Programmatic Muddle Management

Dimitrios S. Kolovos, Nicholas Matragkas,
Horacio Hoyos Rodríguez, and Richard F. Paige

Department of Computer Science, University of York,
Deramore Lane, York, YO10 5GH, UK
{dimitris.kolovos, nicholas.matragkas,
hhr502, richard.paige}@york.ac.uk

Abstract. In this paper we demonstrate how diagrams constructed using general-purpose drawing tools in the context of agile language development processes can be annotated and consumed by model management programs (such as simulators, model-to-model and model-to-text transformations). The aim of this work is to enable engineers to engage in programmatic model management early in the language development process, so that they can explore whether or not the languages and models constructed are fit for purpose. We demonstrate a proof-of-concept prototype developed atop the Epsilon platform and a flexible graph definition language (GraphML).

1 Introduction

The quality and usefulness of a Domain Specific Language (DSL) depends on accurately identifying the domain concepts, their features and relationships. As such, the involvement of domain experts in the language development process is crucial. In the early stages of the language development process, domain experts often provide informal example diagrams/sketches from which engineers can infer a first version of the metamodel of the envisioned language. To obtain additional feeback, engineers then need to develop an initial version of a language-specific modelling tool that enables domain experts to further experiment with the language. This typically constitutes the first step of an iterative process during which the metamodel of the language can undergo several revisions. When 3-layer modelling frameworks such as MOF/EMF are used, for each change in the metamodel, language engineers need to update and re-deploy a new version of the modelling tool, and for non-additive changes to the metamodel they also need to provide support for automated migration of older models.

To achieve shorter and more efficient iteration cycles, several techniques that challenge this top-down metamodel-centric approach have recently been proposed. In such approaches, the early phases of the language development process involve the construction of *example diagrams* using flexible drawing tools, which can be used to (semi-)automatically infer a rigid metamodel only once sufficient confidence in the completeness and maturity of the language has been developed.

In this paper we argue that example diagrams constructed in the context of this process should also be processable by model management programs (such as simulators, model-to-model and model-to-text transformations) so that engineers can develop additional and early confidence that the constructed language is fit for purpose. The rest of the paper is organised as follows. In Section 2 we provide an overview of related work in the field of bottom-up and agile metamodelling. In Section 3 we illustrate an approach for enabling engineers to engage in programmatic model management activities early in the language development process, and demonstrate a proof-of-concept prototype developed atop the Epsilon platform and a flexible graph definition language (GraphML). In Section 4 we conclude and provide directions to further work.

2 Background and Motivation

In [1], the authors propose an example-driven approach where users are able to construct informal diagrams using the Dia drawing tool, and these diagrams are then used to infer appropriate metamodels in an interactive manner. Similarly, in [2] the authors introduce a systematic semi-automated approach to create visual DSLs from a set of domain model examples provided by an end-user. The MetAmodel Recovery System (MARS) [3] is a semi-automatic inference-based system for recovering a metamodel from a set of instance models through application of grammar inference algorithms. This approach does not rely on example models provided by end-users, but it relies on models, which no longer conform to a metamodel due to its evolution. In [4], the authors present a tool (GraCoT) that supports co-development of EMF models and metamodels, in a loosely-coupled manner that promotes agility and simplifies the process of co-evolution.

To our knowledge, research in this area so far has focused solely on agile model construction and automated metamodel inference. In our view, to further validate the maturity and completeness of a metamodel, it is also important for language engineers to develop some confidence that models conforming to this metamodel can support the automated model management operations involved in the envisioned MDE workflow (simulation, model-to-model and model-to-text transformation etc.)

3 Proposed Approach

In this paper we illustrate an approach for rendering diagrams constructed using general-purpose drawing tools amenable to programmatic model management. An overview of the proposed approach is illustrated in Figure 1. Consistently with previously-proposed bottom-up metamodelling techniques, in this approach language engineers and domain experts can start the language development process by drawing diagrams depicting example models, which (conceptually) conform to the envisioned language, using a general purpose diagram drawing tool.

In the next stage, engineers can augment these conceptual diagrams using a set of predefined textual annotations (discussed in Section 3.2) to specify the types and features of diagram elements of interest in an agile manner. Annotated diagrams are then automatically transformed into an intermediate representation (*muddle*) that can be programmatically managed using existing model management languages.

In this work we use GraphML, the conceptual metamodel of which is illustrated in Figure 2, for diagram drawing, and languages of the Epsilon platform [5] for automated model management, but in principle this approach should be applicable to other diagram formats and model management languages.

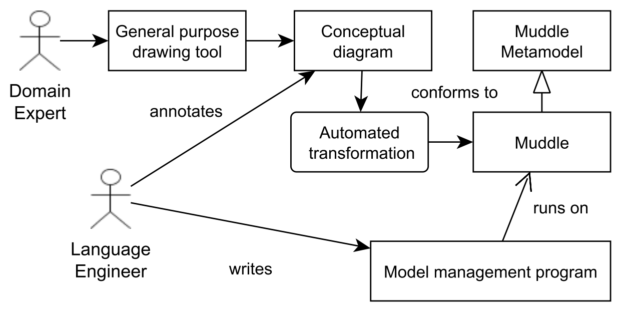

Fig. 1. Process Overview

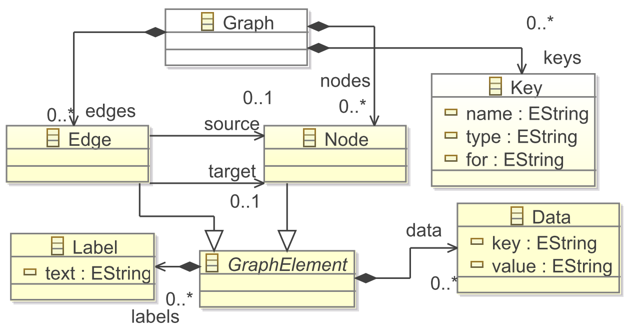

Fig. 2. GraphML Metamodel

3.1 Running Example

We illustrate the process of constructing, annotating, and programmatically managing GraphML diagrams through a running example. In this example, our aim is to define a flowchart language that supports timed events and delays. To develop some confidence that the envisioned language is feature-complete, we also need to implement a proof-of-concept program that can execute/simulate models that conform to the language.

We start by using the yEd[1] GraphML-compliant tool to draw an example diagram that conceptually conforms to the envisioned flowchart language. The diagram, illustrated in Figure 3 consists of labeled rectangles which conceptually represent actions, a diamond which represents a decision, directed edges which represent transitions, a hexagon that represents the triggering event, a circle which represents a delay, and a hexagon which represents the time at which the attached event should fire for the first time.

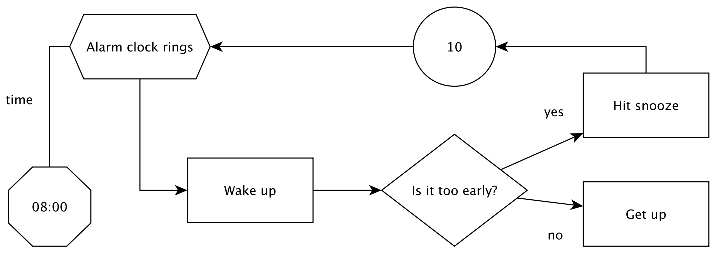

Fig. 3. Flowchart Diagram

We now take a leap and in Listing 1.1 we present the implementation of a simple simulator for such flowcharts, expressed in the Epsilon Object Language [6], an imperative OCL-based model query and transformation language. We provide a brief overview of the behaviour and the organisation of the simulator and then demonstrate how we need to annotate the diagram of Figure 3 so that the simulator program can use it as an input model that can drive its execution.

```
var event = Event.all.selectOne(e|e.entryPoint = true);
var time = event.time.hours.toMinutes();
event.process();

operation Event process() {
  ("Event: " + self.name + " at " + time.toHours()).println();
  self.outgoing.at(0).target.process();
}

operation Action process() {
  ("Action: " + self.name).println();
  if (not self.outgoing.isEmpty()) {
    self.outgoing.at(0).target.process();
  }
}
operation Decision process() {
  ("Decision: " + self.name).println();
  var random = self.outgoing.random();
  ("Chose: " + random.name).println();
  random.target.process();
}

operation Delay process() {
  time = time + self.mins;
  ("Waited for " + self.mins + "mins").println();
```

[1] http://www.yworks.com/en/products_yed_about.html

```
26  self.outgoing.at(0).target.process();
27 }
28
29 operation String toMinutes() : Integer {
30  var parts = self.split(":");
31  return parts[0].asInteger() * 60 + parts[1].asInteger();
32 }
33
34 operation Integer toHours() : String {
35  return (self / 60).asString().pad(2, "0", false) +
36    ":" + (self - (self / 60)*60).asString().pad(2, "0", false);
37 }
```

Listing 1.1. Simple flowchart simulator

- Assuming that a flowchart can contain many events, in line 1 we select one event that has its *entryPoint* attribute set to true;
- In line 3, we keep a copy of the time (converted to minutes) at which this event is fired for the first time;
- In line 4, we process the target of the first outgoing transition of the event; Calls to *process()* operations are dynamically dispatched depending on the type of their context object, and behave as discussed below;
- The *Event.process()* operation prints a message and processes the target of its first outgoing transition;
- The *Action.process()* operation prints a message and then, if the action has any outgoing transition, it processes the target of the first of them;
- The *Decision.process()* operation chooses a random outgoing transition, prints its name and processes its target;
- The *Delay.process()* operation adds the delay time to the global time, prints a message and then processes the target of its first outgoing transition;
- The *toMinutes()* and *toHours()* operations can convert HH:MM-formatted time strings to integers (number of minutes) and vice versa.

A sample execution trace of the simulator appears below.

```
Event: Alarm clock rings at 08:00
Action: Wake up
Decision: Is it too early?
Chose: yes
Action: Hit snooze
Waited for 10mins
Event: Alarm clock rings at 08:10
Action: Wake up
Decision: Is it too early?
Chose: no
Action: Get up
```

3.2 Annotating GraphML Diagrams

To facilitate the execution of model management programs such as the one illustrated in Listing 1.1, we need to annotate diagram elements with additional information. For example, we need to declare that the type of all rectangle nodes in this diagram is *Action*, and that the type of directed edges is *Transition*. As GraphML does not provide built-in support for capturing type-related information for nodes and edges, we need to use GraphML's extensibility facilities[2] to define *Type* extension fields for nodes and edges.

[2] http://docs.yworks.com/yfiles/doc/developers-guide/graphml.html

The value of the *Type* extension field of a node/edge needs to adhere to the `name (> name)*` pattern, where > is used to denote inheritance. For example, by setting the *Type* field of the *Wake up* node to *Action* > *FlowchartElement*, we define that the node is an instance of the *Action* type, and that the *FlowchartElement* type is a super-class of *Action*. All types are unique by name and are created the first time they are encountered in the diagram. For example, by subsequently setting the *Type* field of *Hit snooze* to *Action*, we are reusing the *Action* type defined in *Wake up* instead of creating a new one. Beyond type-related information, we also need to capture additional information using the following GraphML extensions summarised in Table 1.

Table 1: GraphML extensions

Extension	**For**	**Description**	**Pattern**
Properties	Node, Edge	Descriptors and values for primitive attributes of nodes/edges	(int|String|boolean|real)? (*)? name (= value)?
Default	Node, Edge	Descriptor of the slot under which the first label of the node/edge should be made accessible	(int|String|boolean|real)? name
Source role	Edge	Descriptor of the role of the source end of the edge	name (*)?
Target role	Edge	Descriptor of the role of the target end of the edge	name (*)?
Role in source	Edge	Descriptor of the role of the edge in its source node	name (*)?
Role in target	Edge	Descriptor of the role of the edge in its target node	name (*)?

The value of the *Properties* field of a node/edge can contain zero or more lines of text. Each line needs to adhere to the pattern above and define the type, multiplicity, name and value of the property. For example, by setting the value of the *Type* field of the *Alarm clock rings* node to *Event* and the text of its *Properties* field to *boolean startEvent = true*, we define that the node has a single-valued boolean *startEvent* property, with a value set to *true*.

The value of the *Default* field should conform to the pattern above and define the name of the default slot of the node/edge and, optionally, its primitive type (defaults to String). For example, by setting the *Default* field of the *Wake up* node to *name*, the first label of the node that does not match the property descriptor pattern (in this case, the *Wake up* label), will be made accessible through a *name* property of type String.

The values of the *Source role*, *Target role*, *Role in source*, and *Role in target* fields of an edge define the name and multiplicity of the respective roles. For example, in the *yes* transition we define the following values for these properties:

Source role: *source*, Target role: *target*, Role in source: *outgoing **, Role in target: *incoming **.

3.3 Deriving a Muddle

The next step of the process is to parse the annotated GraphML diagram and construct an intermediate model (*muddle*) that conforms to the metamodel of Figure 4. This is achieved through a multi-pass transformation which is transparent to the end-user and which comprises the following steps.

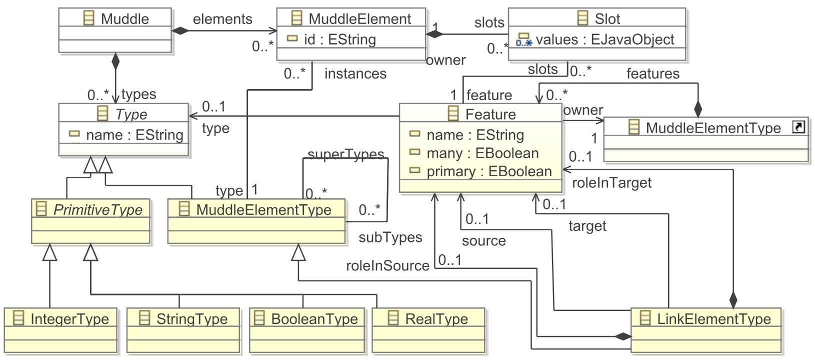

Fig. 4. Intermediate (Muddle) Metamodel

1. For every typed node in the graph, it creates an empty *MuddleElement* in the intermediate model and its corresponding *MuddleElementType* (if the latter does not already exist). It also looks for nodes for which the *Default* field has a valid value. When this happens, the value of the *Default* field is used to produce a primary *Feature* which is added to the type of the created *MuddleElement*;
2. Iterates through the created elements and creates/populates their *slots*, based on the descriptors provided in the *Properties* field of the node. Again, for each new property a *Feature* is created and added to the type of the element. As such, by setting the value of the *Properties* field of *Alarm clock rings* to *boolean startEvent = true*, all model elements of type *Event* obtain a single-valued *startEvent* boolean feature;
3. Iterates through the labeled and untyped edges of the graph (e.g. the *time* edge in the diagram of Figure 3). For each edge, it adds an untyped *Feature* to the type of its source muddle element, a respective *Slot* to the source muddle element, and adds the target of the edge to the values of the slot;
4. Iterates through the unlabeled and untyped edges of the graph and attempts to fit their targets into appropriate slots of the source muddle elements (i.e. slots that already contain at least one value of the same type);

5. For every typed edge of the graph it creates an empty *MuddleElement* and its corresponding *LinkElementType*, similar to what was discussed for nodes in step 1. It also attempts to create primary, role in source, role in target, source and target *Features* for the created *LinkElementType*s;
6. Iterates through the typed edges of the graph and creates/populates their slots similar to what was discussed in step 2;
7. Adjusts the multiplicities of features based on the maximum number of values of their slots. In this process, single-valued features, slots of which contain more than one values become multi-valued (but not the opposite).

3.4 Consuming Muddles in Epsilon Programs

Epsilon provides an abstraction layer (Epsilon Model Connectivity – EMC[3]) that shields the languages of the platform from the intricacies of concrete model representations and enables them to access models conforming to a wide range of technologies. To enable Epsilon languages to access muddles, we have developed a new driver that implements the set of interfaces required by EMC. Due to space restrictions, a detailed discussion on the new driver is beyond the scope of this paper.

The driver enables all languages in Epsilon to query muddles. For example, in addition to the simulator of Listing 1.1, Listing 1.2 demonstrates an exemplar constraint written in the validation language of the platform (EVL[4]), and Listing 1.3 demonstrates an exemplar model-to-text transformation written in EGL[5].

```
context Decision {
  constraint HasMoreThanOneOutgoingTransitions {
    check: self.outgoing.size() > 2
    message: "Decision " + self.name + " needs to have at least 2 outgoing
         transitions"
  }
}
```

Listing 1.2. Validation constraint for flowchart models

```
The flowchart has [%=Action.all.size()%] actions:
  [%for (action in Action.all) {%]
  - [%=action.name%]
  [%}%]
```

Listing 1.3. Model-to-text transformation for flowchart models

4 Conclusions and Further Work

In this paper we have argued for the importance of enabling engineers to engage in exploratory model management operations early on in the language development process and demonstrated an approach and a prototype that enables

[3] http://www.eclipse.org/epsilon/doc/emc
[4] http://www.eclipse.org/epsilon/doc/evl
[5] http://www.eclipse.org/epsilon/doc/egl

engineers to annotate and programmatically manage GraphML diagrams using languages of the Epsilon platform. In the future, we plan to investigate supporting additional GraphML constructs such as subgraphs and hyperedges.

In our view, while constructing diagrams using using general-purpose drawing tools can be very useful in the early phases of the language development process, it can become cumbersome and error-prone as the example diagrams and the DSL become larger and more mature - at which stage a transition to a language-specific modelling tool should be consider. To reduce the overhead of this transition, we plan to investigate inferring annotated metamodels that can then be consumed by tools such as Eugenia[6] to automatically generate language-specific model editors.

Acknowledgements

This research was part supported by the EPSRC, through the Large-Scale Complex IT Systems project (EP/F001096/1) and by the EU, through the Automated Measurement and Analysis of Open Source Software (OSSMETER) FP7 STREP project (318736).

References

1. Jesús Sánchez-Cuadrado, Juan Lara, and Esther Guerra. Bottom-up metamodelling: An interactive approach. In Robert France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 3–19. Springer Berlin Heidelberg, 2012.
2. Hyun Cho, J. Gray, and E. Syriani. Creating visual domain-specific modeling languages from end-user demonstration. In *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*, pages 22–28, 2012.
3. Faizan Javed, Marjan Mernik, Jeff Gray, and Barrett R. Bryant. Mars: A metamodel recovery system using grammar inference. *Inf. Softw. Technol.*, 50(9-10):948–968, August 2008.
4. Villalobos J. Gómez P., Sánchez M. Gracot, a tool for co-creation of models and metamodels in specific domains. In *Proc. Academics Tooling with Eclipse (ACME 2013) at European Conference on Object-Oriented Programming (ECOOP2013).* ACM, 2013.
5. Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, Fiona A.C. Polack. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *Proc. 14th IEEE International Conference on Engineering of Complex Computer Systems*, Potsdam, Germany, 2009.
6. Dimitrios S. Kolovos, Richard F.Paige and Fiona A.C. Polack. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.

[6] `http://www.eclipse.org/epsilon/doc/eugenia`