Comparing XText with ATOM3 for the traffic model case

Tom Pauwaert

tom.pauwaert@student.ua.ac.be

Abstract

This report is a comparisson of the ATOM3 modeling environment and the XText eclipse plugin for modeling of DSLs. The comparisson is made through the study of a single case, the traffic simulation problem. The solution within each framework is discussed based on three major categories, firstly the construction of the model, secondly the validation of the model and lastly the handling of the transformations. In conclusion the major difference between the two modeling approaches seems to be due to the distinction in visual vs textual approach.

Keywords: ATOM3, XText, Traffic Simulation, Comparisson, Case Study

1. Introduction

. In this report we will compare the textual DSL framework XText, using the Eclipse plugin, with the graphical DSL tool ATOM3. In section one the general differences will be highlighted in the way the two tools handle the construction of meta-models and models, the model checking capabilities and lastly how transformations or simulations are handled by the tool.

. After the general comparison the ATOM3 model will be highlighted and explained. This is followed by the presentation of the XText model and its explanation. In the fourth section the main differences between the models and the way they function will be presented. To conclude this report the final section will present the findings of this project.

2. Comparing XText and ATOM3

The comparison of the tools will be categorized in four sections mimicking the most important parts of the tools. Firstly we will look at the way the models and meta-models work in each of the tools, afterwards we look at how model checking is achieved. This is followed up by taking a deeper look at the handling of model transformations or simulations.

2.1. Model Construction

. As in most modeling tools models are based on a meta-model, which in turn is based on meta-meta-model, this is no different for ATOM3 and XText. The main difference between XText and ATOM3 here is that one defines the models on a textual basis and the other defines them graphically, respectively.

. In ATOM3 the meta-meta-model consists of classes and with attributes, generalizations and associations. The meta-model then defines which classes can exist within the model and how they can be connected together, through associations. Note that when defining associations within classes you can specify multiplicities which will already leave you of some model checks that should be done at a later stage. Of course since ATOM3 is a graphical tool we also wish to specify how each class should be represented visually, this is done by specifying the graphical property of each element in your meta-model.

. After you have fully constructed your meta-model you can start a new ATOM3 environment and load the formalism, that is, your own meta-model in ATOM3. This will provide you with an intuitive graphical interface for building your models.

. The XText plugin for Eclipse uses an entirely different approach, naturally. The meta-model is constructed by defining a meta-model as a language using an Ant-LR-like syntax. The meta-model then defines the syntax of the model. In this case when we write a model we write it in the language, using the syntax, specified in the meta-model.

. The plugin also creates a different modeling environment for your newly defined language. The meta-model is defined in a Java project; this project can be run as a new eclipse environment. A new model can then be created in the new eclipse environment by specifying using the associated file extension for the model file, as specified in the naming scheme of the meta-model. The editor then supports syntax highlighting and the model checking. Which brings us to the next point, how do models get validated?

2.2. Model Validation

. Both ATOM3 and have powerful model validation methods. In ATOM3 model validation can be done essentially at two different times. Firstly during the construction of the meta-model through the multiplicities defined on associations between elements. A second round of validation may be done during model construction with automatically triggering python code for example upon creation, deletion, or alteration of a model element. This python code may access the internal in-memory structure representing the elements that have been modeled.

. The ability to access the internal memory representation of the model is probably both a strong point and a weak point of ATOM3; it is a strength as it allows the user for very powerful model checks by analyzing all interconnections between objects in the model. The reason it could be perceived as being a weakness is because it somewhat breaks the abstraction of the model. More so it a shame that the method of defining graphical constraints is not elaborate enough for every of validation check one could require.

. On to XText; here we find that there model checking is also very powerful. Like ATOM3 there are also two main ways to check your models. The first type of checks are the instant checks which are triggered automatically every time you change the model. The other type are checks which are more expensive and which are only triggered upon saving of the model. Instant checks are defined in OCL, which makes them quite easy to write but not overly powerful. For the really complicated model validation operations you would need more than just syntax, you would require elaborate context.

. When writing expensive and complicated validation operations of the second type you may access the internal representation of the model, much like in ATOM3. These internal model elements are based upon the Eclipse Modeling Framework elements. The checks are written in Java and may use as many custom helper classes as need be.

. One very big advantage of the XText validation over the ATOM3 validation is that the XText validation may display errors at the exact place the error has occurred with very informative descriptions of the error. This is, to my knowledge, not possible with ATOM3.

2.3. Model Transformation

. This section actually presents the biggest difference between ATOM3 and XText. ATOM3 providing a visual modeling environment also allows the modeler to model the simulation of the model in a visual manner. In its entirety ATOM3 has a very nice way to simulate models.

. There's, once more, two ways for model simulation. Firstly you could code the transformation entirely using the inner representation of the model which, though powerful, seems to defeat the purpose of the graphic modeling a bit. The second and most intuitive way to perform the transformations would be to use the graph transformation engine that is present in ATOM3. By specifying a left hand graph pattern and a right hand resulting pattern you can easily define the transformation in your model. Each pattern matching set is called a rule, rules which can be given different priorities, so that rules with higher priority will always get picked first if the LHS pattern matches in the model graph. The graph transformation in its own right is a powerful mechanism, combining this with the python checks that can check whichever prerequisite that you can imagine makes it a truly brilliant way of defining graph (model) transformations.

. Then there is the actual simulation of the model. During simulation the model will be checked to see whether any rule can be applied, if no rules can be applied the simulation stops. When there are rules that could be triggered in the models current state there are a few options. Either ATOM3 randomly selects a rule to trigger, the user selects one of the available rules or all the triggerable rules are fired in parallel. This last method however requires the rules to be mutually exclusive.

. Model simulation in XText happens entirely differently. The lists of elements constructed in the model are parsed using a template file. This template file has a main template on the top level and can call subtemplates. The textual model is processed by the template and the output of the templates is written to one or more files. These files are the result of the transformation.

. The biggest conceptual difference between XText and ATOM3 here is that ATOM3 does in-model transformations. That is, the original model is altered. The XText transformation leaves the original model intact and generates an entirely different model. Note however, that this is also perfectly

possible with ATOM3. This implies that ATOM3 can have some sort of distinction between simulation and transformation; more precisely ATOM3 can also simulate instead of just transform whereas XText may only transform, plausibly into another model that may be used for simulation.

3. The ATOM3 Model

. In this section I'll explain a few things about the ATOM3 model we created. First I'll show the meta-model which in itself is quite self-explanatory. Then I'll explain how the model does its validation checks. To conclude this section the transformation will be presented with an example rule.

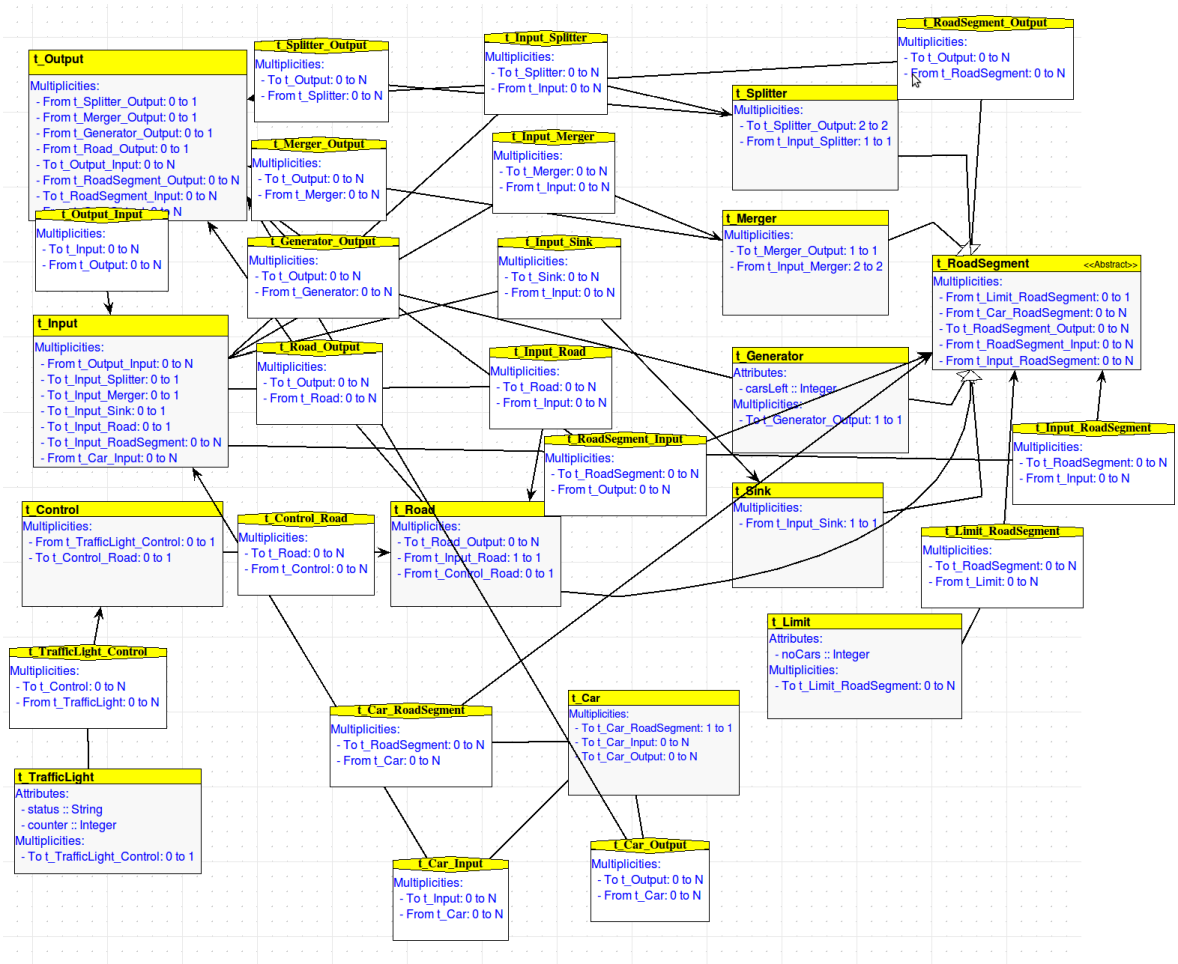

Figure 1: ATOM3 Meta-Model

. The meta-model itself defines elements for each road type, Road, Generator, Sink, Splitter and Merger which are each generalized from the RoadSegment. In addition there is the input/output elements and all associations with the road elements, there is a control element that connects to the traffic light and last but not least a limit element which can connect to as many RoadSegments as it likes. The car can connect to road segments, to input and to outputs.

. Now to ensure that the model is valid there have to be restrictions on what can be connected to what. Fortunately, a lot of this is very easily achieved in ATOM3. Our model already restricts which elements can be connected to each other and the multiplicities further constrict any possible problems related to how many elements of which can be connected to how many elements of another type. Another possibility for the creation of an invalid model would be to put to many cars on roads where the limit connected to those segments does not allow for such a number of cars. This kind of check is achieved by accessing the internal model structure using python code. The check is triggered upon connection of the cars. An example of such python code for checks follows below.

. Lastly, we present how the transformations work within the ATOM3 model. This is done graphically by specifying graph transformation rules. These rules have a priority, which enables the modeler to create multi-phase model transformations. Furthermore the priorities are used by ATOM3 to determine in which order the rule space is searched for rules that can be triggered from the current state of the model (see figure 2).

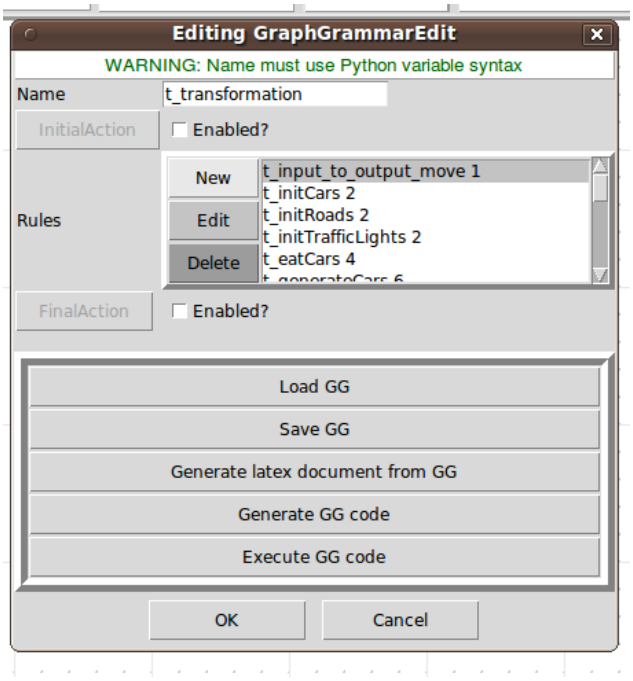

Figure 2: ATOM3 Transformation Rules

. As these rules are graph transformation rules they naturally also contain a left-hand-side (LHS) pattern which will be used for pattern matching against the current state of the model and a right-hand-side (RHS) which is what the LHS pattern will be converted into after transformation. These are the basics the graph transformation rules. However they are far more powerful, as graph transformation can sometimes be limited in its applicability due to lack of available context ATOM3 provides additional features.

. In the ATOM3 graph transformation rules it is possible to perform checks before the triggering of the rule. This will be used by ATOM3 together with the priority and LHS graph pattern to detect whether or not this rule is applicable in the current state of the model. These checks are written in python code and access the model through its representation in the internal memory. An example of a rule can be found in figure (see figure 3). An example of the python code used for checking rule applicability can be found below.

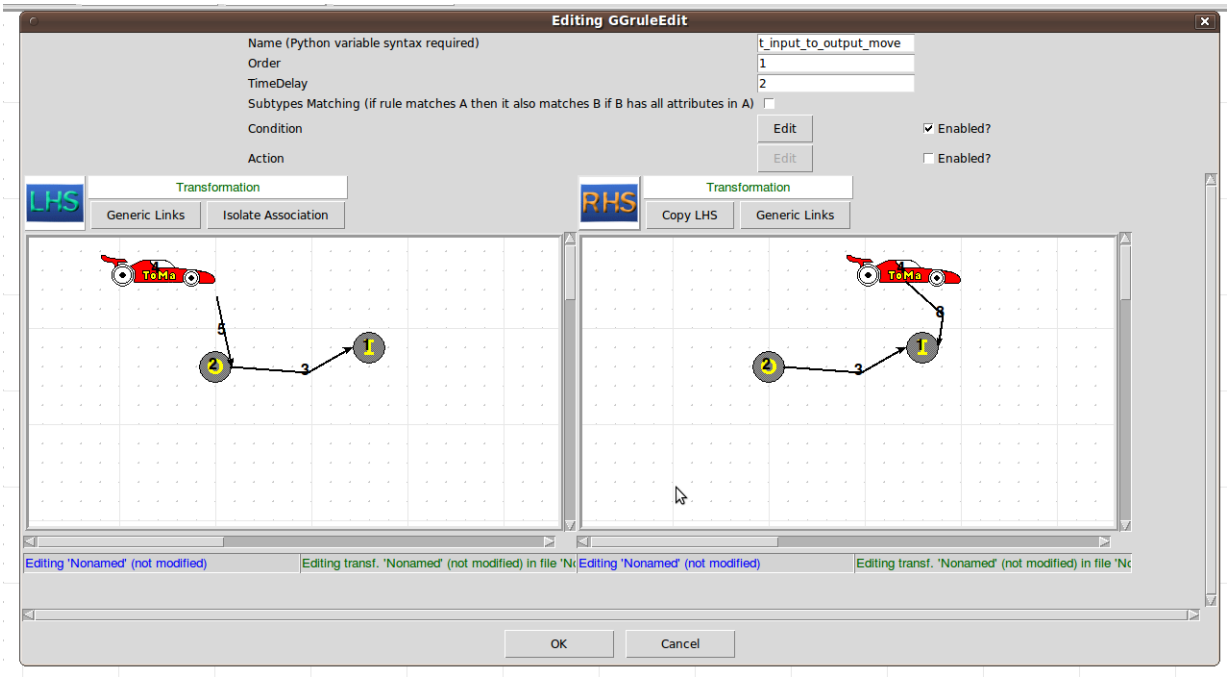

Figure 3: ATOM3 Input-to-Output Rule

```
...
...
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
car = self.getMatched(graphID, self.LHS.nodeWithLabel(4))
node = node.out_connections_[0].out_connections_[0]

limits = []
for potentialLimitArc in node.in_connections_:
    if (potentialLimitArc.__class__.__name__ == 't_Limit_RoadSegment'):
        limits.append(potentialLimitArc.in_connections_[0])

available = True
for limit in limits:
    available = available and not isLimitReached(limit,car)

if (available):
    print "Road segment is available - class name %s" % node.__class__.__name__
```

```
else:
    print "Road segment is unavailable - class name %s" % node.__class__.__name__

return available
```

Figure 4: Python code used in a rule applicability check.

4. The XText Model

. Constructing the model for the traffic case, or more specifically the DSL for the traffic problem, with XText is completely different than constructing it with ATOM3. The construction of the meta-model takes the form of defining a DSL language in an Ant-LR like syntax. Based on this DSL the XText plugin will generate an eclipse instance for you that provides editing support for your models in the DSL. The model in the editor will be validated, there will some syntax completion and last but not least it provides you with syntax highlighting.

```
grammar org.xtext.Traffic with org.eclipse.xtext.common.Terminals

generate traffic "http://www.xtext.org/Traffic"

Model:
'traffic_model' name=ID '{'
(elements+=RoadSegmentDecl
| generatorRates+=GeneratorRate
| links+=RoadLink
| cars+=CarDeclaration
| carPlacements+=CarPlacement
| limits += LimitDeclaration
| roadLimits += LimitOnRoad
| trafficLights += TrafficLight
| trafficLPlaces += TrafficLPlace
| SL_COMMENT
| ML_COMMENT
)+
'}';

TrafficLPlace:
```

```
light=[TrafficLight] '.' 'control' '(' road=[RoadSegmentDecl] ')';

TrafficLight:
name=ID ':' 'TrafficLight' (hasState?='(' state=INT ')')?;

LimitOnRoad:
limit=[LimitDeclaration] '.' 'limit' '(' road=[RoadSegmentDecl] ')';

LimitDeclaration:
name=ID ':' 'Limit' '(' limit=INT ')';

GeneratorRate:
generator=[RoadSegmentDecl] '.' 'rate' '(' rate=INT ')';

CarDeclaration:
name=ID ':' 'Car';

CarPlacement:
(car=[CarDeclaration|ID] '.' 'on' '(' onRoad=[RoadSegmentDecl] ')')
;

RoadSegmentDecl:
name=ID ':' type=RoadType ;

RoadType:
'Merger' |
'Splitter' |
'Generator' |
'Sink' |
'Road';

RoadLink:
(from=[RoadSegmentDecl] '.' 'out' '->' to=[RoadSegmentDecl] '.' 'in')
;
```

. When we take a look at the meta-model, or the grammar of the DSL more precisely, we can see a few different things. The top level element called *Model* is the entry point rule. It then specifies all the different subrules that

can be called which all define elements in the model, set certain attributes or link elements to each other etc. Note also that the grammar for XText allows the use of cross-referencing of instances. Lets take a look at the last line of the grammar:

```
RoadLink:
(from=[RoadSegmentDecl] '.' 'out' '->' to=[RoadSegmentDecl] '.' 'in')
```

Lets take a closer look at *from=[RoadSegmentDecl]*. This syntax adds an attribute to the ecore element associated with the RoadLink element. The ecore elements form the in-memory representation used by the XText plugin to represent your model. More concretely, the element RoadLink has a 'from' attribute which is cross-referenced, meaning that it links to another *actual instance* of another ecore element. In this case it links to a previously declared RoadSegment element. When we take a look at another line in the grammer, you find another peculiar syntax element.

```
| links+=RoadLink
```

The meaning of this line is actually quite intuitive, it means that it makes a list of all the *RoadLink* elements XText finds in the model and stores them in the links attribute of the Model ecore element.

. So far for the model, lets explain how we can validate the models content. As I already noted in previous sections there are two major ways of checking validity of a model, that is through OCL rules and through extensive Java-programmed rules that use the internal structure of the model to get as much context as possible. There is however still a third implict validation check. This is validation through cross-referencing. Using a cross-reference in the grammar means that the element will link to another *existing* element. This is means that it can not link to any element that does not exist. This has the following effect on our model: never can a person link to RoadSegments to one another if one them does not actually exist, more precisely, if it has not yet been defined. In the meta-model this sort of cross-reference existence check is used for all statements that are not declarations of elements.

. Besides the implicit cross-reference checks I utilized an OCL check for a simple integer between 0 and 6 constraint for the traffic light state and elaborate expensive model checks. The semantic meaning of the both the models (ATOM3 and in XText) are identical. Consequently the XText model has,

just like the ATOM3 model a limit constraint check. Upon linking a limit to roadsegements, or cars to roadsegments, if that would cause the limit to be passed the model editor will generate an error message. This specific validation check is written in Java against the ecore elements structure (the internal representation of the model in the XText plugin).

. Last but not least the transformation in XText which is entirely different from the transformation in ATOM3. The transformation here happens based on a template. The template is shown below:

```
IMPORT org::xtext::traffic
EXTENSION templates::Extensions

DEFINE main FOR Model
FILE name+"_simulator.java"
import model.*;

public class name_simulator {
public static void main(String[] args) {

// Create model elements & add to model if needed.
Model m = new Model("name");
EXPAND model_elements FOREACH elements-

// Set generator rates
EXPAND model_generators FOREACH generatorRates-

// Create cars in model.
EXPAND model_cars FOREACH cars-

// Link the model elements to one another.
EXPAND link_model FOREACH links-

// Place the cars on the model
EXPAND place_cars FOREACH carPlacements-

// Create the limits.
EXPAND model_limits FOREACH limits-
```

```
// Limit the roads
EXPAND limit_roads FOREACH roadLimits-

// Create traffic lights
EXPAND model_trafficlights FOREACH trafficLights-

// Link trafic lights to roads.
EXPAND place_trafficlights FOREACH trafficLPlaces-

// Simulate the model.
// Simulation ends when no more cars can be moved and/or generated.
while(m.hasMoreSteps()){
m.simulateStep();
}
}
}
ENDFILE
ENDDEFINE

DEFINE model_elements FOR RoadSegmentDecl-
IF type.compareTo("Sink") == 0-
Sink name = new Sink("name");
m.addSink(name);
ELSEIF type.compareTo("Generator") == 0-
Generator name = new Generator("name");
m.addGenerator(name);
ELSEIF type.compareTo("Merger") == 0-


Merger name = new Merger("name");
ELSEIF type.compareTo("Splitter") == 0-
Splitter name = new Splitter("name");
ELSEIF type.compareTo("Road") == 0-
Road name = new Road("name");
ENDIF-
ENDDEFINE
```

. The template is quite straightforward. For the Model element, which is the main element in the grammar and represented by an associated ecore element, it runs the main template function. This function creates a file and writes content to the file. This process is the transformation of the written text representing the model in the DSL to a likewise representation of the model in java code. This implementation of the transformation also utilises prewritten java files that have classes in it, such as the 'Model' class, Merger class, etc.

. Note once again that the major difference between XText and ATOM3 is that ATOM3 does inplace transformations on the current state of the model while XText generates an entirely new entity that is the transformed version of the original model. In that sense the ATOM3 graph transformations could also be interpreted as 'simulations' while the XText transformation is a simple 1 to 1 transformation of original language to target language. In this particular case however I have created the model output file so such that it becomes an executable version of the model such that it can be simulated to validate the generated simulation output or just the simulation behaviour.

5. Comparing the two techniques

. When we compare the two modeling approaches, either by using ATOM3 to model visually or XText to model textually, the differences between the capabilities of the two approaches are minimal. So while they are equally powerful, the main difference between ATOM3 and XText is one that does cause them to excel on different fields. After getting hands on experience with both modelling tools I would dare state that ATOM3, due to its graphical nature and its 'simulation'-like transformations, is more suited for simulations of visual models. This would include models of petri-nets, statecharts, electrical components, etc. XText on the other hands due to its textual nature and inherent 'text-transformations' would be a lot more suitable for modeling domain specific languages that are naturally represented using text. Furthermore XText brightly shines in its ability to convert models to other textual formats. When you wish to transform your model to other textual languages, XText is definitely the way to go.

. Concluding this section we find that both tools excel in their respective approaches towards modelling. ATOM3 excels all things that require visual

models, XText excels in all things that are textual by nature. Both approaches are equally powerful and choosing one over the other should only be based on the visual vs textual needs of your model. Since both tools are quite easy to use, though XText does seem to have a slightly higher learning curve, textual vs visual should be the deciding factor in tool choice. Except when one has strong personal preference as really both tools are equally powerful.

. Future work in this section could include further comparissons of other tools for modelling or more elaborate problems to be modeled. One situation that seems like it would give rise to interesting results would be to either use visual input to provide textual output or the other way around. In this case the difference of the tools will be better highlighted and more interesting data would be found. In this situation the visual vs textual difference mostly overrules all other differences.

References

[ATOM3] *http://atom3.cs.mcgill.ca/*

[XText] *http://wiki.eclipse.org/Xtext*