Model Driven Software Development Approach on Procedural Modeling of Buildings

Murat Kurtcephe, Oğuzcan Oğuz, Buğra M. Yıldız
Bilkent University Computer Engineering Dept.
Ankara, Turkey
kurtcephe, oguzcan @cs.bilkent.edu.tr
b_yildiz@ug.bilkent.edu.tr

Abstract— **To construct virtual city models, sufficiently detailed and variously styled building models are required. Traditional production of such models demands extensive manual work and time. This process could be automated using the procedural methods. The aim of this work is to realize a DSL for procedural building modeling. Two approaches, a MOF based from-scratch approach and UML2 profiling, are applied to define the meta-model of the intended DSL. Moving from this DSL, model-to-model and model-to-text transformations are done using ATL and Xpand languages.**

Keywords-Model driven software development; Procedural modeling of buildings ; shape grammar; MOF; UML; Model-to-Text Transformation; Model-to-Model Transformation. ATL; openArchitectureWare Xpand.

I. Introduction

Modeling and visualization of large and complex environments is a popular research area in computer graphics. Recent developments in processors and graphics cards, the amount of available memory, and the development of computer graphics modeling and rendering techniques facilitate to run high quality simulations. To this end, virtual cities should be modeled in order to be used in such simulations.

One major component of a virtual city that affects the realism is the building models. In a virtual city, building models should have a high level of geometric detail and be consistent with the architectural style of the virtual environment. Today's machine capabilities facilitate the visualization of both large and complex 3D models. Applications cover a large spectrum, from military training and city planning to video games and tourism. Modeling exactly existent cities to be used in such applications can be tedious and is not intended in this work. Modeling each and every building in detail by hand using 3D models is inefficient, even the use of aerial images or airborne laser scan data requires a great deal of manual work.

For solving the problem defined above, we realize a domain specific modeling language in this project. The DSL's concrete syntax is defined in a textual form so that the users of the architects, designers or any other people who work on building modeling do not have to deal with complex 3D models manually.

The domain specific language is defined by means of meta-modeling, for procedurally generating building models. The meta-modeling process is done using two approaches: MOF-based from scratch meta-modeling and UML profiling mechanism. The identified domain concepts that correspond to entities in these meta-models are based on CGA shape grammar [9].

To achieve interoperability, increased understandability and productivity, we also realized two model transformations. Model-to-text transformation transforms a given model into a pre-defined XML format that interpreted and then, output XML file is used to generate 3D building models by another existing system. Also a model-to-model transformation is defined that is used to transform given model to a less complete but more expressive form, a graph model. The modifications or changes occurring as the result of production rules are easier to see on a simple planar graph sketch of the graph model that is produced by model-to-model transformation.

The paper starts with the domain concepts' descriptions in section II. Following this, DSL grammar is defined in section III. In sections IV and V, meta-model instantiated from MOF and meta-model created by applying UML profiling approach are described. Concrete syntax definition and example models are given in Section VI. Model transformation details can be found in Section VII. Section VIII is about related work done on procedural modeling of buildings. The paper ends with a conclusion section.

II. Procedural building of models

Shape: The grammar works with a configuration of shapes: a shape consists of a string symbol, geometry (geometric attributes) and numeric attributes. A shape is either a terminal shape or a non-terminal shape. The most important geometric attributes are the position P, three orthogonal vectors X, Y, and Z, describing a coordinate system, and a size vector S. These attributes define an oriented bounding box in space called scope (Figure 1). Some of the other geometric attributes of the shapes can be defined with respect to the associated scope(i.e. With respect to the local oriented bounding box of the object)

The variability of the number of geometric and numeric attributes, enables the definition of various shapes with different geometries.

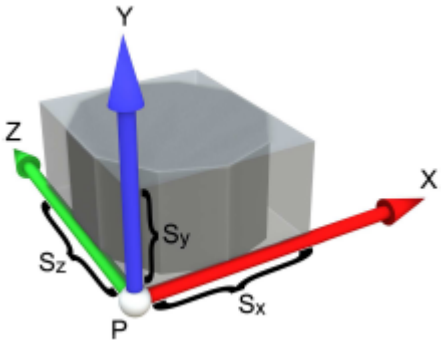


Figure 1. The scope of a shape

A configuration is a finite set of basic shapes. The derivation process can start with an arbitrary configuration of shapes. The initial configuration should, however, include a number of non-terminal shapes. The derivation process is composed of a number of steps. At each step, an active non-terminal shape B is selected in the configuration and an applying rule with highest probability applied to the selected shape. At the end of the step, a set of newly created shapes BNEW is added to the configuration and the shape B is set inactive. The process continues until there are not any active non-terminal shapes left in the configuration. A final configuration contains a number of terminal shapes which defines the geometry of the generated model. To have finer control over the derivation process, each rule assigned a priority. Among the rules that apply to the same shape, the rule with the highest priority is selected and applied to the shape. Therefore, the derivation proceeds from low detail to high detail. Also it is worth to mention that the derivation process should be a sequential process to allow for the characterization of the structure i.e. the spatial distribution of features and components.

Rules

Context-free **production rules** to transform shapes to other shapes. The production rules have the following form:

id: predecessor : cond → successor : prob

where id is a unique identifier for the rule, predecessor is a symbol identifying a non-terminal shape that is to be replaced with the shape with symbol successor, and "cond" is condition (logical expression) that has to evaluate to true in order for the rule to be applied. A rule can have more than one successor with associated probabilities. The successor rule is selected with probability prob.

Successor shapes can be defined by five different ways. The simplest successor shape is just another symbol list defining a set of successor shapes. Other than that a successor shape can be defined by one of the following ways.

Scope Rules: Scope rules define transformations on the scope of the given predecessor shape. A transformation can be a translation denoted by $T(t_x, t_y, t_z)$, a scaling denoted by $S(s_x, s_y, s_z)$ or a rotation around one of the three axes denoted by R_x (angle), R_y(angle) and R_z(angle). By defining a stack, we could save and restore the current scope by pushing it in to save and popping it out to restore. In addition to the rules to modify the scope, an insertion rule creates a given shape with the current scope. An insertion rule is denoted by I(objId). An example is given below which defines the shape in Figure 2.

1: A → [T(0,0,6) S(8,10,18) I("cube")]T(6,0,0) S(7,13,18) I("cube") T(0,0,16) S(8,15,8) I("cylinder")

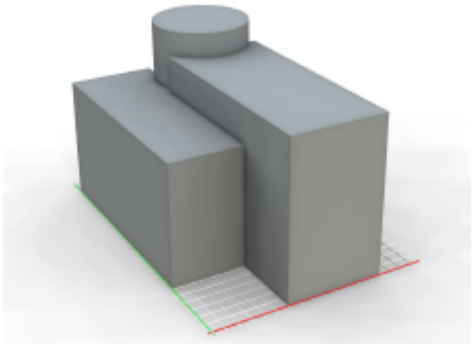


Figure 2. A simple building mass model composed of three shape primitives

Basic Split Rule: The basic split rule is used to split the scope of the given shape along one of three axes. The split sizes and ratios is defined in the rule. For every sub-scope there are listed a number of successor shapes which are to be put into the sub-scope. An example split rule is given below.

1: floor → Subdiv("X",2,1r,1r,2){ B | A | A | B }

The 'r' symbol states that the defined size is relative and is proportional to the associated size of the provided scope. The size parameters without an 'r' symbol are absolute values independent of the size of the scope of the given shape.

Repeat Rule: A repeat rule splits the scope of the given shape along the specified axis into equally sized sub-shapes. The successor shapes are of the same kind given in the body of the rule. Splits are generated for the given scope as long as there is space; the minimum size of a split is stated in the rule. An example repeat rule follows.

1: floor → Repeat("X",2){ B }

Component Split Rule: A component split rule splits the given shape into its lesser dimensional components. The type of the components to be generated is stated in the rule and can be one of: "faces", "edges", "vertices's" or "side faces". A component split rule has the following notation:

1: a → Comp(type, param){ A | B | ... | Z }

The "param" parameter lists the indices of the generated components that are to be replaced by the listed successor shapes. If the "param" list is left empty, then all of the generated components is replaced by the stated successor shape.

III. DSL Grammar

The domain specific grammar is provided below. The start symbol is ShapeGrammar that contains a number of PrioritySet symbols. A PrioritySet has a PriorityID and a set of ProductionRuleList. ProductionRuleList acts as a list of ProductionRule symbols. A ProductionRule has a RuleID, a NonTerminalShape as the predecessor shape, a guard condition and a SuccesorList which is a list of successor various types.

We didn't need to give the further explanation for the well known primitive concepts such as Natural Number, Identifier, boolean and float.

*ShapeGrammer ::= PrioritySet(PrioritySet)**

PrioritySet ::= PriorityID ProductionRuleList

PriorityID ::= Natural Number

*ProductionRuleList ::= (ProductionRule)**

ProductionRule ::= RuleID NonTerminalShape (Condition)? SuccessorList

RuleID ::= Identifier

Condition ::= Boolean Expression

*SuccessorList ::= Successor Probability| (SuccesorList)**

Propability ::= float

Successor ::= Shape | ComponentSplitRule | RepeatRule | ScopeRule | BasicSplitRule

Shape ::= NonTerminalShape | TerminalShape

NonTerminalShape ::= Symbol Scope

TerminalShape ::= EmptyShape | PredefinedShape

EmptyShape ::= E (Empty)

PredefinedShape ::= Symbol Scope FileName

FileName ::= Identifier

Symbol ::= Identifier

Scope ::= X Y Z S P

X ::= Vector

Y ::= Vector

Z ::= Vector

S ::= Vector

P ::= Vector

Vector ::= float float float

ComponentSplitRule::=ComponentSplitType ParameterShapes ParameterInts

*ParameterShapes ::= Shape (Shape)**

*ParameterInts ::= (Natural Number)**

ComponentSplitType ::= Identifier

RepeatRule ::= Axis SplitRulePart

Axis ::= Identifier

SplitRulePart ::= Size isRelative Shape

Size ::= float

isRelative ::= boolean

*BasicSplitRule ::= Axis SplitRulePart (SplitRulePart)**

*ScopeRule ::= ScopeRulePart (ScopeRulePart)**

ScopeRulePart ::= InsertionRule | TranslationRule | ScaleRule | RotationRule | StackRule

InsertionRule ::= InsertionItem

InsertionItem ::= Shape

TranslationRule ::= TranslationAmount

TranslationAmount ::= Vector

ScaleRule ::= ScaleAmount

ScaleAmount ::= Vector

RotationRule ::= RotationAxes Angle

Angle ::= float

RotationAxes ::= Identifier

StackRule ::= StackParameter

StackParameter ::= ScopeRule

IV. Definition of meta-model Based on Mof-from Scratch

In this section of the paper, the definition of the meta-model which is defined based on the MOF (Modeling Object Framework) will be described in detail.

A. Abstract Syntax of the Meta-Model

The abstract syntax of the meta-model is created by using graphical tools. The representation of the model is very similar to the UML model.

It is not possible to describe all abstract syntax by once, so to clarify the diagram which represents the abstract syntax, the diagram will be divided according to the relations between

elements. Each part will be described separately. The complete diagram of the ECore meta-model is provided in Appendix A.

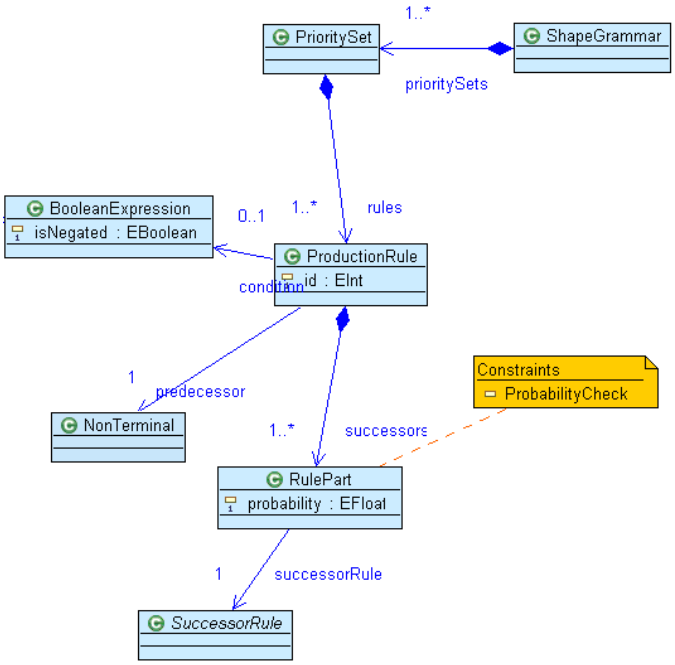


Figure 3. A section of the meta-model.

As it can be seen from Figure 3, as a start point shape grammar concept is selected. A shape grammar has a number of priority sets. Priority sets are used for defining the priority relationship between the production rules. A priority set consists of production rules. Every rule used in this shape grammar is a production rule.

Production rules have ids, they can have a Boolean expression as a guarding condition. Also each production rule has a predecessor which is a non-terminal shape. This non-terminal shape is converted into the successor shapes. A successor shape is a component which consists of a number of rule parts. A rule part is used in the abstract syntax since a production rule can have more than one successor rules.

Before describing the details of the rules, the Boolean expression concept should be defined. Figure 4 shows how to define Boolean expressions which will be used as guards in the production rules. A Boolean expression can be negated. This means when the property is negated is set then the neagtion of the expression should be the result.

Each Boolean expression can be a primitive Boolean expression which has two types: Occlusion test and basic Boolean. Basic Boolean is simply the 'True' and 'False' values of Boolean concept. The occlusion test has three types of occlusion which are defined as an enumeration.

A Boolean expression can be a comparison which takes two float values and compares them by using the comparator sign provided. Comparator signs are defined by using an enumeration.

A Composite Boolean expression consists of two Boolean expressions. It can be used for defining more complex Boolean expressions by using 'and' and 'or' connectors.

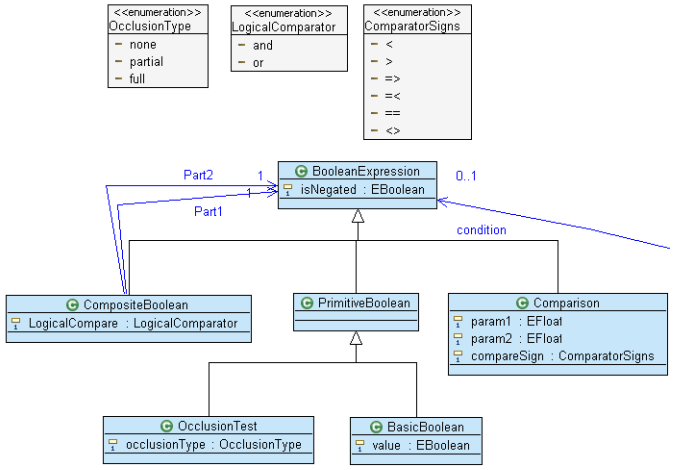


Figure 4. The boolean expression concept

Successor rule has five different kinds: Substitution rule, composite split rule, repeat rule, basic split rule and scope rules. First of all, simple rules will be described. Figure 5 shows the abstract syntax of the rules used in the system.

Component split rules are used when splitting a shape into shapes of lesser or higher dimensions. The type of a component split rule is defined as an enumeration. When component split rule is invoked, the parameter list will hold the indexes of the components, each component can be assigned to a shape. For allowing this there is a relationship between component split rule and shape.

A repeat rule has an axis with which the rule will split the given shape aligned. It takes the size of the split and the shape element which is produced at the end of the split.

As it can be understood from its name basic split rule, splits the given shape into new shapes according to the given size and can produce shapes of different types. A relative split size can be given, for differentiating the normal sizes and relative size a Boolean flag is used.

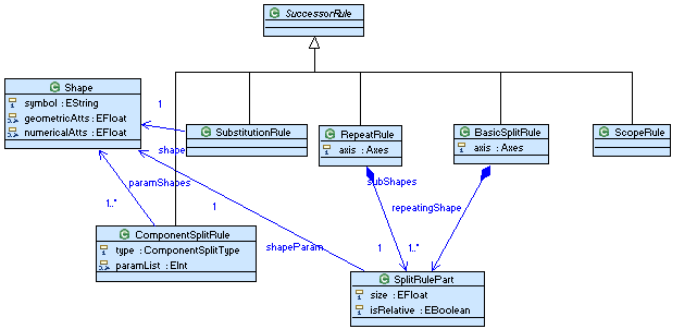


Figure 5. Five types of successor rule

A substitution rule is another kind of successor rule. The rule simply substitutes the shape given as predecessor with a

provided shape. To clarify, details of shape is given in Figure 6.

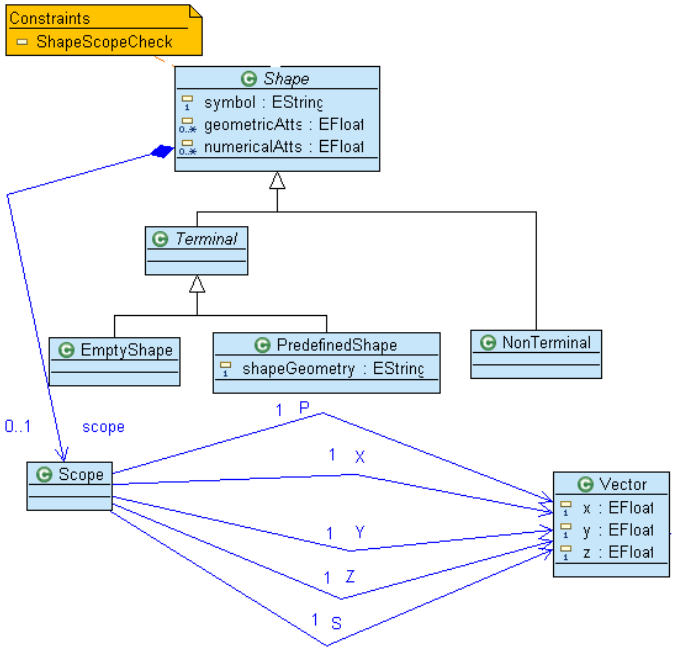


Figure 6. Shape concept

Each shape has a unique symbol. A shape can be terminal or non-terminal. Terminal shapes can be empty shapes or predefined shapes. Each predefined shape has a shape geometry which corresponds to a file name and a path. Non-terminal shapes are the shapes which are defined as in-between shapes and eventually they will be converted to terminal shapes.

Each shape has a number of geometric and numerical attributes. As shown in Figure 6, a shape has a scope which consists of five different vectors named: P, X, Y, Z and S. P gives the reference point of the shape, X, Y and Z is used for describing a local coordinate system. S is a size vector which is the size of the scope.

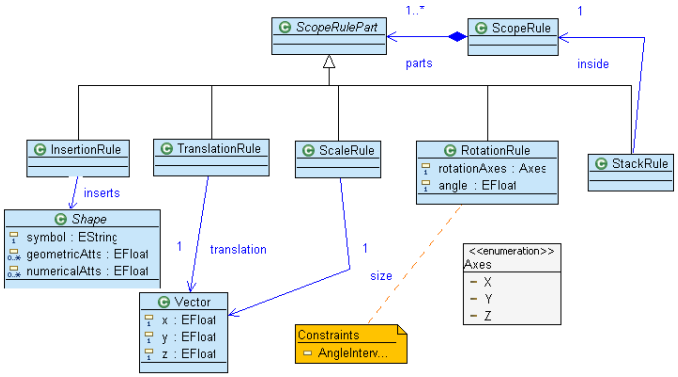


Figure 7. Scope rules

Last production rule defined in the meta-model is scope rule (Figure 7). Scope rules are used for modifying shapes by making operations on their scope. These operations consist of insertion, translation, scaling and rotation. Also there is another concept called stack rule which will be described in detail.

Insertion takes a shape and inserts it to the current scope. Translation rule uses a vector for translating the current scope. Scaling rules take the size of the new scope as a vector. Rotation rule uses rotation axes which are defined by using enumeration and angle for rotating the scope.

Stack rules are used for saving the scope temporarily and restoring it later when it is needed.

B. Static Semantics (OCL)

As it can be seen in the figures of the abstract syntax part, static semantics of the meta-model is defined by using annotation boxes.

Inside these boxes the static semantics are defined by using OCL (Object Constraint Language). These rules ensures the well formedness of shape grammar models. The constraints will be given with their explanation below.

Constraint No	Constraint
1	context RulePart inv: self.probabilty <=1 and self.probability >0
2	context RotationRule inv: self.angle >=0 and self.angle<=360
3	context Shape inv: if self.oclIsTypeOf(Empty) then self.scope->size() = 0 else self.scope->size() = 1 endif
4	context ComponentSplitRule inv: if self.paramList->size() = 0 then self.paramShapes->size() = 1 else self.paramList->size() = self.paramShapes->size() endif
5	context ProductionRule inv: if self.successors->size() > 1 then self.successors.probability->sum = 1 endif

Constraint 1 checks the interval of the probabilities. A probability value can be between 0 and 1 (1 included).

Constraint 2 limits the interval of the angle between 0 and 360.

An empty shape can not have a scope, if it is not an empty shape, then that shape can only have one scope. This is forced by Constraint 3.

Constraint 4 states that when a component split rule has no 'paramList' element then it has to send all components to the same shape, otherwise it has to have same number of `paramShapes` as given in `paramList`. Details can be found on Figure 5.

The successors of a production rule should have probability value '1' when they are summed. This is expressed in Constraint 5.

V. Definition of Meta-model Using Uml 2.0 Profiling

In this section, to define the meta-model another method called 'UML Profiling' is used. UML 2.0 profiling is a mechanism which allows developers to extend and use the components of UML for developing their own meta-model.

A. Abstract Syntax

In our case, UML profile of the domain is very similar to the meta-model defined according to MOF from scratch. In MOF from scratch method we used ECORE elements such as 'EClass', 'EReference' or 'EAnnotation'. In UML profiling 'UML Class' will be used instead of 'EClass'. EAnnotation will not be used instead of that comment boxes of UML can be used.

Not the whole UML profiling diagram will be described in detail because it is very similar to the meta-model based on MOF. Only there are some minor representation differences. Therefore, a small portion of the meta-model will be presented for showing the similarities and differences. The complete diagram of the profile is provided in Appendix B.

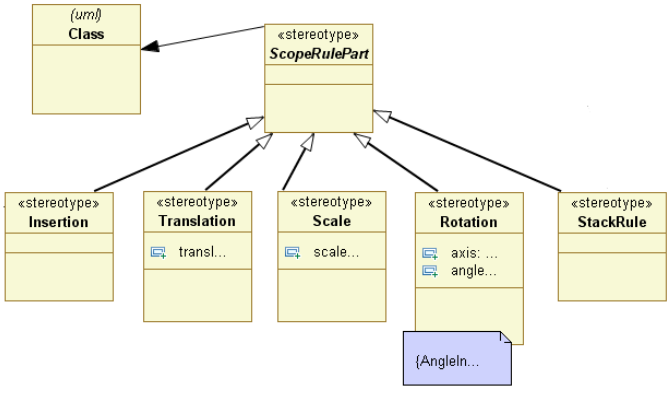


Figure 8. Scope rules by UML profiling

The 'EClass' corresponds to the stereotypes which are extended from UML 'Class'. Generalization relationship is also same. Constraints are written inside of comment boxes which are supported by UML.

B. Textual Profile Description

Since the stereotyped classes are instances of UML:Class entity and there is no tagged value for stereotypes, we didn't include *Base Class* and *Tags* columns in the table. Also Constraints and Description columns are combined. The table is provided in Appendix C.

VI. Concrete Syntax and Example Models

A. Concrete Syntax

Our language's concrete syntax is defined in textual form. Since models' size can be very large and sometimes, recursive definitions are required, textual representation is the best way to cope with these problems. If we are to use a graphical notation, large models would seem complicated and it would be hard to follow links between the rules and the shapes.

"PRIORITY 1:

<ProductionRules>

PRIORITY 2:

<ProductionRules>"

The whole ShapeGrammar object is defined like the example above. A ShapeGrammer defines the whole model and consists of PrioritySets. PrioritySets are defined with *PRIORITY #:* where # is an integer defining priority level. Following the PrioritySet header, the body of the PrioritySet comes. The body consists of ProductionRules. A PrioritySet ends with the header of the following PrioritySet.

"PRIORITY 1:

1: <ProductionRule>

2: <ProductionRule>

3: <ProductionRule>"

Each ProductionRule has an identification number. This is represented as *"#: <ProductionRule>"*, where # is the identification number followed by a *':'* character. ProductionRule is written following *"#: "* string.

*<Predecessor>(:<Condition>)(<SuccessorList>)**

*<SuccessorList> ::= ~><Successor> <Probability>| <SuccesorList>**

This is the structure of a ProductionRule. Predecessor is followed by a *':'* character. Also, Probability is expressed by a *':'* character. Probability is a float and represented as Default English representation such as "5.6" (An integer following a dot for representing decimal points). The Parenthesis content is optional and Condition is represented like Boolean expressions

in Java. The followings are examples: *a> b, 6<>h(not equal)* or predefined functions like *Shape.occ("all")=="none", Shape.occ("part")<>"full"*or just *true/false.*

Terminal shapes have the following representations. If it is an EmptyShape, it is denoted by ε. PredefinedShape is defined as *Pshape("*.obj").* PredefinedShape gives reference to the external Shape object whose file name is *.obj. A NonTerminal shape is defined with an identifier (called as symbol in abstract syntaxes). For example, *door, wall, window*, etc. Each <*Successor*><*Probability*> pair is separated from each other with a *"~>"*.

A Sucessor can be a Shape, or one of the other kinds of SuccessorRules. ComponentSplitRule, one of the SuccessorRules is represented like:

CS(<type>,<parameter>){ A| B| C|...Z}

<*type*> can be *"faces","edges","verticles"* or *"sidefaces"*. <*parameter*> is optional and takes an integer value. The capital letters present the symbol of the resulted shape as the result of split operation. An example for ComponentSplitRule can be *CS("edge", 3){ A}*.

RepeatRule is another kind of SuccessorRules. It is denoted by *"RR"*. Following this, the axis and division size come. The structure is formulated as:

RR(<axis>,<size>){ <Shape>}

<*axis*> can be X, Y or Z as well as combinations of these(XY, YZ, XZ or XYZ). <*size*> can be a float. <*Shape*> corresponds to a Shape's symbol. An example of this rule is: *RR("XY", 3){ cell}*

BasicSplitRule is denoted as "BS". Like RepeatRule, BasicSplitRule also takes <*axis*> as parameter and following <*axis*>, split sizes are given.

BS(<axis>, <sizes>){ <ShapeList>}

<*sizes*> are float values separated by commas. Similarly, <*ShapeList*> is a list of Shape symbols separated by commas. An example can be *BS("XY", 3, 5, 7){ balcony| floor| door}*.

ScopeRule is composed a number of scope rules. The notation for the individual scope rules is given below.

T(<t_x>,<t_y>,<t_z>) denotes a translation rule. The translation t vector is defined by three components.

S(<s_x>,<s_y>,<s_z>) denotes a scaling rule. The scaling factors along the axes are defined by three size values.

R<axis>(<angle>) denotes a rotation rule. <*axis*> states the axis around which the rotation is performed. <*angle*> is the angle of rotations.

I(<symbol>) denotes an insertion symbol. <*symbol*> is the symbol associated with the shape to be inserted.

The push and pop operations are denoted by *'['* and *']'* symbols respectively.

B. *Example Model 1*

The first example model is an instance of shape grammar defined using concrete syntax described before. The ShapeGrammar instance is the whole model in Figure 9 and 10. This ShapeGrammer contains two PrioritySets having priority ids as 1 and 2. PrioritySet 1 has a BasicSplitRule as the ProductionRule. PrioritySet 2 has the ProductionRules starting from 2 to 8.

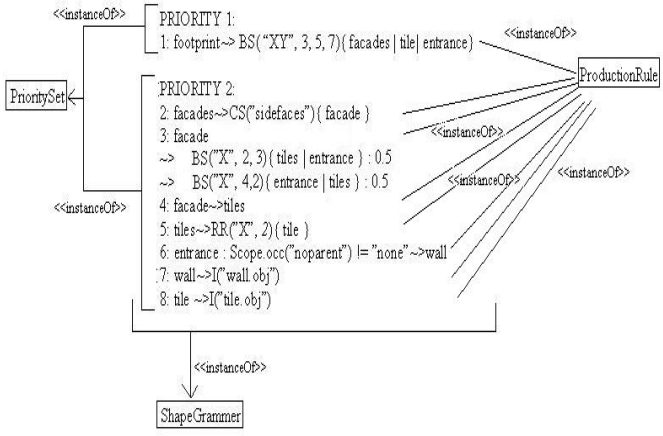


Figure 9. Example model 1-a

In Figure 10, production rule instances are shown. As an example, ProductionRule 3 has two RuleParts with probabilities 0.5. First RulePart defines the operation of splitting NonTerminal shape, facade, into tiles and entrance through X axis where tiles' length will be 2 and entrance's length will be 3.

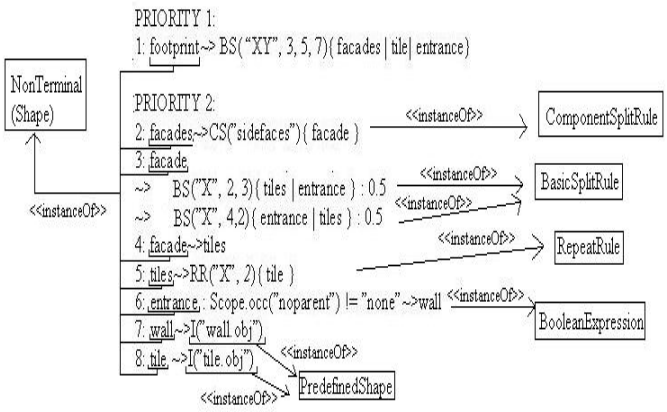


Figure 10. Example model 1-b

C. *Example Model 2*

This model is the representation of the first PrioritySet defined in the previous model example using UML profiling mechanism (Figure 11).

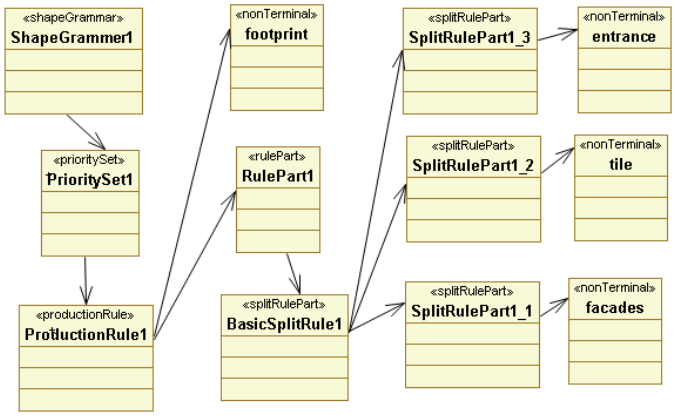


Figure 11. Example model 2 defined using UML profile.

The root is ShapeGrammar1 defined using the stereotype of ShapeGrammer. PrioritySet1 is using stereotype PritoritySet and belongs to ShapeGrammar1. PrioritySet1 has ProductionRule1 as ProductionRule. The footPrint element is the predecessor of ProductionRule1. BasicSplitRule1 defines the transformation function of footprint into entrance, tile and facdes that are all NonTerminals.

VII. Model transformations

A. *Motivation*

One of the claims of MDSD is supporting automatic code generation from models. In procedural building the output of the transformation of the model that is defined in domain specific language gives an output (XML for our case) that can be used by existent model generation tools. So portability of the model across different tools is provided by model transformation.

A model can be used to define different buildings since it involves probabilities of turning the shapes into different kind of shapes. This means by using this model different kind of buildings can be generated. Also the ease of updating the model saves most of the work which should be done by hand. By model transformation, the user can change the model in DSL much easier than doing the same job with manually modifying complex 3D models. This creates the opportunity of highly increased productivity.

Since the generated models are visualized, it can be used for architecture lessons. Students can play on the model and they can see the result on screen. After the model is transformed into XML file, a game programmer can take this model use it for creating building models for a game. Also the models can be changed and transformed automatically to have different kind of styles.

Another motivation for model transformation is to have better understandability of the models. Since our concrete syntax is textual, it saves great time and effort on making changes. However, a large shape grammar model in textual form can be difficult to design, understand and/or visualize in mind. This is due to the mandatory components that are included in the model such as conditions, different type of rule components etc. To design and understand the derivation that a shape grammar produces, on the other hand, a simple representation is required for visualization of shape grammars. A simple and explanatory derivation graph for a shape grammar can be used for this purpose. This derivation graph simply tells which shapes generate which shapes by which rules. So a transformation from a shape grammar to a corresponding graph model would be beneficial.

B. *Model-to-Model Transformation*

The meta-model for derivation graph is shown is Figure 12. Graph element denotes a derivation graph. Each node, as in the shape grammar meta-model, stands for a terminal node, empty node or a non-terminal node. Nodes are connected to each other with directed rule edges: A non-terminal node has a number of rule edges which denote the derivation rules that the node is the predecessor of. A rule edge denotes for a derivation rule which is a rule part in the shape grammar meta-model, of a type (component split, basic split, repeat, substitution) and have the nodes corresponds to the derived shapes as its target nodes. Rule type is assigned the type of the rule and ruleId is the id of the production rule of the corresponding shape grammar.

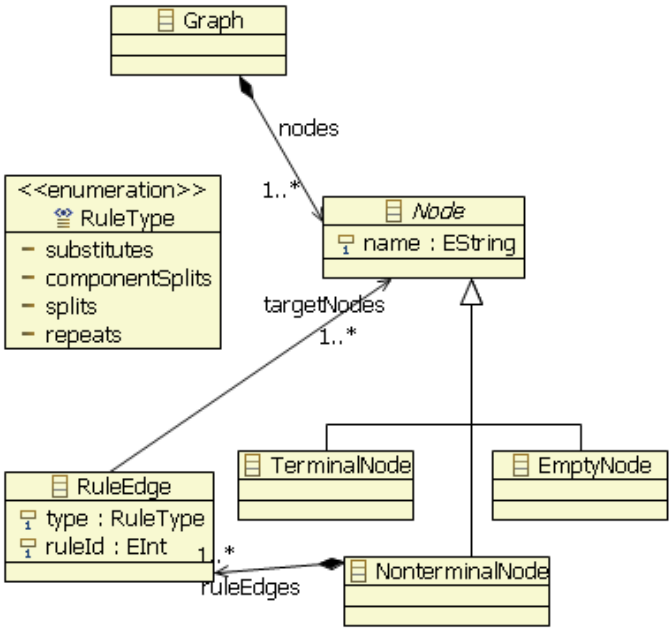


Figure 12. Derivation graph meta-model

The transformation is achieved using ATL (Atlas Transformation Language). The complete ATL code can be found in Appendix D. For the root of the shape grammar model, which is a ShapeGrammar element, transformation creates the root Graph element of the output model. This is perfomred by the rule named "root". During the transformation, every shape in the input shape grammar model is transformed into a node of the associated type in the output derivation graph model. These are performed by the rules named "non-terminal-nodes", "terminal-nodes" and "empty-nodes". Through the shape-node transformations, the root Graph element of the output model is associated with all the nodes that are generated, which is actually coded in the rule named "root". Then we need to generate RuleEdge elements which correspond to the RulePart elements of the input model; this transformation is performed by the rule named "rules". Note that, there are only the rule edges that correspond to derivation type of rules defined in the target meta-model; the scope rules are not cared since they are not derivation rules. The targetNodes attribute of RuleEdge elements are set with the shapes that the associated rule generates. The type of the RuleEdge is set according to the type of the associated rule. Through the RulePart-RuleEdge transformation, every non-terminal node is connected to the RuleEdges of which the non-terminal node is the predecessor, which is actually coded in the rule named "non-terminal-nodes". Also a helper rule named "getTargetNodes" is used to fetch the shapes that are generated for a rule.

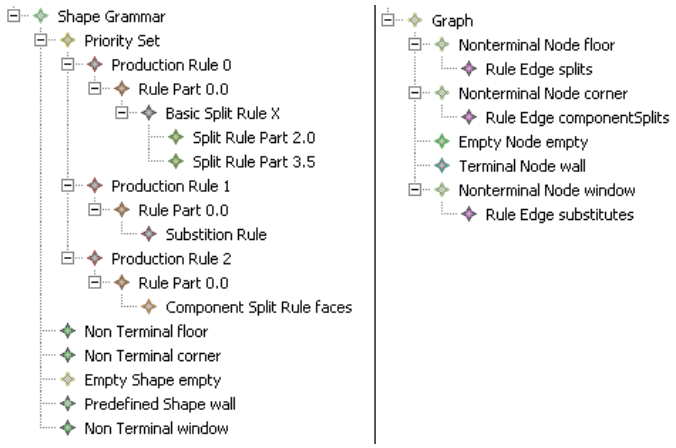


Figure 13. Input model (on the left) vs. output model (on the right).

To demonstrate the defined transformation, an example input model and the resulting output model is shown in Figure 13. The input model does not have any meaning and only supplied for demonstration purposes. As can be seen in the figure, the input shape grammar model include three non-terminal shapes named "floor", "corner" and "window", an empty shape and a predefined shape named "wall". For simplicity, there is only one priority set and three production rules associated within. Predecessor shapes of the production rules with id's 0, 1 and 2, are "floor", "window" and "corner" shapes respectively. By the production rule with id 0, "floor" is split into "window" and "wall" shapes. By the production rule with id 1, "window" is substituted with a "corner" shape. By the last production rule, a "corner" shape is split into its components; "wall"and empty shapes are generated.

When the transformation is applied, the output model is generated (Figure 13). The root Graph element contains all the nodes that correspond to the shapes defined in the input model. For every non-terminal node there are rule edges, in our case one per non-terminal node, which correspond to the rule parts of the given input model that have the non-terminal node as predecessor. Lastly, the rule edges are connected to the nodes that the corresponding rule parts generate. The resulting derivation graph is better represented by a graph diagram; after all the purpose of the defined model-to-model transformation is to promote understandability and the ease of design. A considerably larger input model would serve the demonstration of the promoted understandability better since the shape grammar model would be much more complex in that case. Besides promoting the understandability, we also stated that the design of a shape grammar may become easier using a derivation graph. However, to enable that, we also need a backward transformation from derivation graph to shape grammar. Once the draft design of the shape grammar is done with the help of a derivation graph, we can transform the resulting graph into an incomplete shape grammar model. Then the necessary additions and refinements can be performed on the shape grammar model.

C. Model-to-Text Transformation

In our case, we prefer to transform the model into a XML file which can be interpreted by another existing system. This system can transform the XML file into a graphical shape object which can be viewed by another 3D model viewer. The target XML file's structure is defined beforehand. The transformation is designed to conform for this existing system.

The target system does not support some of the transformation rules. ComponentSplitRule and ScopeRules are not supported by this system.

The transformation stage needs a source model which is defined by using the meta-model defined before. On the left side of Figure 14 overall format of the model is given. A shape grammar consists of a priority set (normally we have more than one but this system does not support assigning any priority of the rules) under this priority set there are rules shown on the right side of the Figure 14.

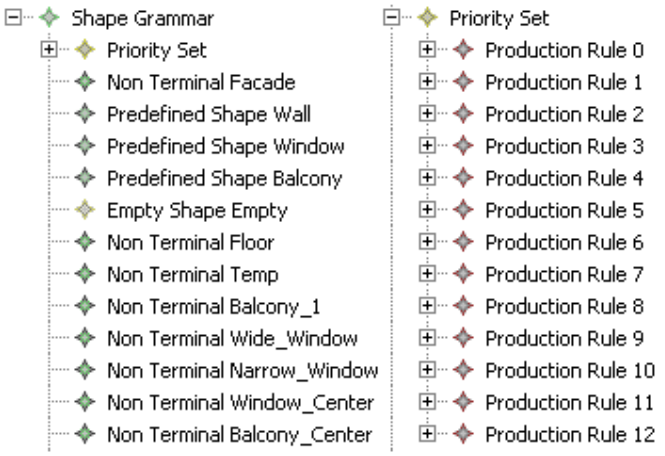


Figure 14. Example model for model-to-text transformation

After the model is initiated, the next step is writing the transformation rules. A `template` which can convert the current model into another is used. This template mechanism can be implemented by different languages; one of them is Xpand of openArchitectureWare. Xpand is used for transforming the model into the XML file.

```
context EmptyShape ERROR
  "Empty shapes can not have a scope!" :
   scope == null;

context ShapeGrammar::ShapeGrammar ERROR
  "The default Facade object is not defined!" :
   shapes.exists(e|e.symbol.compareTo("Facade")==0);


context ShapeGrammar::SuccessorRule ERROR
  "Scope rules are not acceptable for this context!" :
   metaType.name.compareTo("ShapeGrammar::ScopeRule")!=0;

context ShapeGrammar::ShapeGrammar ERROR
  "There can be only one priority set for this context!" :
   prioritySets.size==1;


context ShapeGrammar::PrioritySet ERROR
  "All rules must have a predecessor shape!" :
   rules.forAll(e|e.predecessor!=null);
```

Figure 15. Check language example.

There are two stages of the transformation; one of them controls the model according to the rules written for checking the integrity of the model. For this purpose `Check` language form openArchitectureWare is used. Before transforming the model to target file these rules are executed when there is a violation happens it stops the transformation. The Check language is similar to OCL. Some of the rules are given in Figure 15. These rules are especially written for the current context. As said before the existing system does not support some of the features defined in meta-model. These rules are for checking whether current model is suitable for transformation or not.

Xpand language brings lots of features with it. One of them is the aggregation relations are transformed into lists. Programmer can use this feature for easily traverse the model. Before describing the transformation rules an example target XML will be given for clarification of the transformation.

A part of the transformed XML file is in Figure 16. As it can be seen this format collects the rule under the shapes which are going to be transformed into another shape. For example as `BasicSplitRule` is defined by `<Fixed>` tag, we understand that `Balcony_1` will be transformed into wall,balcony_center and a wall again according to proportions for X axis given on the `<xProportions>` tag. The complete XML file can be found at Appendix E.

```
<Balcony_1>
  <R11>
    <Fixed>
      <xProportions xr1="1.0" xr2="5.0" xr3="1.0"/>
      <yProportions/>
      <Elements>
        <Wall/>
        <Balcony_Center/>
        <Wall/>
      </Elements>
    </Fixed>
  </R11>
</Balcony_1>
<Wide_Window>
  <R7>
    <Fixed>
      <xProportions xr1="1.0" xr2="4.0" xr3="1.0"/>
      <yProportions/>
      <Elements>
        <Wall/>
        <Window_Center/>
        <Wall/>
      </Elements>
    </Fixed>
  </R7>
</Wide_Window>
```

Figure 16. Example XML output of transformation.

Xpand language simply reads all the non terminal shapes under the shape grammar and collects the production rules which are transforming these shapes to other shapes. Then it processes each production rule and determines what kind of transformation rules are there and continues to complete inner part of the rules according to model definition. First part of Xpand code that does this operation is given in Figure 17 (Complete Xpand code can be found at Appendix F.).

```
1«IMPORT ShapeGrammar»
2
3«DEFINE main FOR ShapeGrammar»
4    «FILE "test.xml"»<?xml version="1.0" encoding="utf-8"?>
5    <BuildingGen>
6        <Rules>
7            «EXPAND nonterminals FOREACH shapes-»
8        </Rules>
9
10       <Terminals>
11           «EXPAND terminals FOREACH shapes-»
12       </Terminals>
13   </BuildingGen>
14   «ENDFILE»
15«ENDDEFINE»
```

Figure 17. First part of Xpand code for transformation.

When the engine is fed with the XML that is output of the transformation, the created 3D building model can be seen in Figure 18.

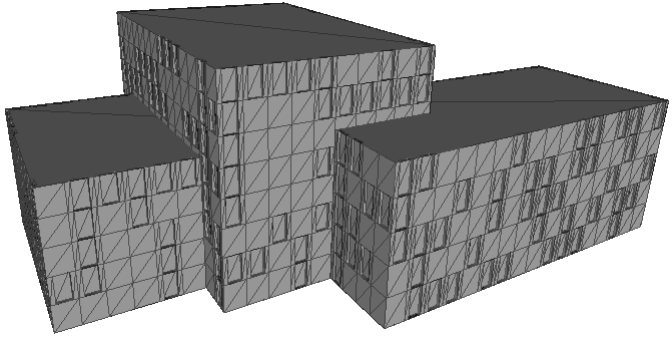
Figure 18. Created 3D model for the example source model

VIII. Related Work on Building Modeling

Previous work related to computationally generating building and city models demonstrates two major approaches. One approach is building model reconstruction using remote sensing and/or computer vision methods. The other approach is procedurally generating building models.

Reconstruction of city models can be performed by processing aerial images to extract the buildings and streets, using computer vision methods [1, 2, 3, and 4]. Another promising approach is to use range scanning with the help of laser airborne scanning and other remote sensing methods [5, 6]. Both of these approaches aim to get the models of the real buildings and real cities. There are quite successful results, however, there are also some problems related with these methods. One of the problems is that these methods are not fully automated. They cannot identify all of the geometric structures in a city because of the high geometrical variation of the buildings. Another problem is that city models with high level of geometric detail can only be constructed if, for every building in the city, specific data is acquired and processed. However, photographing or scanning every building in a city would be quite labor intensive.

Shape processing grammars, mainly L-systems, were applied to the modeling of streets [7]. Procedural modeling of buildings is inspired by the shape grammars [11]. The derivation of general detailed buildings using split grammars was demonstrated to be highly successful [8]. Split grammars are a composition of set grammars and shape grammars [11]. Split grammars split or transform 3D shapes to sub shapes that are included in the volume of the parent shape. Derivation ends when terminal shapes are derived which represents the building design. This derivation is steered by the attributes, so specific building designs and architecture trends could be achieved. During derivation, a parameter matching system is invoked that allows the user to specify multiple high-level design goals and controls randomness to guarantee a consistent output. An idea



of control grammars was introduced that are simple context free grammars which handle the spatial distribution of design ideas not randomly, but in an orderly way that corresponds to architectural principles. CGA Shape grammar is an improvement over split grammars [9]. CGA Shape grammar presents context free shape rules and can make use of complex mass models. Resulting buildings have underlying consistent mass models and high level of geometric detail. CGA Shape Grammar rules can be created from building images to generate a model of an existent building [10]. The meta-model defined in this work largely follows CGA Shape Grammar.

IX. Lessons Learned and Conclusions

In this work we developed the meta-model for procedural modeling of buildings using MOF for from scratch meta-modeling and UML 2.* profiling mechanism for the second way of doing meta-modeling and defining model transformations for the models that will originate from these meta-models. We faced some problems and came up with solutions to these problems not only in modeling and transformation, but also in the domain analysis part.

For procedural modeling domain, Scope was an extra-ordinary concept. It is a part of Shape as a domain dependent concept, but it is not represented in the Concrete Syntax separately since it is a run-time entity which becomes visible with the production rules rather than any other explicit notation. Numeric attributes of Shape creates a similar situation. In this case, we had to decide whether we should include Scope and numeric attribute definition as an class instance and class properties of EClass (of ECore) in the Abstract Syntax. Note that the meta-model is created for modeling purpose and automation of the code. Without defining the scope and numeric attributes in meta-model, the full automation seems not possible, because these entities are used and kept in run-time although they have no explicit declaration or notation. As a general guideline, the meta-model should be defined as coherently and explicitly as possible for full automation and validation.

In our case, it was easier doing the meta-modeling from the scratch compared to the UML profiling. The main reason was the tool availability and time for expertizing on tools that we used. For from scratch case, we used ECore meta-meta-model using Eclipse plug-in which is quite easy to learn and use for a person who is familiar with the modeling concept. For UML Profiling, Papyrus is used. Papyrus is a bit buggy compared to ECore. We expressed the profile in Papyrus, but Papyrus itself is not enough for defining the profile (to make it usable for creating models), so Topcased helped us with defining the profile. Another reason why UML profiling is

more challenging is that it requires enough knowledge on UML modeling subject.

There is no general standard between tools that leads to incompatibility of one model across different tools. Each tool has its own type of file extension, as a result different file types. Although the important point was the ECore and profile model itself, diagram files were not able to be opened. The tools are not only standardized but also not adequate for supporting full functionality for modeling purpose. For example, the plug-in we used for ECore was not supporting UML profiling, so we searched through different tools. and we did profiling using Papyrus that is a open source modeling tool.

As we mentioned previously, tools generally do not support full functionality for modeling. Despite of this fact, we can surely say optimized tools (tools or just a particular function) follow good standardization within itself and its meta meta-model (such as ECore). At first, we spent some time to get use to these tools, but after that we were impressed with the high-level support (such as model variation).

After we derived the concepts and relations between these concepts with the domain analysis, we started with from scratch part. Comparing the time we spend time on UML Profiling and MOF instantiated, MOF part took more time since we came to conclusions about critic and argumentative points. Following scratch meta-modeling, we created UML Profile for our project. Since the entities and relations in the domain are the same and clarified after domain analysis and discussions, for both cases, in the result, UML Profile was not so different from the MOF instantiated meta-model.

For our grammar, as it was described in the lessons also, we couldn't achieve to connect a shape to different rules. The case exists as the result of the tree structure that grammar is converted. A tree structure does not allow its leaves to be connected to the other branches of the tree, the graph structure has this power on the other hand.

As a last point to mention about the domain is Shape Grammar is also a grammar itself. Like any grammar, Shape Grammar also has conversion rules for converting or assigning values to different symbols, or entities. For the case at hand, we assign or covert non-terminal shapes to some other shapes in M1 level initiated from the meta-model in M2 level. This could lead us to misunderstanding and confusion, but we made the distinction between M1 and M2 levels very precise at the end.

For model transformation phase, the first problem that we faced was about the capabilities of target engine that is expected to convert XML code to visual 3D building models. Although our meta-model covers all the functionality expressed in the procedural building modeling, the tool that we used for visualization does not support all these functionalities. So a number of check rules is required to check the validity of the input model.

The tool, Xpand, which we used for model-to-text transformation lacks of documentation support. This case occurred especially for check language. We sometimes found ourselves in try and fail situation to solve some difficulties when using Xpand.

The code generated from transformation in Xpand is in an untidy form. To apply the correct indentation, it requires some hard work. So, a beautifier is required. After a while, the code written for transformation becomes very complicated. To solve this problem, the syntax of Xpand should be reorganized and updated.

The models are not much self explanatory for representing visually outside the tools. This is a common problem for both Papyrus and ATL. The associations between entities are defined as properties of objects in the model in textual form but they are not shown in the figures or list form of the models with connections between these entities.

For Model-to-model transformation case, we used ATL that is primarily a declarative language which is appropriate for the nature of transformations. Due to the endless possibility of variation of the source and target meta-models, transformations can be quite complicated. To be able to support all kinds of transformations, additional expressiveness is achieved imperative parts of the ATL language. However, imperative programming in ATL is strongly discouraged. This is because the virtual machine of ATL does not work on bindings in any specified order and so browsing the target model on the fly may produce unintended consequences. The transformations that are too complex to be easily implemented by a declarative manner could be achieved by a number of pipelined transformations instead of a single transformation. For our experience since we are familiar with imperative programming, at first, we miserably tried to implement a transformation that includes considerable amount of imperative parts. While generating non-terminal shapes, we tried to browse the rule edges that are generated which did not work at all. Then we noticed that we could achieve the aimed information by only browsing the input model and switched to the pure declarative approach.

Apart from the implementation of the model-to-model transformation, the transformation should be defined as precisely as possible. This would require one to know the source and target meta-models, and the semantics of the transformation very well. On most of the cases source and target meta-models are quite different, and, naturally, not all the information contained in the source model can be transformed into the target model. The transformation should only capture and transform all the information of the input model that is required to be expressed in the output model. One other point is that the transformation needs to satisfy the conformance the output model. For our case, since we defined

our target meta-model specifically to represent shape grammars and the source meta-model in mind, we have a quite well understanding of the target and source meta-models and the semantics of the transformation. So the transformation specification is defined very easily which supports the claimed facts.

As the final words, we did a well-performed domain analysis on Shape Grammars and depending on domain concepts and relations; we defined two complete and stable meta-models for this domain. These meta-models served as a reliable basis for the model transformations. For demonstrating model-to-model and model-to-text transformations on the defined meta-model, two model transformations are defined with the motivations of portability, increased productivity and understandability. The results shows that the model driven engineering promises a revolution in software area.

References

[1] F. Jung, B. Jedynak, and D. Geman. Recognizing buildings in aerial images. In Proceedings of the Workshop on Automatic Extraction of Man-Made Objects from Aerial and Space Images (Ascona'97), pages 173–182, 1997.

[2] Y. Liow and T. Pavlidis. Use of shadows for extracting buildings in aerial images. *Computer Vision, Graphics, and Image Processing (CVGIP)*, 49(2):242–277, Feb. 1990.

[3] L. Spreeuwers, K. Schutte, and Z. Houkes. A model driven approach to extract buildings from multi-view aerial imagery. In *Proceedings of the Workshop on Automatic Extraction of Man-Made Objects from Aerial and Space Images (Ascona'97)*, pages 109–118, 1997.

[4] V. Steinhage. On the integration of object modeling and image modeling in automated building extraction for aerial images. In *Proceedings of the Workshop on Automatic Extraction of Man-Made Objects from Aerial and Space Images (Ascona'97)*, pages 139–148, 1997.

[5] N. Haala and C. Brenner. Extraction of buildings and trees in urban environments. *ISPRS Journal of Photogrammetry & Remote Sensing*, 54(2-3):130–137, July 1999.

[6] H.-G. Maas and G. Vosselman. Two algorithms for extracting building models from raw laser altimetry data. *ISPRS Journal of Photogrammetry and Remote Sensing*, 54(2/3):153–163, July 1999.

[7] Y. I. Parish and P. Müller. Procedural modeling of cities. In *ACM Computer Graphics (Proceedings of SIGGRAPH '01)*, pages 301–308, 2001.

[8] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. *ACM Transactions on Graphics (Proceedings of SIGGRAPH '03)*, 22(3):669–677, 2003.

[9] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool. Procedural modeling of buildings. *ACM Transactions on Graphics (Proceedings of SIGGRAPH '06)*, 25(3):614–623, 2006.

[10] P. Müller, G. Zeng, P. Wonka, and L. Van Gool. Image-based procedural modeling of facades. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'07)*, 26(3), Article no. 85, 2007.

[11] Stiny, G. 1980. Introduction to shape and shape grammars. *Environment and Planning B 7*, 343–361.

Appendix A

Diagram of the meta-model realized using ECore.

Appendix B

Diagram of the meta-model realized UML profiling.

Appendix C

Textual representation of the UML profile.

Appendix D

ATL transformation code.

Appendix E

Complete XML output of model-to-text transformation

Appendix F

Complete Xpand code for model-to-text transformation