

Modelling with IdeoDSL

Project Report

ECSE 539 - Software Language Engineering
(Fall 2015)

Group 4

Rida Abou-Haidar, 260426399

Yanis Hattab, 260535922

Saige McVea, 260466561

1. Introduction

The following is a report describing “IdeoDSL”, our term project for ECSE 539 - Software Language Engineering. Because we were asked to *design and implement* a domain-specific modelling language (DSL), most sections of this report can be divided into two distinct parts: what we hoped to develop, and what was accomplished—henceforth denoted as our “intention” and our “implementation” respectively. We begin by describing our selected modelling notation at a high-level, including a realistic use-case to help readers understand how the DSL can be applied. Then, we explain how to install and run the source code of the project. The next section details both the abstract syntax with the help of an Ecore metamodel, and the concrete syntax alongside a number of diagrams and examples. Finally, we discuss how models created with our DSL can be analyzed, as well as the problems we faced during development. The report concludes with a brief evaluation of the tools employed and possible future directions for the project.

2. Description of the Notation

Intention (Proposal submitted October 18th)

Motivated by the 42nd Canadian general election, we initially set out to develop a vertical domain-specific modelling language for electoral systems (to be called “DemocDSL”, as the source files of our Eclipse project are named). Our goal was to allow *Constituents* and *Parties* to be modelled as the primary objects, but more importantly, we hoped to facilitate the depiction of important relationships such as *Influence* and *Agreement*. Central to this idea was (de)composition; constituents would be self-composed of other constituents until the *atomic constituent* object could no longer be divided. Similarly, relationships could be broken down into their atomic features, each with a distinct weight, such that atomic relationships could be summed to calculate the value of a parent relationship.

For example, in a Canadian federal election, the set of all Canadian citizens make up the atomic constituents of an instance of our model. These can then be grouped according to electoral districts, provinces, households, or even age-groups and other demographic-based criteria. Agreement can exist between every Constituent, every Party, as well as between Constituents and Parties. Agreement for this use-case might be founded on factors such as environmental concern, economic socialism, criminal justice, etc. Influence can similarly exist between all Constituents and Parties, and might contain atomic features such as ‘eloquence of party leader’, ‘strength of friendship’, or ‘followers on Twitter’. The model could be analyzed in a number of ways. The first is to assume that all constituents act according to their strongest agreement, in which case simply calculating the party for which the most atomic constituents feel the strongest agreement in any particular group would determine who won (a third relationship class could therefore be “Selection”). However, a more realistic assumption would be that if the person is impressionable enough, their selection might change according to the atomic constituents influencing them. In this case, you can assign the internally held portion of agreement to be x , the externally-determined portion of agreement to be y , and the sum $x + y$ to be the *actionable* agreement that would determine an atomic constituent’s selection in the same manner as that described above. We believed this notation was not only theoretically compelling for its methodical representation of “group-think” and bias, but would be truly useful if well-executed.

Implementation of IdeoDSL

Unfortunately, implementing a DSL with the above representation proved to be infeasible. Because of difficulties in attempting to differentiate between constituents and parties, which is obviously important for model analysis (party

members have a selection too), and problems caused by trying to represent multiple-composition in Ecore with suitable generated code, we were forced to make certain adjustments (see section [6. Problems encountered](#)). In the current implementation, all Constituents are atomic objects and parties are not represented. This simplified the abstraction of Influence, because it enforced a realistic one-to-one weighted relationship between an *influencer* and an *influencee*. Influence cannot be broken down into atomic features, it is a relationship object with only weight as an attribute. Similarly, we could not conquer the composition of agreement, so instead, we modeled Beliefs as primary objects. A third element of the model is an Ideology, representing a composition of many beliefs. Beliefs belong to either a constituent *or* an ideology, and they have an associated value describing how strongly the object feels that belief is significant. In more formal terms, we have actually developed a *horizontal* DSL describing how network effects of influence between constituents can alter decision-making normally founded on personal values. Our modelling language was thus appropriately renamed to “IdeoDSL”.

To suggest alternate applications from the one described earlier, IdeoDSL could be used to make decisions regarding whether a small business should provide its employees with health insurance, or buy a new espresso machine for the main office based on the individual beliefs of all stakeholders (which could include only executives, or all employees affected by the choice). This example might seem trivial, but it demonstrates the flexibility that now exists in the notation by simply omitting the Party class. The language is *declarative* because it describes mainly the prevailing ideologies and beliefs of a population, rather than how these beliefs should be analyzed, or any executable algorithm describing the control-flow that needs to be followed. IdeoDSL is also *graphical* (see the section [4. Concepts - Concrete syntax](#) below) because it uses a visual model, and *external*, as we defined our own syntax. Finally, a model created using this notation is *interpretative* only, as applying a full model-to-text transformation to generate an executable application was simply not possible. Although IdeoDSL is substantially different from the modelling language we initially proposed, we believe we sacrificed conceptual elegance for practicality and usability.

3. Usage

Running the source

After importing the Eclipse project with a Modelling version of Eclipse, one can start the model editor by running *DemocDSL.java*, found in the *ca.mcgill.emf.democdsl.application* package. This will initialize a model instance and start the editing GUI. The GUI will first ask the user to load a saved model, and if no file is selected, a new filename is provided. Pressing *Cancel* will open a blank model. In the case where a model already exists and is selected, the application will try to load both the model and its layout into the model view (the layout filename that the application will load is the model filename appended with *.csv*) and start the GUI. A failure to load the model (e.g. an invalid xml file) will lead to a blank model while a failure to load the layout will just stack the elements in the upper left corner of the model view. Saving a model from the GUI is done by clicking the “Save Model” button, which will open a dialog to choose where to save the model—the application will not automatically perform this action when a model is closed. Models should be saved as *xml* files and the extension should be written **with** the filename while saving. If the user doesn’t specify a name or clicks *Cancel*, the model will be saved as *model.xml* (with its layout as *model.xml.csv*).

Creating a model

When the source code is correctly installed and *DemocDSL.java* is running, two windows are displayed to the modeler. The first window, the Model View, allows the user to keep track of the model’s current elements. The Model View window allows the user to move around Ideology, Belief, and Constituent objects, thus helping to keep

the model representation organized as well as allowing for customization. The second window displayed is the Editor View. The Editor view acts as the main GUI and allows the creation of any modeling element such as Ideology, Belief, Constituent, Influence between Constituents, and the weighted adherence of an element to some Belief.

A typical model creation procedure begins by adding the desired Constituents one-by-one while giving each a name and independence. The same can be done with Ideologies which only require a name. To associate a Belief to a Constituent or an Ideology, the Belief must first be added to the set of beliefs available in model. To accomplish this, use the “Create Belief” portion of the GUI to create a new Belief by providing a unique name. Note that when a belief is created, the Model View window is not modified. It will only be modified if the belief is linked to a suitable element within the model. Finally, adding an Influence between two constituents can be done by determining the “influencer” and the “influenced”, then providing a weight to their influential power. Because two Constituents can have mutual influence between one another, each influence is represented in the Model View as a distinct arrow.

When the modeler creates objects from the Editor, an Ideology is represented in the modelview as a rounded rectangle, Constituents are represented as rectangles, and Beliefs are shown as ellipses. More details on the concrete syntax of IdeoDSL are provided in the following section ([4. Concepts - Concrete syntax](#)).

Our implementation provides a sample analysis to demonstrate the flexibility of the tool. The class (*VoteAnalysis.java* found in the *ca.mcgill.emf.democdsl.analysis* package) can compute for what Ideology the Constituents would theoretically vote in a model, that model must be passed to its constructor. A constituent can only vote for one Ideology and the ability to vote for an ideology only exists when they share all beliefs. The weighted sum of belief values is then computed to determine the Ideology for which a Constituent would vote. A summary of total votes is returned as a HashMap relating ideologies and supporting votes, which is then displayed as a list in a new window in the GUI. To start the analysis of a created model, click the “View Results” button. The model view will color the ideology with the most votes green. For the sake of simplicity, neither influence nor the Ideology’s value for each belief is taken into account in the analysis at this time.

Finally, the Editor reports with an error message when an Influence or a Belief is not added to the model upon request from the modeler. The error can be caused by a number of reasons, and the particular reason that applies is revealed at the top of the GUI. In the case of an error, the model state is not changed, so the modeler has the opportunity to make a correction then try to add the element again if they so choose.

4. Concepts

Abstract syntax (implementation)

Domain structure

The abstract syntax of IdeoDSL is fairly straightforward. As mentioned previously, the project was initially termed DemocDSL, and as such, the base class in our metamodel still carries this title. Other elements of the metamodel include Constituents, Ideologies, Influences, and Beliefs. DemocDSL objects have a single property, *name*, which could be used to describe the domain being modelled (e.g. “Banking”, “Government”, or “Healthcare”). Ideologies, Constituents, and Beliefs also have a ‘name’ attribute. Constituents have an integer denoting their independence, and Beliefs have an integer for their value. The Influence class only has ‘weight’ as an attribute. It is important to note that there exists bidirectional cyclic references between Constituents and Influences. This means that Constituents can have many outgoing Influences (“influencesOut”), and they can have many incoming Influences

(“influencesIn”). The Influence object itself represents a relationship, so it can only reference exactly one “influencer” and *be referenced* by one “influenced”. Please refer to figure 1 on the next page for a visual representation of our metamodel.

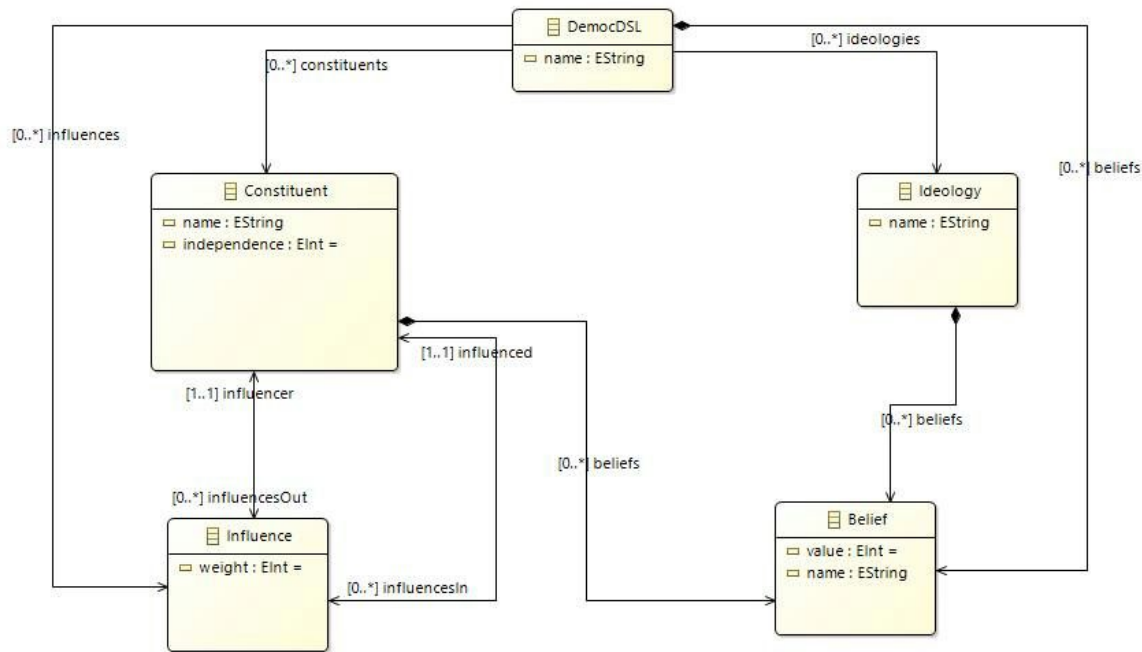


Figure 1 - Class diagram of IdeoDSL metamodel

As can be observed by examining the diagram above, three Ecore classes compose Beliefs, those being the DemocDSL parent class, Constituents, and Ideologies. We will admit that this is a workaround to manipulate the generated code: when a model is created, the user first enters the name of a Belief to create it, then does so for each of the needed Beliefs that apply to the domain being modeled. These beliefs do not yet have values and are stored in *DemocDSL*. Only after having named the belief can one assign this Belief (with a value) to an Ideology or a Constituent. We then create another instance of the belief with the same name as the one in *DemocDSL* and add it to its proprietary. Therefore, the composition of Beliefs in DemocDSL simply serves as a container for unassigned beliefs. The composition of Beliefs in Constituents **or** Ideologies ensures mutual exclusiveness. In a way, this workaround helps to facilitate well-formedness. Ideologies, Constituents, initial Beliefs, and Influences are referenced by the base DemocDSL class, which allows us to persist the models with less effort.

Well-formedness




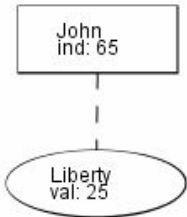
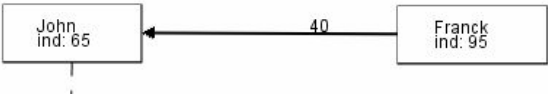
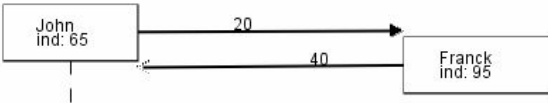
At this time, we have not implemented any well-formedness rules to constraint the set of valid models defined in IdeoDSL using the Object Constraint Language (OCL). However, there are a number of constraints that we would define if implementing them in OCL on our Ecore metamodel were more practical. These include:

- a Constituent’s independence must be [0, 100]
- an Influence cannot reference and *be referenced* by the same Constituent
- an Influence’s weight must be [0, 100]
- a Belief’s value must be [0, 100]

Figure 2 - OCL constraints that could be used to ensure well-formedness

Concrete syntax (implementation)

When a model is created by running our Eclipse project, instances of the model have the following representation:

<p>Constituent:</p> 	<p>Represented by a rectangle of size proportional to its name length. Name and independence are displayed as text inside the rectangle.</p>
<p>Ideology: Default look</p>  <p>Ideology: Highlighted as result of analysis</p> 	<p>Represented as a rectangle with rounded corners of size proportional to its name length. Has a name only, displayed as text. If an analysis is run where one ideology gets highest score, it is possible to set the color of the ideology to green.</p>
<p>Belief:</p> 	<p>Represented as an Ellipse with the link to its proprietary as a dashed line. Only appears if linked to a Constituent or an Ideology in the model. Its name and value are displayed as text inside the ellipse.</p>
<p>Influence: one-way</p>  <p>Influence: two-way</p> 	<p>Represented as a bold line ended by a directed arrow between the Constituent element that is the <i>source</i> of the influence to the Constituent element that is the <i>target</i> of the influence. The weight of the influence is displayed as text near the center of the arrow. The arrow ends at the closest corner of the target rectangle to the source rectangle center. Bidirectional influences between two constituents are represented using two arrows.</p>

It should be taken into account that **no third-party tools** were used to generate these figures. The GUI *and* the model are coded from scratch. We therefore regard this concrete syntax as a major success, considering our

experience with computer graphics and the time constraints on the project itself. Please refer to the section below for an ideal (not implemented) representation of IdeoDSL models, as well as section 6 for problems encountered in attempting to create a Sirius *Viewpoint Specification* project for our metamodel.

Concrete syntax (intention)

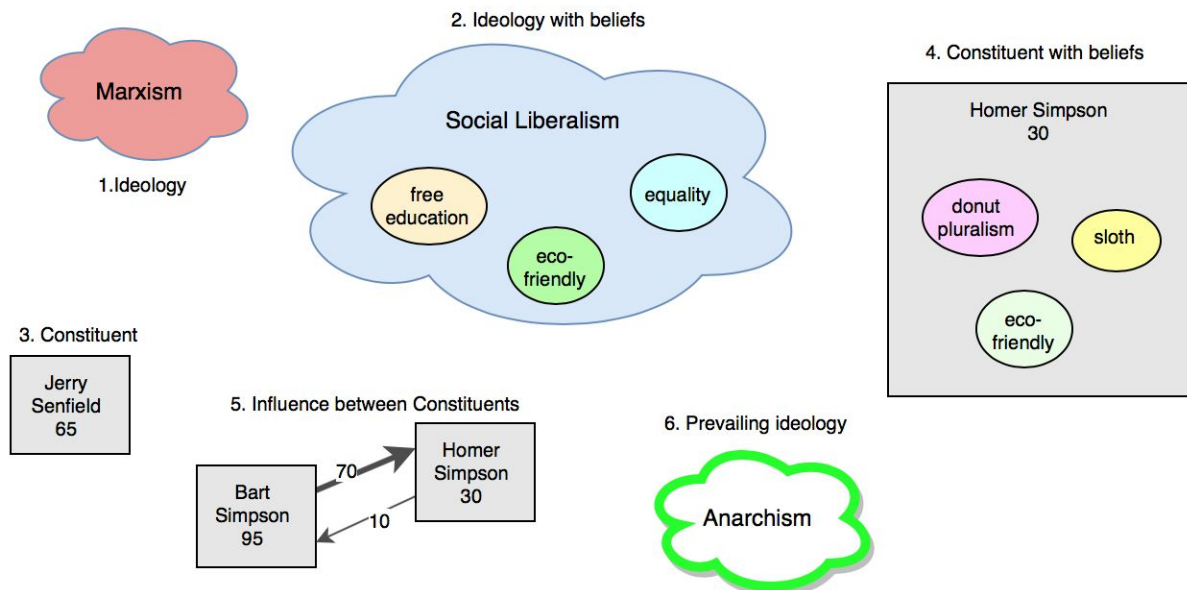


Figure 3 - ideal representation of IdeoDSL modelling features

Assuming our metamodel goes unchanged, figure 3 above depicts an ideal representation of IdeoDSL modelling features. It takes advantage of various design elements such as shape, color (hue and saturation), and space (shadows) to improve understanding and model usability. This graphical notation is better-suited to our chosen DSL because it now depicts a number of Abstract syntax features using visual language.

For example, when the modeller is viewing an IdeoDSL model at a high-level, they would only see ideologies and constituents as in (1.) and (3.) respectively. If they were to “zoom in”, they would then be able to see the beliefs *composing* that element, as in (2.) and (4.). Moreover, instead of numerical values being associated with these beliefs, saturation is used instead to describe the element’s associated value for any particular belief: examining (2.) and (4.), we see that Homer Simpson does not think that eco-friendliness is very important. Finally, distinct shapes describe the classification of the element. Constituents are rectangles, Beliefs are ellipses, and Ideologies are clouds. Influences are denoted by arrows, which have dual textual and visual encoding of weight through the application of progressive line thickness and labelling. Upon model analysis, the color and shadow of the prevailing Ideology would resemble that which is depicted in (6.).

5. Analysis

Although our metamodel is simple and contains only a few numerical values, the flexibility of the metamodel presents a wide range of analyses that could be performed using the information contained in model instances. The

parts of the metamodel supporting model analysis include Influence weight, Belief value, and Constituent independence. A non-exhaustive description of these possibilities are described hereafter.

i. Evaluation of prevailing ideologies

We hypothesize that this analysis would be the most common use-case, and it consists of the two important variations mentioned in section 2. Description of the notation - Intention. As described earlier, a simple calculation for each Constituent would be:

$$\forall c \in C : \min(g \in G \left[\sum_{b_i \in B} |\Delta b_i| \right])$$

where b_i is the value of a Belief held by both a Constituent c and an Ideology g , B is the set of all beliefs, and Δb_i is the difference between a Constituent's value of that belief and the Ideology's value of a belief with the same name. The differences between all beliefs is summed, and a Constituent's selection is the Ideology contained in the set of Ideologies for which the aggregate difference is a minimum (read: the Constituent has minimal disagreement with that Ideology). When this operation is performed over the set of all Constituents, the Ideology with the greatest number of supporting Constituents is the prevailing Ideology of the population.

A more realistic approach is to perform the same calculation, but factor in a particular Constituent's likelihood to be influenced by other Constituents. The formula therefore becomes:

$$\forall c \in C : \min(g \in G \left[\sum_{b_i \in B} |\Delta(\alpha b_i + (1-\alpha)\beta_{b_i})| \right])$$

$$\text{where: } \beta_{b_i} = \left[\sum_{n \in N} w_n b_{in} \right] \div (N \times W) \quad \text{and,}$$

$$W = \sum w$$

To clarify, we now consider the Constituent's independence α (mnemonically “autonomy”) and $(1-\alpha)$ the portion of their support which is determined by their influencers. The value β_{b_i} describes the weighted average of their influencers' value for any particular belief. This weight is described above as w_n , and the sum of all weights as W . Again, the Ideology with the greatest number of supporting Constituents is the prevailing Ideology of the population. Please note however, that inherent Belief values held by Constituents are **immutable**. This prevents the much more complicated calculation where the effect of aggregated Influence cascades throughout the model.

ii. Determination of key influencers

Another possible analysis that could be performed as a secondary step to the one discussed above is the determination of key influencers within a population. To put the objective more simply, an IdeoDSL model could be used to strategically target Constituents with the greatest ability to alter the *selection* of other Constituents. Although it goes beyond the scope of this report to define a mathematical formula serving such a purpose, there exists a number of linear regression and search models that could be used to accomplish this task.

iii. Estimation of optimal grouping

Similar to the application described above, IdeoDSL models could be used to undermine normal democratic systems. By grouping constituents such that support for a particular Ideology was contained in only a few

subpopulations, the outcome of the calculation of “*prevailing support*” for an Ideology could be manipulated. Examine the base case: Constituents (a, b, c, d) can choose between Ideology X or Y. From their beliefs, a and b choose X whereas c and d choose Y. However, a and b are grouped in the selection process; c and d each have their own group. The outcome is therefore 1 choice of X and 2 choices of Y. Consequently Y is the “winner” of the selection process, as it is falsely deemed the prevailing Ideology of the population. In reality, a more intelligent version of this technique could be used to “rig” elections by changing the physical boundaries of electoral districts.

iv. Assessment of target constituents

In a more pragmatic application of possible analysis, models created with IdeoDSL could be used for advertising and marketing. Although objects mapping to these entities are not represented in the metamodel, they would simply be described as an Ecore class with Beliefs and *influencesOut* on Constituents (the *influenced*, see figure 2), but no *influencesIn*. Judicious placement of instances of this class within a model could be used to sway constituents using the same techniques as those described in analysis (ii).

6. Problems encountered

As a group, we encountered a number of obstacles at different stages of the development process. First and foremost were problems encountered using EMF itself. Probably a result of our limited knowledge of its features and their appropriate applications, we only utilized EClasses, EReferences, EComposition, and EAttributes (EString and EInt) in our metamodel. We struggled in our attempts to model abstract classes and in our efforts to use multiple-composition for the Beliefs object. Because we were most likely trying to force EMF into accomplishing a task which conceptually should never be permitted (or that we simply lack the experience to correctly implement), we made compromises to achieve the functionality we desired. One example is that which was referenced earlier, relating to the composition of Beliefs in Constituents, Ideologies, and DemocDSL. Composition of Beliefs in Constituents and Ideologies ensures that Beliefs belong to only one distinct class, however composition in DemocDSL is not per se necessary. This was done because it made serialization of the model easier but also serves as a container for beliefs without a *value*. Finally, multiple instances of Beliefs are created when it may have been more conceptually elegant to only create Beliefs having unique names once, and use another EClass object to model Constituents’ values of that particular belief. This complicated the metamodel more than we believed necessary, as we wanted to maintain a close coupling between the metamodel and our initial abstract syntax, so we avoided that solution. Finally, this choice also facilitates belief comparison between constituents and ideologies: we only have to check for name equality on the belief objects.

Another difficulty we encountered relates to model execution and constraints ensuring the well-formedness of models. Most of these features are implemented in the controller and the GUI of our project, instead of at the metamodel level. Although we certainly could have used OCL to describe many of the constraints described above, it seemed unrealistic to have our implementation interact with another Eclipse modelling library, in addition to the potential conflicts and bugs it might cause. For now, validity of models is guaranteed by simply disallowing the modeller to create a syntactically incorrect model. The nature of our GUI prohibits most invalid operations as it is based on lists that are refreshed continuously, so the modeller can only select valid elements (see figure 4). That being said, our GUI is also restricted in the sense that it does not allow for the deletion or editing of model elements. At least for the purpose of this project, we assume that the modeler has complete knowledge of the system he wants to model, and therefore has no need to edit an existing element (or even less to remove one). In practice, implementing the deletion and editing tools would require additional code and further checks for well-formedness. This obviously presents the opportunity to introduce errors and potentially break the functionality of the current implementation. In order to maintain focus on the actual purpose of the project, we agreed to sacrifice this features.

The metamodel and the generated code, however, could support this functionality, and only the controller, editor, and Model View would require additional modifications.

A third problem we encountered relates to the GUI. Substantial efforts were put into the current implementation, as everything was done manually using Java2D primitives. We made many compromises in order to have a viewable model, and further effort was put into displaying a readable representation to the modeller. For example, the calculation of the correct placement of arrows was coded by hand to ensure maximal readability of the model, and the interface the modeller uses to enter object attributes uses the Java Swing toolkit. In addition to the time devoted to create the present version, considerable time was also devoted to an attempt to use the [Sirius](#) Eclipse project. Sirius however, is still rather new, with an official release only available since 2013, and the documentation is still rigid in the sense that it provides minimal support for beginners. After wasting much time attempting to import a custom metamodel into a *Viewpoint Specification Project*, it was discovered that only the autocomplete feature of Sirius was not working properly, and time constraints on the project would not allow further effort to be allocated to what still could be unsuccessful.

In terms of what went well in our project, we are actually quite satisfied with what we were able to achieve. The IdeoDSL modelling notation has a strong theoretical foundation and was therefore intellectually compelling to discuss. The horizontal nature of the language allowed us to easily think of many concrete applications, which helped with our abstract understanding of scenarios that could arise as a result of the metamodel definition. We believe that it also has high potential for practical analyses, and if adequate development efforts were invested (say, twice the team and many more weeks), it might even get some real use. Despite its apparent simplicity, we also hold high esteem for the GUI implementation. It lends the ability to move elements around, with arrows adjusting automatically so the user can produce a model that can be saved as a picture then discussed in presentations.

7. Conclusion

While developing the IdeoDSL modelling notation, we were forced to acknowledge the true complexity that exists in developing tools used to support model-driven engineering (MDE). Although we, as a group, criticized jUCMNav, Umple, EMF, and even Sirius at various points in this course, creating our own domain-specific modelling language made us realize that these tooling projects require substantial effort to achieve. The use of code-generation and third-party plugins is certainly helpful, so we are tempted to believe that *tooling will become easier as tooling improves*, however, maintaining, expanding, and improving the foundational projects still requires a thorough understanding of (meta)modeling.

The results, when successful, are very rewarding and we recognize the potential for a well-executed DSL to substantially reduce development time in large projects. If we were to continue our work on IdeoDSL, we would first ensure that we were using the full power of EMF in a way that was correct for our unique requirements. Then, once the metamodel was truly secure, we could improve GUI such that manual interactions and acceptance testing was easier, and so that working with IdeoDSL was more enjoyable. Further improvement would be to provide an analysis framework where one can code his own analysis strategy and plug it seamlessly into our implementation to suit his needs. We believe that the more general and customizable the tool, the more broadband acceptance it will reach. Expanding the test suite would then also be necessary, and from here, working on implementing realistic model analyses options based on actual human sciences studies would be the most useful. Although we are perhaps not the best-suited team to take on the responsibility of modeling ideologies (nor analyzing how they could be manipulated), we hope that in the future, the goal of this project will become fully realized.