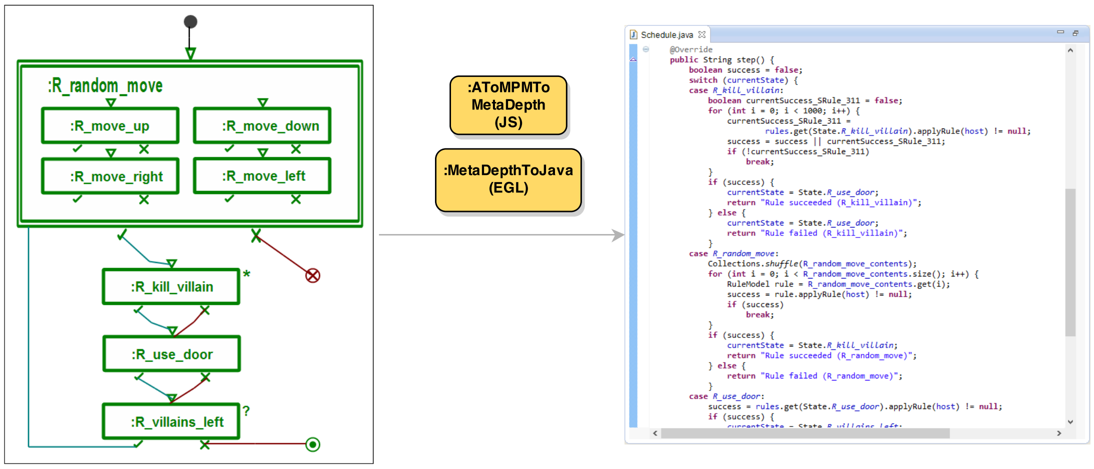
```java
public class R_kill_villain extends RuleModel {

    // LHS_68
    public HashMap<Integer, Node> match(Model host) {
        HashMap<Integer, Node> match = new HashMap<Integer, Node>();
        ArrayList<Node> candidates;
        candidates = new ArrayList<Node>();
        candidates.addAll(host.getNodesOfKind("Tile"));
        for (Node __pTile_127 : candidates) {
            if (!match.containsValue(__pTile_127) && isOfKind("Tile", __pTile_127) && _
                match.put(0, __pTile_127);
                candidates = new ArrayList<Node>();
                candidates.addAll(host.getOutgoing(__pTile_127));
                for (Node __pTileToChar_133 : candidates) {
                    if (!match.containsValue(__pTileToChar_133) && isOfType("TileToChar
                        match.put(6, __pTileToChar_133);
                        candidates = new ArrayList<Node>();
                        candidates.addAll(host.getOutgoing(__pTileToChar_133));
                        for (Node __pHero_132 : candidates) {
                            if (!match.containsValue(__pHero_132) && isOfType("Hero", _
                                match.put(3, __pHero_132);
                                candidates = new ArrayList<Node>();
                                candidates.addAll(host.getNodesOfKind("Tile"));
                                for (Node __pTile_128 : candidates) {
                                    if (!match.containsValue(__pTile_128) && isOfKind("
                                        match.put(1, __pTile_128);
                                        candidates = new ArrayList<Node>();
                                        candidates.addAll(host.getOutgoing(__pTile_128)
                                        for (Node __pTileToChar_154 : candidates) {
                                            if (!match.containsValue(__pTileToChar_154)
                                                match.put(8, __pTileToChar_154);
                                                candidates = new ArrayList<Node>();
                                                candidates.addAll(host.getOutgoing(__pT
                                                for (Node __pVillain_153 : candidates)
                                                    if (!match.containsValue(__pVillain
                                                        match.put(4, __pVillain_153);
                                                        // Match successful, check glob
                                                        if(LHS_68.condition(match))
```

Figure 12: Rule-to-Java

We considered two approaches for implementing the rule matching technique. One was to create a centralized engine (as part of the Java framework) that takes as input a rule and a host model, applies the rule and then produce the new transformed host model. The other was to add the rule matching functionality inside each generated rule model, specific to the rule. For the first option, the pattern matching algorithm would be implemented using Java and for the second using EGL (to generate Java files). Even though Java is a better suited language for implementing such an algorithm, doing it at the generation stage has the advantage of simplifying the algorithm since it only has to work for the specific rule where it is located. Creating a generic matching engine was deemed as a more complicated process. It also would have performed less optimally since it had to be able to process all the possible rules of the domain. Therefore the second option was chosen with the only downside being that EGL was not as expressive and powerful as Java for this task.

The matching engine for each rule consists of up to two methods for matching the LHS and/or NAC patterns and one method for transforming the matched elements. The matching algorithm consists of two parallel depth-first searches (DFS) on the nodes of the host model and those of the pattern to be matched. This has been implemented by generating a chain of nested *for* loops, one for each node of the pattern. Inside each *for* loop, all the possible matching candidates (nodes of the host model) are compared to the pattern node of the *for* loop until a match is found, in which case the next nested *for* loop is entered, or until no more candidates are left, in which case the current for-loop is exited (i.e. it backtracks) and the previous one continues iterating through its candidates. The generated algorithm also takes into account the case of unconnected group of nodes in the host model, i.e. strong components. This is solved by detecting the strong components of the host model beforehand and then restarting the previously explained DFS on each of them.



```java
    @Override
    public String step() {
        boolean success = false;
        switch (currentState) {
        case R_kill_villain:
            boolean currentSuccess_SRule_311 = false;
            for (int i = 0; i < 1000; i++) {
                currentSuccess_SRule_311 =
                        rules.get(State.R_kill_villain).applyRule(host) != null;
                success = success || currentSuccess_SRule_311;
                if (!currentSuccess_SRule_311)
                    break;
            }
            if (success) {
                currentState = State.R_use_door;
                return "Rule succeeded (R_kill_villain)";
            } else {
                currentState = State.R_use_door;
                return "Rule failed (R_kill_villain)";
            }
        case R_random_move:
            Collections.shuffle(R_random_move_contents);
            for (int i = 0; i < R_random_move_contents.size(); i++) {
                RuleModel rule = R_random_move_contents.get(i);
                success = rule.applyRule(host) != null;
                if (success)
                    break;
            }
            if (success) {
                currentState = State.R_kill_villain;
                return "Rule succeeded (R_random_move)";
            } else {
                return "Rule failed (R_random_move)";
            }
        case R_use_door:
            success = rules.get(State.R_use_door).applyRule(host) != null;
            if (success) {
                currentState = State.R_villains_left;
```

Figure 13: Schedule-to-Java

4.5.2 Schedule

The schedule of a transformation also requires an engine in order to execute its control flow in the target environment. This was implemented following the same approach as the transformation rules, embedding the functionality to each individual schedule model instead of creating a centralized generic scheduling engine to serve all of them. The schedule main functionality is contained within the *step* method, which attempts to apply one rule and, according to whether the attempt was successful or not, it passes control to the next rule.

The current scheduler EGL transformation supports the following MoTif's rule blocks that can be defined using this language:

- ARule (atomic), applies the rule on one match.
- SRule (star), applies the rule recursively until no matches are found.
- QRule (query), applies a match for the LHS.
- BRule (branch), non-deterministically selects one successful branch of execution.
- CRule (composite), refers to another (sub-)transformation.

An example of the input and output of applying the scheduler

4.6 Android

Android is an operating system for mobile devices such as smartphones that we have chosen as the target environment of this project. We are considering Android as a language even though the actual language it uses is Java because a Java application still has to undergo a transformation before it can be used as an Android *app*. The files have to be packaged in the .apk format which contains the executable byte code files. This packaging is performed by using the Ant tool to execute tasks provided by the Android Software Development Tool (SDK).

4.6.1 Framework

In order to run the generated Java code as an Android app, a framework is required. This framework is also implemented using Java but it uses the Android API and is therefore specific to this section.

The first part of the framework is the *Activity* class. An Activity is the entry point of an Android app and it provides the screen with which the user interacts. The layout of an app is contained in the Activity.

The second framework element is the View that is used for displaying the actual model, e.g. the game world map in the RPG DSL. The View is analogous to the concrete syntax metamodel of the AToMPM environment, its purpose is to display all the contents of a given model in the canvas of the environment. Since the concrete syntax metamodel of AToMPM is not being used in the generation process (reason explained in subsection 4.1), each of the formalisms of this work, e.g. RPG or Pathfinding, require a separate View class that takes care of its display. In this section we explain in detail all the transformation steps of the code generation process of our DSM solution. We use three different transformations that are applied to a model for translating it to each of the languages explained in section 4. An overview of this process, using the RPG formalism as example, is illustrated in Figure 7.

4.7 Java to Android: ANT

The last transformation converts all the generated and framework Java files to the APK package format that can be executed in an Android device. This transformation is performed using the Apache Ant command-line tool used for automating the software build process and is geared towards Java projects, like the one at hand. Using Android's SDK we have implemented an Ant script for automatically compiling the source files, building the APK, installing the APK on a device and finally running it in such device. This script has been integrated into AToMPM's plugin for exporting models (See subsection 4.1) and is executed after the models have been converted to Java format, allowing the designer to perform the complete generation process (consisting of the three sub-transformations of this section) with only one click.

5 Future Work

Having the test DSLs defined and their generation process automated sets up the field for experimenting with the language combination techniques of [7].

They shall be applied to the main RPG DSL in order to create an extended one which is able to creating games that can be run on the Android platform and that have extra features such as pathfinding, user input, debugging capabilities, along with other possible future extensions.

Since the RPG and Pathfinding languages have been defined using the same formalisms, that is class diagrams for their abstract syntax and rule-based transformations for their semantics, their combination falls into the category of homogeneous. In order to combine them, the abstract syntax and semantics models have to be joined. This task could be much more difficult if the formalisms with which the languages were designed were different. For example if the RPG semantics were defined using rule-based transformations and the Pathfinding language using state charts. In that case, a translation would have to be performed from one formalism to the other or they could both be translated into a formalism in between.

Regarding the combination of their abstract syntax, the technique that currently seems most appropriate is combination by *inheritance*. This could be done by letting a *Tile* (and thus all its subtypes) of the RPG DSL be a subtype of the *Node* type of the Pathfinding DSL. This would allow tiles to be regarded as graph nodes to which the pathfinding algorithm could be applied.

For the semantics, the transformation of the Pathfinding DSL would remain untouched, since it could still be applied to tiles after the abstract syntax combination. But the semantics of the RPG DSL would have to be modified in order to use the new information given by the Pathfinding DSL, namely the shortest path between two tiles. The intent is that, instead of having a character moving to a random tile on its turn, it would move to the tile that brings him in the right direction towards its goal (e.g. an enemy character or an item that it wants to pick up). The exact way the semantics of these two DSLs would be combined is still not clear and is the subject of the future work that will focus solely in DSL combination.

References

[1] Juan De Lara and Esther Guerra. Deep meta-modelling with metadepth. In *Objects, Models, Components, Patterns*, pages 1–20. Springer, 2010.

[2] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. Wiley-IEEE Computer Society Press, 2008.

[3] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit transformation modeling. In *Models in Software Engineering*, pages 240–255. Springer, 2010.

[4] Levi Lucio, Sadaf Mustafiz, J. Denil, B. Meyers, and Vangheluwe H. The formalism transformation graph as a guide to model driven engineering. 2012.

[5] Eugene Syriani, Jeff Gray, and Hans Vangheluwe. Modeling a model transformation language. In *Domain Engineering*, pages 211–237. Springer, 2013.

[6] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. Atompm: A web-based modeling environment. In *Demos/Posters/StudentResearch@ MoDELS*, pages 21–25, 2013.

[7] Rafael Ugaz. Weaving of domain-specific modelling languages a literature review. 2014.