Assigning Semantics to Graphical Concrete Syntaxes

Athanasios Zolotas, Dimitrios S. Kolovos,
Nicholas Matragkas and Richard F. Paige

Department of Computer Science
University of York, York, UK
Email: {amz502, dimitris.kolovos, nicholas.matragkas, richard.paige}@york.ac.uk

Abstract. Graphical editors that are used in the domain of Model-Driven Engineering (MDE) follow specific conventions to denote relations between elements such as edges and containments. However, existing research suggests that there are other visual aspects that can better encode these relations such as the shape, the position and colour of the elements. In this paper, we propose the use of these physical variables to denote information regarding attributes and relations between elements. Running examples of DSLs in which such paradigms can be of benefit are presented.

1 Introduction

In the majority of graphical model editors, nodes are used to represent different types and edges to denote relations between them. Some editors also allow the use of containments to group elements that conceptually belong to the same container while others allow the replacement of geometrical shapes with icons.

The importance of visual notation in diagrams and the impact on understanding them has been confirmed by different empirical studies [1], [2], [3]. Visual characteristics of the notation can help to better understand the modelled concepts while changes to them affect the final understanding, even if the semantics of the concepts have not changed. However, research on the design of graphical concrete syntaxes for modelling languages and Domain-Specific Languages (DSLs) in general suggests that diagram-based information related to the colour, size, location of model elements is ignored by the metamodel tools as the majority of effort is put on the semantics of the notation rather that the visual representation. [3]

In the emerging community of bottom-up and flexible modelling, where domain experts are invited to create example models of the envisioned DSL, the traditional graphical MDE conventions are not always easy to follow. For instance, in a DSL that will be used to graphically design the seating plan for an event, a domain expert will likely place each guest close to the table he/she belongs to. A person not familiar with the traditional MDE conventions is not likely to use edges to connect guests to the tables or place the guests on the

tables to express the notion of containment nor would they create an attribute for each guest to hold the ID of the table she belongs to. The former notation arguably looks more natural than the latter.

Even in cases where bottom-up modelling is not used or where users are familiar with the traditional MDE conventions, the use of such physical characteristics could be beneficial. In the same example above, it would be more natural to change the table of a guest by just moving her around different tables, rather than changing the value of the appropriate attribute, or by deleting an existing node and drawing another node to the new table.

In this paper we argue that some of the physical characteristics of diagram elements can be used to extract useful information about their underlying model elements. We present examples where such information is useful. In the examples we use a flexible modelling technique, called Muddles [4], to represent our models benefiting from its model querying capabilities to evaluate our claims.

The rest of this paper is structured as follows. In Section 2 related work in the field of notation design is discussed and bottom-up flexible modelling techniques are presented. Section 3 includes a brief presentation of the Muddles approach. In Section 4 we present an number of physical attributes and illustrate via running-examples how can assist in extracting useful information like the types of elements and relations between them. In Section 5 we conclude the paper and outline plans for future work.

2 Related Work

In [3], Moody proposes a set of rules that should be followed when creating graphical notation for a modelling language. He highlights the importance of the physics of notations in the development of DSLs and the fact that this is a neglected issue so far. The *theory of communication* by Shannon and Weaver [5], is adapted by Moody for the domain of graphical notations: the effectiveness of the communication of a diagram can be increased by choosing the most appropriate notation conventions of these that the human mind can process. In [6], Bertin identified a set of 8 visual variables that can encode and reveal information about the elements they represent or their relations in a graphical way. These variables are: the *horizontal position*, the *vertical position*, the *shape*, the *size*, the *colour*, the *value* (referred as "brightness" [3]), the *orientation* and the *texture*.

In [7], [8] the authors propose a set of metamodels that can be used in the classification of visual languages taking into account spatial information. In [9] the authors present a parsing technique that can incorporated into freehand editors and turn them into syntax-aware, using different criteria such as spatial relationships. Finally, Baar [10], proposes the formal definition of the concrete syntax of modelling languages.

In the field of bottom-up metamodelling, [11] proposes the use of example models to semi-automatically infer the metamodel. In [12], the example models created by domain experts using drawing tools can be used to construct the

metamodel. In [13], models that do not conform to their metamodel because the latter evolved, can be used to recover the metamodel they are instances of. Finally, in [4], users, using a simple drawing tool, define example models which are then amenable to programmatic model management (validation, transformation, etc.).

In this work we implement the examples using an extended version of the flexible modelling technique in [4]. The same process could be followed using any other editor for flexible or traditional metamodel-based modelling.

3 Muddles

In this section, we present the basic details that will help the reader understand how the Muddles approach [4] works. In addition, we present the work carried out to extend the Muddles to keep information about the visual properties of the models.

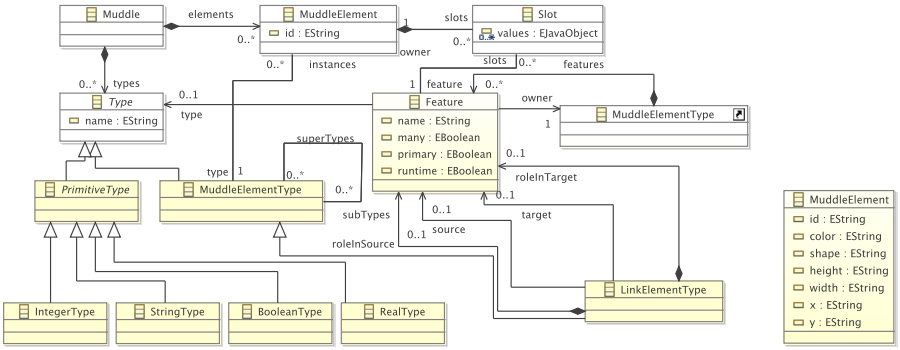
3.1 Overview

The Muddles approach [4] proposes the use of general drawing tools, for the construction of diagrams that are amenable to programmatic model management. More specifically, domain experts use a GraphML-compliant drawing tool (the yEd Editor[1] in their work) to express the example models, which conform to their envisioned metamodel. Engineers annotate these drawings to specify types and attributes for each element. The annotated diagram is then automatically transformed to an intermediate Muddle model (the Muddle metamodel is shown in Figure 1a). The Epsilon platform [14] provides an abstraction layer (the Epsilon Model Connectivity - EMC[2]) that allows access to models that conform to a range of technologies. A driver that implements EMC's interfaces and allows Epsilon to consume muddles was developed. Using the driver, model management programs (M2T transformations, Validation, etc.) can be written and executed on the muddles.

For a better understanding of the above process the authors in [4] provided an example which is presented here. In their example, the goal is to create a new flowchart-like language.

The process starts with the creation of a drawing of an example flowchart (see Figure 2). The next step is the annotation of the diagram elements with information to allow programmatic management. For instance, in this case one needs to declare that the type of the rectangles as an *Action* and the type of the directed edges as a *Transition*. The types are not bound with the shape but with each element (in another example one rectangle can be of type *Action* while another one can be of type *Process*). Types and type-related information like properties (attributes of the type), roles and multiplicity of edges can be provided using the fields in the yEd's custom properties dialog (see Figure 3). More details about these properties are presented in [4].

[1] `http://www.yworks.com/en/products_yed_about.html`
[2] `http://eclipse.org/epsilon/doc/emc/`



(a) The Muddle metamodel (b) Changes

Fig. 1.

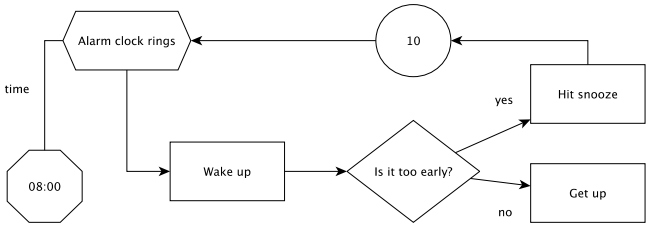


Fig. 2. An example diagram

This type-related information are keywords that will be used by the model management programs to elements of the diagram. For example, writing the following Epsilon Object Language (EOL) [15] script will return the names of all the elements of Type *Action*. (In this case, *name* was declared as a property of the *Action* node by writing *String name = "..."* in the *Properties* field of the node.)

```
var actions = Action.all();
for (a in actions) {
  ("Action: " + a.name).println();
}
```

Listing 1.1. EOL commands executed on the drawing

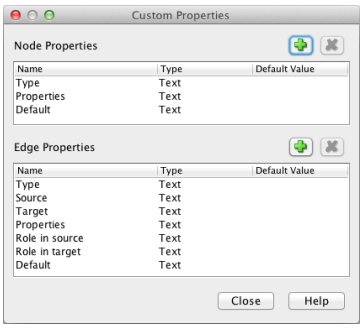


Fig. 3. Custom properties dialog

3.2 Extending Muddles

The current Muddles metamodel and the implementation of the EMC driver for muddles discard information regarding graphical and spatial properties of each element. These properties are the: *x* and *y coordinates*, the *width* and *height*, the *shape* and the *colour* of each Muddle Element.

Firstly, we extended the muddles metamodel to allow Muddle Elements hold the above information. The changes are shown in Figure 1b. The graphical and spatial properties of each element are now stored as attributes in the MuddleElement class.

Secondly, we implemented the required functionality in the EMC Muddles driver to be able to parse the GraphML file (the drawing), retrieve the information from it and store them in the muddle model instance. Further technical details about the new features of the EMC driver will not be discussed as they are beyond the scope of this paper.

4 Physical Attributes and Application Scenarios

In our running examples we demonstrate how 5 physical characteristics of the elements of graphical models can be used to extract relations and attributes. These are:

- Proximity: the distance between two or more elements.
- Colour: the colour of the element.
- Shape: the shape of the node.
- Size: the area of the node.
- Overlap: the intersection between two or more elements.

For each example, we implemented a set of functions to calculate the desired characteristic and executed it on the diagram using the querying capabilities of the Epsilon platform.

4.1 Proximity

The fact that an element is closer to another than a third one may infer that it is related to the former rather than the latter. In our scenario, we designed an example model of an envisioned DSL where the organiser of an event needs to assign guests to the tables and later perform model management actions on them (e.g. M2T transformations to generate invitation letters).

A possible example model could be the one shown in Figure 4 where 24 nodes of type *Guest* are assigned to 3 different nodes of type *Table*. Naturally, each guest belongs to the table that he is closest to. In a traditional graphical modelling editor, this relationship could be specified by creating an edge from each guest to the table it belongs, or by placing guests inside "table containments", or by assigning an attribute for each guest that declares his/her table. In our approach, this assignment is done by placing them closest to the table they belong to.

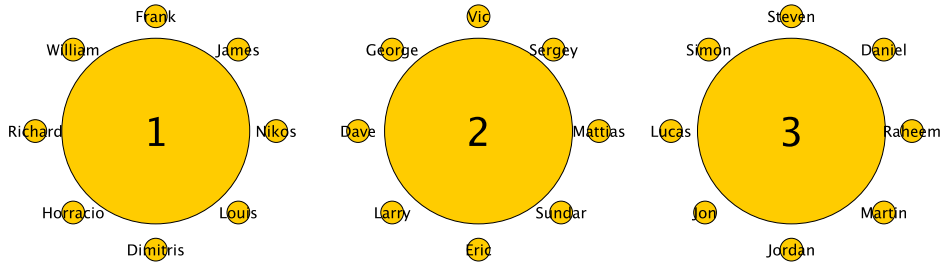


Fig. 4. Tables and Guests

To achieve this, an EOL function to calculate the proximity between two elements was implemented. Using this re-usable function (calculateProximity) we can query the models and get the relation of interest. The code for calculating and returning the closest table is illustrated in Listing 1.2.

```
function Guest getTable() {
  var minDistance = calculateProximity(getCircleCenterX(self), getCircleCenterX(
      tables.get(0)), getCircleCenterY(self), getCircleCenterY(tables.get(0)));
  theTable = tables.get(0);
  for (t in tables) {
    var candidateDistance = calculateProximity(getCircleCenterX(self),
        getCircleCenterX(t), getCircleCenterY(self), getCircleCenterY(t));
    if (candidateDistance < minDistance) {
      minDistance = candidateDistance; theTable = t;
    }
  }
  return theTable;
}
```

Listing 1.2. Get closest node of type Table

Indeed, if we query the model using the EOL statement of Listing 1.3 the correct table is returned.

```
var james = Guest.all.selectOne(p|p.name = "James");
("James belongs to table " + james.getTable().number + ".").println();
Output:
James belongs to table 1.
```

Listing 1.3. Query Guest's Table and output

We should note that the *proximity* characteristic may be error prone, as there might be cases that the user believes that a node is closer to the desired node while in reality it is closer to another.

4.2 Colour

In some cases, the colour of nodes or edges can declare that they belong to the same group or that they are of the same type.

In this scenario, we create an example model of an DSL that can be used to described football line-ups (see Figure 5). Each player belongs to a team illustrated by the colour of the node that represents each player. In a traditional MDE manner, this property could be defined in many different ways. Among others, one could use a string attribute for the name of the player's team or connect players of the same team with edges declaring a "team-mates" relation.

For this category, the function that returns the colour of the node is already implemented as part of our extended Muddles metamodel and driver (the extended Muddle metamodel stores the colour of the Element as an attribute - see

Fig. 5. Players and Teams

Figure 1(b)). We can create a mapping of colour with team and then query a Player using EOL to get his team. This is shown in Listing 1.4.

```
var teamsMap = new Map;
teamsMap.put("#FF6600", "Netherlands");
teamsMap.put("#333333", "Germany");
var vanPersie = Player.all.selectOne(f|f.name = "Van Persie");
("Van Persie plays for " + vanPersie.getTeam()).println();

function Player getTeam() {
  var color = self.getColor();
  return tMap.get(color);
}
Output:
Van Persie plays for Netherlands
```

Listing 1.4. Get Player's team implementation and output

4.3 Shape and Size

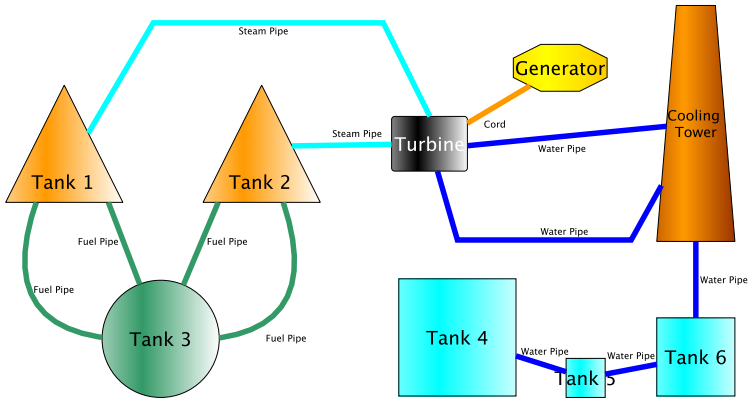


Fig. 6. Nuclear energy production

In some DSLs, the shape or the size of a node may encode information about its type or its attributes. We demonstrate that with an example DSL that can be used to design liquid tank configurations. In this scenario, the shape that is used to describe a tank, declares the subtype of the tank (Water, Uranium, etc.) By creating a mapping as in the previous example, we can query the model and identify the subtype of each tank.

In addition, the size (and the area) that each tank has can give us information about two other attributes of each tank like the "Size Category" and "Capacity". We can calculate the area of the tank to find its capacity and assign it to a predefined size category (small, medium, large). The querying code to get the type, the size category and the capacity is given in Listing 6.

```
...
for (t in tanks) {
  (t.name + " is a " + t.getSizeCategory() + " (" + t.getArea() + " litres) " + t.
      getTankType()).println();
}

function Tank getTankType() {
  return shapesMapping.get(self.getShape());
}

function Tank getSizeCategory() {
  if(self.getArea()<5000.0){
    return "Small";
  } else if (self.getArea()<20000.0) {
    return "Medium";
  } else {
    return "Large";
  }
}
Output:
Tank 4 is a Large (22500.0 litres) Water Tank
Tank 5 is a Small (2500.0 litres) Water Tank
Tank 1 is a Medium (11250.0 litres) Steam Tank
...
```

Listing 1.5. Get tank's type, size category and capacity implementation and output

The *size* characteristic can be error-prone. Mistakes can be made if the shape's area is close to the thresholds that defines different size categories. For instance, one tank may look like a small tank, but in reality it is medium.

4.4 Overlap

An overlap between two or more elements can provide us with information regarding their types and attributes. In a DSL that allows the creation of Venn diagrams this can be very useful. For instance, the Venn diagram of Figure 7 is an example of a model that would be an instance of a Venn DSL. In this case, the overlap between a node of type "Person" (yellow rectangles) with a circle denotes that the Person belongs to that set.

For this category, we can define a function to calculate whether two elements overlap or not. We can then re-use this function to query the model and receive, for instance the signatures of all the members of the department as seen in Listing 1.6.

```
var persons = Person.all;
var raBox = RA.all.first();
var rsBox = RS.all.first();
var esBox = ES.all.first();

for (p in persons) {
```

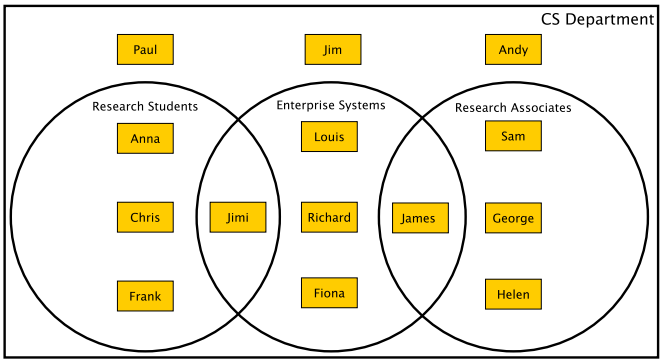


Fig. 7. Computer Science department Venn diagram

```
  p.getSignature().println();
}

function Person getSignature() {
  if((self.overlaps(raBox)) and (not (self.overlaps(esBox)))) {
    return self.name + " is a RA in the CS Department.";
  } else if ((self.overlaps(raBox)) and (self.overlaps(esBox))) {
    return self.name + " is a RA in the CS Department and member of the ES group.";
  } else if ((self.overlaps(rsBox)) and (self.overlaps(esBox))) {
    return self.name + " is a RS in the CS Department and member of the ES group.";
  } else if ((self.overlaps(rsBox)) and (not (self.overlaps(esBox)))) {
    return self.name + " is a RS in the CS Department.";
  } else if ((not(self.overlaps(rsBox))) and (not (self.overlaps(esBox))) and (not (
      self.overlaps(raBox)))) {
    return self.name + " is member of the CS Department.";
  } else if ((not(self.overlaps(rsBox))) and (self.overlaps(esBox)) and (not (self.
      overlaps(raBox)))) {
    return self.name + " is member of the CS Department and member of the ES group.";
  }
}
Output:
James is a RA in the CS Department and member of the ES group.
Andy is member of the CS Department.
Chris is a RS in the CS Department.
...
```

Listing 1.6. Get person's signature implementation and output

5 Conclusions and Future Work

Physical characteristics included in graphical models can be used to extract meaningful information about the models and their elements. In this work, we presented examples demonstrating how they can be utilised to extend the current conventions for representing relations and attributes of model elements.

We believe that such an approach can be useful especially in the flexible modelling area were the involvement of stakeholders who are unfamiliar with the traditional conventions is common.

In the future, we plan to investigate how other physical attributes (like texture or orientation) that can, according to the literature, encode information about the diagram be used in MDE to help us represent better relations and attributes of elements.

Acknowledgments

This work was carried out in cooperation with Digital Lightspeed Solutions Ltd, and was part supported by the Engineering and Physical Sciences Research Council (EPSRC) through the Large Scale Complex IT Systems (LSCITS) initiative, and by the EU, through the MONDO FP7 STREP project (#611125).

References

1. Nordbotten, J.C., Crosby, M.E.: The effect of graphic style on data model interpretation. Information Systems Journal **9**(2) (1999) 139–155
2. Hitchman, S.: The details of conceptual modelling notations are important-a comparison of relationship normative language. Communications of the Association for Information Systems **9**(1) (2002) 10
3. Moody, D.L.: The physics of notations: toward a scientific basis for constructing visual notations in software engineering. Software Engineering, IEEE Transactions on **35**(6) (2009) 756–779
4. Kolovos, D.S., Matragkas, N., Rodríguez, H.H., Paige, R.F.: Programmatic muddle management. XM 2013–Extreme Modeling Workshop (2013) 2
5. Shannon, C.E., Weaver, W.: The mathematical theory of communication. (2002)
6. Bertin, J.: Semiology of graphics: diagrams, networks, maps. (1983)
7. Bottoni, P., Grau, A.: A suite of metamodels as a basis for a classification of visual languages. In: Visual Languages and Human Centric Computing, 2004 IEEE Symposium on, IEEE (2004) 83–90
8. Bottoni, P., Costagliola, G.: On the definition of visual languages and their editors. In: Diagrammatic Representation and Inference. Springer (2002) 305–319
9. Costagliola, G., Deufemia, V., Polese, G., Risi, M.: Building syntax-aware editors for visual languages. Journal of Visual Languages & Computing **16**(6) (2005) 508–540
10. Baar, T.: Correctly defined concrete syntax for visual modeling languages. In: Model Driven Engineering Languages and Systems. Springer (2006) 111–125
11. Cho, H., Gray, J., Syriani, E.: Creating visual domain-specific modeling languages from end-user demonstration. In: Modeling in Software Engineering (MISE), 2012 ICSE Workshop on, IEEE (2012) 22–28
12. Sánchez-Cuadrado, J., De Lara, J., Guerra, E.: Bottom-up meta-modelling: An interactive approach. Springer (2012)
13. Javed, F., Mernik, M., Gray, J., Bryant, B.R.: Mars: A metamodel recovery system using grammar inference. Information and Software Technology **50**(9) (2008) 948–968
14. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.: The design of a conceptual framework and technical infrastructure for model management language engineering. In: Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on, IEEE (2009) 162–171
15. Kolovos, D.S., Paige, R.F., Polack, F.A.: The epsilon object language (eol). In: Model Driven Architecture–Foundations and Applications, Springer (2006) 128–142