On General-purpose Textual Modeling Languages

Martin Mazanec and Ondřej Macek

Department of Computer Science, FEL, Czech Technical University,
Karlovo namesti 13, Praha, Czech Republic
{mazanma3, macekond}@fel.cvut.cz

Abstract. Modeling is an important part of the software development process because it allows for a better understanding of the domain as well as an understanding of the software structure and function. Among general-purpose modeling languages dominate the graphical ones such as UML; textual modeling languages are not as popular though they have a big potential. In this paper we define the important features of textual modeling languages and then we compare existing general-purpose textual modeling languages according to these criteria to show if they meet their potential. Based on the comparison results and our experience, we propose our own modeling language called Earl Grey whose basics are presented in this paper together with our experience from creating this language.

Keywords: textual modeling, modeling languages, model driven development

1 Introduction

Modeling is an integral part of the software development process, where it helps to explain the static part of the system (data the software works with and software inner structures and states) and the dynamic part of the system (how the software works). A lot of modeling languages exists; the best known and most widespread is the Unified Modeling Language (UML) [12], which represents the so called general purpose modeling languages. Besides the general purpose modeling languages, domain specific languages (DSL) [3] exist, whose aim is to describe a concrete domain only, therefore their usage is limited. Nevertheless the DSLs are very popular nowadays, in contrast with UML a lot of DSLs are textual. This may be caused by the fact that general purpose language is primarily focused on the possibility to describe many various problems, where the graphical representation could be helpful even if it could cause a problem later with the ambiguity of graphical structures and their meanings [2]; moreover many graphical models are extended by some textual information which completes or refines the model interpretation (e.g. together with UML models the Object Constraint Language (OCL) [11] is used). On the other hand DSLs are focused on a single domain and are often integrated into the software code so no ambiguity is allowed. The textual modeling languages (TML) can benefit from the popularity of DSLs and they can be improved by using DSL's best practices.

J. Pokorný, V. Snášel, K. Richta (Eds.): Dateso 2012, pp. 1–12, ISBN 978-80-7378-171-2.

The unambiguity of model symbols and constructs interpretation is particularly important for Model Driven Development (MDD) because transformations between various models and layers of software require clearly defined inputs in order to maintain consistency between models. That is why there are attempts to define the UML language formally [2] or to create a general-purpose modeling language with exact specification and therefore with no problems with interpretation of models. The attempts for new general-purpose language are often created as textual languages because the textual languages allow easy formal definition, and moreover they are not limited by modeling tool capabilities and maturity [4].

The next reason why the new modeling languages are textual is that it is quite easy to create a new language and integrate it into a development environment such as Eclipse or NetBeans; another possible reason for the textual modeling is the simplicity of creating a textual model (especially for developers which are used to create programs this way) compared to intrusive form filling in graphical modeling tools [4].

The textual modeling languages have big ambitions, but there is no framework that could help to evaluate TML capability and maturity; therefore we define a set of features a TML should have to help us decide which language is the best one. Requested features are based on our experience and experiments with existing TMLs and are discussed in detail later, as well as the evaluation of existing TMLs. Based on the evaluation and experiments with TMLs we decide to create an alternative TML, which will fulfill the defined requirements better than existing TMLS - its name is Earl Grey (EG). The creation of the EG language was not an easy task and we had to solve several serious problems during the language proposal. There is a discussion of these problems together with possible solutions in this paper.

The paper is organized as follows: a set of required features of TML is defined in Section 2, then existing TMLs are evaluated in Section 3 and experimental language is presented in Section 4 together with a discussion of problems connected with the proposal of a general-purpose TML.

2 Required Features of TML

The required features of a TML are defined and discussed in this section. The list provides an overview of the most important ones and all mentioned features should be considered during the creation of a TML so it can be used without problems. Defined features should help with user experience with the language as well as with its usage in MDD or other automatic processing. It is important to realize the TML features are different from features of general programming languages such as Java or C, because the TMLs have a different purpose, and thus rather than focus on type system or object-orientation of the language, it is important to focus on other features, such as readability and unambiguity, which are similar to the features of DSLs [5] [4]. A lot of these features cannot

be evaluated by an exact measurement, however they can help the user to decide whether to use the language or not.

The Ability to Describe Whole Software As we focus on general purpose TMLs the ability to describe whole Software is an important feature of such a language. According to [6] there are five views on the software - use case, logical view, process view, development view and physical view. This means each TML has to be able to describe static and dynamic part of the system from the customer and developer point of view. There is probably no such a language that can describe all views at once; rather there is a set of languages each describing one view of the software. This construction is similar to the UML, where there are different kinds of models for different views. It is important that a meta-model of the TML exists so that the constructs used in one view description can be recognized in another view and so the dependencies between models can be traced and used. TML has to provide enough expression power to describe a modeled subject. This criterion is hard to evaluate because there is no way how to measure the completeness of modeling language (what should be part of the model and what exceeds models limits); sometimes a compromise between completeness and easy or general usage has to be made.

The Ability to Describe Various Levels of Abstraction In the MDD we differ four levels of model abstraction - a level describing a domain on computing independent level (CIM), a level describing a software independently on a concrete implementation platform (PIM), a level describing the software in the context of concrete implementation platform (PSM) and physical deployment; the last level is the software code itself. In the situation when we use TML, the last two levels can be considered the same - from our point of view it is not necessary to distinguish between the source code (and configuration files) and the PSM model, because both defines the same situation with respect to concrete platform.

The description on the CIM level has to be able to cover the customer domain in a way in which the model could be understood by a customer, and at the same time it should provide enough information for a software analyst or developer. The description on the PIM level should extend the information from CIM by adding some implementation details that explain how the software will be implemented, however they will not be specific for a concrete platform or framework.

Readability and Simplicity of Language The readability is very important for each language especially if it should be used for communication with a customer (CIM level). The TML has to be easy to read and easy to write so that models can be created or validated by a customer with almost no technical skills. The simplicity of a language is not only a customer requirement, because developers appreciate a language that is easy to write and read, too.

Unambiguity of TML Expressions The lack of unambiguity is the main problem of most graphical modeling languages therefore it is important for each TML to provide expressions with no ambiguities. This criterion is important for MDD because the ambiguities in models lead to misinterpretations during model-to-model transformations or code generation. Part of the expression unambiguity is the definition of relations between TML constructs - such as the meaning of association between objects or the extension of one object by another.

Supportability and Integrability The requirement on supportability and integrability is defined by [5], where supportability means TMLs feasible to provide DSL support via tools, for typical model and program management, e.g., creating, deleting, editing, debugging, transforming and integrability means the language, and its tools, can be used in unison with other languages and tools with minimal effort. This is essential to integrating the TML with other facilities used in the engineering process. An alternative requirement for TML is extensibility, i.e., that the TML can be extended to support additional constructs and concepts. It means the stable core language that could be extended is over frequent changes of the language.

We believe the most important features of a TML are readability for people, unambiguity of language expressions and capability to describe the whole software, because these features determines the acceptance of the TML by users.

3 Comparison of Existing TMLs

This section provides an overview of existing textual modeling languages and their comparison according the criteria defined in Section 2. The overview provided in this paper could not cover all existing languages; on the other hand it should provide an representative overview (for next languages evaluation see [9]).

To provide a comparison we decide to create a set of UML models that represent different views on a system and that contains several nontrivial constructs of the language. Three of the used models can be seen in Figure 1 there is a class model, a model of activities and a state model. The models do not cover the whole expressive capability of UML, rather they represent only a part from all models we used for evaluation. The purpose of this chapter is to illustrate the method and show the basic concepts of presented TMLs.

3.1 PlantUML

PlantUML [13] is a language that allows the describing of UML models directly within the source code of software. The UML models are then part of the code (as specialized comments) which is useful as there is one source of information. PlantUML can describe all required views on the software on different levels of abstraction; it contains definitions for modeling of use cases, class models,

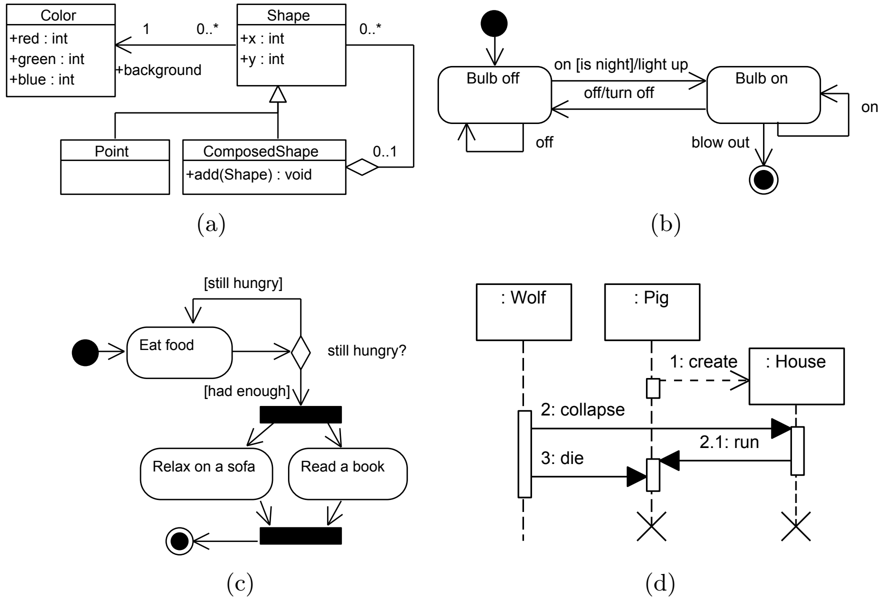

Fig. 1: A selection from the set of UML models used for TMLs evaluation. There is a class model (1a), state model (1b), activity model (1c) and sequence model (1d) in the figure; together these models represent both the static and dynamic view on the software.

state and activity models. The integrability is guaranteed by the integration of PlantUML into Eclipse IDE.

The problem of PlantUML is its readability, as the language copies not only the UML standard, but also the graphical constructs (see Listing 1) that expect the user to be familiar with UML. The next problem is with the usage of state or activity models that become confusing and unreadable.

```
class Color {
  +red : int
  +green : int
  +blue : int
}
class Shape { ... }
class Point {}
class ComposedShape { ... }

Shape <|-- Point
Shape <|-- ComposedShape
ComposedShape "0..1" o-- "0..*" Shape
Shape "0..*" --> "1" Color
```

Listing 1: The class model from Figure 1a in the PlantUML language, the usage of arrows expects the user is familiar with the UML syntax.

```
state "Bulb Off" as bulbOff
state "Bulb On" as bulbOn

[*] --> bulbOff
bulbOff --> bulbOff : off
bulbOff --> bulbOn : on [is night]/light up
bulbOn --> bulbOff : off/turn off
bulbOn --> bulbOn : on
bulbOn --> [*] : blow out
```

Listing 2: The state model from Figure 1b in the PlantUML language, usage of pseudostates could be confusing.

```
(*) --> "Eat food"
if "still hungry?" then
  -->[still hungry] "Eat food"
else
  ->[had enough] ===Fork===
endif
===Fork=== --> "Relax on a sofa"
===Fork=== --> "Read a book"
"Read a book" --> ===Join==
"Relax on a sofa" --> ===Join==
===Join== --> (*)
```

Listing 3: The activity model from Figure 1c in the PlantUML language is hard to read for larger models.

```
package testpackage;
  class Color
    attribute red : Integer;
    attribute green : Integer;
    attribute blue : Integer;
  end;
  class Shape end;
  class Point specializes Shape end;
  class ComposedShape specializes Shape end;

  association
    navigable role background : Color[1];
    role shape : Shape[*];
  end;
  aggregation
    navigable role child : Shape[*];
    navigable role parent : ComposedShape[0, 1];
  end;
end.
```

Listing 4: TextUML version of the class model from Figure 1a is quite verbose therefore it is not simple to create a model.

3.2 TextUML

Text UML [1] is TML that is specialized only at class model, therefore the capability of describing various views on the software is of course limited. This limitation is partially compensated by the readability of its models that are readable even for non-developers. The grammar of the language can be understood intuitively as it refers to UML and common programming language, however we were not able to find any formal definition of the language. In contrast with PlantUML, TextUML does not suppose the user knows UML concepts; on the other

hand it could be considered verbose as it requires a large amount of information about classes and relations, even though some of them are not necessary.

3.3 Umple

The main goal of the Umple language [8] is model-oriented programming that is based on class and state modeling and code generation into Java, Ruby and other languages. The mentioned models are the only one that can be used for modeling so some views on software are missing (use cases, processes). The Umple is the only language that provides the definition of its grammar.

The models are readable and no major complication with language usage were observed. The Umple language is integrated in Eclipse IDE or the online service Umple Online [7] can be used.

```
class Color {
  int red;
  int green;
  int blue;
}
class Shape { ... }
class ComposedShape {
  isA Shape;
  void add(Shape e);
}
```

```
class Point {
  isA Shape;
}

association {
  0..1 ComposedShape -- 0..* Shape;
}
association {
   0..* Shape -> 1 Color;
}
```

Listing 5: Umple version of the class model from Figure 1a, there are no serious problems.

```
class Bulb {
  state {
    Initial {
      init -> BulbOff;
    }
    BulbOff {
      on [isNight] -> / {lightUp();} BulbOn;
      off -> BulbOff;
    }
    BulbOn {
      off -> / {turnOff();} BulbOff;
      on -> BulbOn;
      blowOut -> Final;
    }
    Final { }
  }
}
```

Listing 6: Umple version of the state model from Figure 1b, there are no serious problems.

3.4 yUML

The purpose of the yUML [15] tool is fast and easy creation and publication of UML class model and activity model diagrams. The language is focused on only two languages, and its syntax does not allow for modeling large models in a readable way. Therefore yUML will probably remain a tool for the creation and sharing of small code snippets rather than become a widely accepted standard for general-purpose TML.

```
[Color|red;green;blue]<1-0..*[Shape]
[Shape]^[Point]
[Shape]^[ComposedShape]
[ComposedShape]<>0..1-0..*[Shape]
```

Listing 7: The yUML version of the class model from Figure 1a shows that a definition of a class with many attributes could be confusing.

```
(start)->(Eat food)
(Eat food)-><if>had enough->|fork|
<if>still hungry->(Eat food)
|fork|->(Relax on a sofa)->|join|
|fork|->(Read a book)->|join|->(end)
```

Listing 8: The yUML activity model is quite verbose and becomes confusing for large models.

3.5 Comparison Summary

The experiments show that existing TMLs do not meet the defined criteria and so the potential of TMLs. The main problem is that the TMLs designer tries to describe the UML model, not the domain (classes, states etc.) and that is in contrast with the recommendations of [16] and it reduces the usability and readability of large models.

Feature	PlantUML	TextUML	Umple	yUML	EG	UM
Multiple views on software	yes	no	no	no	yes	yes
Readability and Simplicity	no	yes	yes	no	yes	yes
Provided Language Definition	Grammar	-	Grammar	-	Grammar	MO
Integrability	Eclipse	Eclipse	Eclipse/Online	Online	Eclipse	man

Table 1: The overview of experiments with existing TMLs and UML. Most of the tested TMLs could not describe all views on software; some of them have problems with readability.

4 Experimental Modeling Language

The previous sections show that the existing TMLs do not meet the potential of textual modeling and they are not usually able to cover all requested views on software and their semantic was not specified, rather it was based on a previous knowledge of UML or common programming languages and intuitive understanding of concepts such as generalization, association or aggregation. Moreover, the TMLs are often hard to read and write and they are not usable for large models. Therefore we decide to create our own textual modeling language called Earl Grey (EG), whose concepts and creation is explained in the following text. In

this paper we do not focus on formal specification, as it is beyond the scope of this paper; instead we will focus more on user experience with the language. The main purpose of this section is to show Earl Grey's fundamental differences from other textual modeling languages.

The EG is implemented as an Eclipse plugin using Xtext [14] and the latest version of its grammar is available on-line [9]. There is an implementation of class and state model at the moment. Use case and activity model languages should be finished in the near future, therefore EG should cover all requested views on the software. All EG models are connected with the CIM or PIM group of models and the PSM is represented by the code itself.

The next sections discuss problems we have to solve during the Earl Grey implementation. The majority of the problems are caused by differences in presentation of information in graph and textual environment. We will compare our construct mostly with PlantUML because from the aforementioned TMLs, it is the most complex one.

4.1 Language for Class Modeling

When creating each model we focus on creating a language that will fit the modeling problem, whereas a lot of TMLs try to rewrite the UML models by text. A typical example is the PlantUML language and its description of associations between classes in a class model. There are examples of PlantUML associations in Listing 1, and you can see the symbols are visually close to the UML symbols, which is good if users already understand UML but for users who will meet modeling for the first time these symbols will be confusing. In contrast we decide to use a textual representation of each association type as you can see in Listing 4.

```
class Color
  red : int
  green : int
end
class Shape
  /*...*/
end
class Point isA Shape
end
class ComposedShape isA Shape
  /*...*/
end
```

```
aggregation
  0..1 ComposedShape /*start*/
  0..* Shape /*end*/
end

association
  0..* Shape
  1 Color
end
```

Listing 9: EG version of the class model

We believe the representation proposed in Listing 9 is more readable for users of the language, and they obtain much more information than from PlantUML symbols. The associations are represented as a sentence so it can be read and understood with no need to know the meanings of pseudo-graphical symbols in PlantUML. Next we prefer `isA` for inheritance indication over `extends` or graphical symbol `<|--`, the reason is educational and expression `isA` should help with inheritance usage (the problem of bad inheritance usage is described e.g. in [10]). We believe the `isA` construct will improve the design of future code.

4.2 Language for State Model

The PlantUML state model language (in Listing 2) does not allow logic structuring of a created model; all states and transitions are in one large cluster that decreases the readability of the model.

The proposed language allows the splitting of the model into small sections – one section for each state and its transitions. This structure improves not only the readability of a model but also the change of transitions. In a PlantUML model a user has to check the whole model to find the changed transition, whereas in the EG state model the information is right in the state, where the transition starts.

The next change we made against PlantUML is that we omit the usage of graphical symbols – arrows (as in class model) and asterisk that PlantUML uses for representation of initial and final (pseudo)state of the machine. Instead of asterisk symbol `(*)` we use the keywords `initial` or `final`.

```
initial "Init"
  do
    light up -> "Bulb off"
  end
end

state "Bulb off"
  off do
    -> "Bulb off"
  end
  on do
    if is night then
     light up -> "Bulb on"
  end
end
```

```
state "Bulb on"
  off do
    turn off -> "Bulb off"
  end
  on do
    -> "Bulb on"
  end
  blow Out do
      -> End
  end
end

final "End"
end
```

Listing 10: EG version of the state model

4.3 Language for Behavior Modeling

To model the behavior is an important part of software modeling. The UML language provides two models - activity and sequence model; we use them as templates because we want to preserve user experience with these models and their expression abilities.

The first sequence model describe the communication (message sending) among objects (for example of a model see Figure 1d). In the case that we try to describe the same information textually, we face the problem of low readability of the model because it is hard to provide a textual description of object interactions so that the model is logically structured and allows easy orientation. There is the textual version of the model from Figure 1d created according EG grammar in Listing 11. We believe the list of messages in the EG model becomes confusing for large models and it will not satisfy the condition of readability. The expression capability of UML sequence model lies in the lifelines representation of objects and in the time ordering of messages in left-to-right and top-to-bottom directions; therefore we try to create a similar user experience in TML, but all attempts end up with constructs that were unreadable even for small models.

The visual representation of messages and their sequences provides the next level of user experience that the textual language cannot provide.

```
sequence WolfAndPig
  Pig creates House
    Wolf calls House.collapse
      House calls Pig.run
    Wolf calls Pig.die
end
```

Listing 11: The textual sequence model is not as readable as the graphical; in the case of parallelism or conditional branches it becomes confusing.

Next possibility for the behavior modeling in UML is an activity diagram that focuses on business processes modeling and workflow representation. The representation of workflow is quite hard in TML; the existing languages yUML and PlantUML only rewrite the workflow as a set of activities and transitions between them, and the resulting model has no logical structure and it is unsuitable for large models.

The TML is capable of describing activities and transitions in sufficient detail in a single swim-lane; on the other hand there are difficulties with modeling of decisions (conditional branching), parallelization and cross swim-lanes transitions and relationships. The reason is the loss of graphical information during the rewriting of a model from graphical to textual form. When the model is rewritten we are able to capture the processes, however we lose the information about their flow.

There is not a solution that will help to solve problems with capturing the sequence or flow in TML in a readable and structured way. The compromise can be made between flow capturing and logical structure of a textual model.

5 Conclusion and Future Work

The textual modeling languages are said to have a great potential, and in this paper we discussed features required for the success of a general-purpose TML among users, the most important are readability for people and unambiguity of language expressions.

Already existing TMLs suffer from lack of both important features, which is often caused by a usage of UML-like symbols in textual language. The UML-like symbols and concepts are hard to read and often hard to interpret ambiguously. Therefore we decided to create our own TML that does meet defined criteria. There are presented several basic language grammar constructions that should improve the requested features.

Our experience is that it is easy to create a language for class and state modeling, but on the other hand the creation of a language describing behavior is very complicated, because it is hard to textually represent sequences of messages in a structured, well-arranged way. The future work in the area of TML should focus on finding a way of sufficient behavior modeling, and at the moment a pseudocode looks to be the best way.

References

1. R. Chaves. TextUML Toolkit - textuml. `http://sourceforge.net/apps/mediawiki/textuml/index.php?title=TextUML_Toolkit`, 2011.
2. A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *The Unified Modeling Language.«UML»'98: Beyond the Notation*, pages 514–514, 1999.
3. M. Fowler. *Domain-Specific Languages.* Addison-Wesley Professional, 1st edition, 2010.
4. H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Text-based Modeling. *4th International Workshop on Software Language Engineering*, 2007.
5. D. Kolovos, R. Paige, T. Kelly, and F. Polack. Requirements for Domain-specific Languages. *Proc. of ECOOP Workshop on Domain-Specific Program Development.*
6. P. Kruntchen. The 4+1 View Model of Architecture. *IEEE Software*, 12:42–50, 1995.
7. T. Lethbridge. Umple Online. `http://try.umple.org/`, 2012.
8. T. Lethbridge, A. Forward, and O. Badreddin. Umplification: Refactoring to Incrementally Add Abstraction to a Program. *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 220–224, 2010.
9. O. Macek and M. Mazanec. tea-pot/earl-grey - GitHub. `https://github.com/tea-pot/earl-grey`, 2012.
10. B. Meyer. The many faces of inheritance: a taxonomy of taxonomy. *IEEE Computer*, 29(5):105–108, 1996.
11. Object Management Group. Object Constraint Language 2.0, 2006.
12. Object Management Group. Unified Modeling Language Specification 2.3, 2011.
13. A. Roques. PlantUML. `http://plantuml.sourceforge.net/index.html`, 2012.
14. The Eclipse Foundation. Xtext. `http://www.eclipse.org/Xtext/`, 2012.
15. H. Tobbin. Create UML diagrams online in seconds, no special tools needed. `http://yuml.me/`, 2012.
16. D. Wile. Lessons learned from real DSL experiments. *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 1–10, 2003.