On the Way of Bottom-Up Designing Textual Domain-Specific Modelling Languages

Bastian Roth, Matthias Jahn, Stefan Jablonski

University of Bayreuth
Bayreuth, Germany
{bastian.roth, matthias.jahn, stefan.jablonski} @ uni-bayreuth.de

Abstract

The development of domain-specific modelling languages (DSMLs) is not a trivial task. During recent years, a new approach has arisen which enables users to sketch example models that are used as basis for deriving an appropriate DSML. Until now, this bottom-up approach is merely applied to graphical DSMLs. However, the field of textual DSMLs is also very large and we believe that it can benefit from the bottom-up method as well. To really support users during this method it is necessary to equip them with an intuitively utilizable tool. In case of textual DSMLs, this needs to be an editor that allows for entering free text. Hence, in this paper we present the requirements for such an editor and how a basic solution may look like. Finally, we state some further challenges that need to be solved to achieve full support for developing textual DSMLs the bottom-up way.

Categories and Subject Descriptors D.2.1 [**Software Engineering**]: Requirements/Specifications – Elicitation methods (rapid prototyping), Languages, Methodologies, Tools.

General Terms Design, Languages.

Keywords Domain-Specific Modelling Language, Domain-Specific Language, Bottom-Up Modelling, Demonstration-Based Approach, Textual Modelling

1. Introduction

Designing a domain-specific modelling language (DSML) is a complex and hence time-consuming task [5]. Following the widespread top-down method, the DSML needs to be defined first before it can be used to create models in the domain of interest. A user cannot simply describe the reality in form of models because (s)he needs a language for it. Thereby, one has to abstract the language from instances gathered, e.g., from an interview with a domain specialist. That is one reason why modelling tools are seldom used in early phases of software development and why the creation of a DSML is not trivial.

For graphical DSMLs, some solutions to this problem have recently been presented [4, 6, 17]. They allow for sketching diagrams in the manner of prominent drawing tools like Microsoft Visio, Microsoft PowerPoint or Dia. Based on the shapes used within the diagram, their visual properties and their relationships to each other, a DSML is derived which can be used for creating further models. Owing to this approach, less experienced language designers are supported in developing a DSML while having their well-known drawing tool behaviour. Thereby, the user shows the system how her/his DSML is structured instead of specifying it by means of an abstract formalism (e.g., meta modelling). This course of action is often called "bottom-up (meta) modelling" [16, 17] or "demonstration-based approach" [3]. The first term implies a separation from conventional top-down meta modelling so we will refer to this one.

Beyond graphical DSMLs, there is the huge field of textual DSMLs. Many frameworks and systems exist for creating and using textual DSMLs (e.g. Xtext, MPS, EMFText, Spoofax). According to their editing paradigm, these frameworks can be subdivided into two categories, namely free text editors and projectional editors.

The most widespread ones are free text editors. In a first step, they enable users to input arbitrary character sequences. Afterwards, these sequences are validated concerning syntactical correctness using a lexer and a parser which both base upon a formal grammar [1]. On the other hand, there are so-called projectional editors like Jetbrain's Meta Programming System (MPS) and the Intentional Domain Workbench. User interactions with the editor are directly performed on the underlying abstract syntax tree (AST) instead of manipulating character strings [7]. Of course, the user is always faced with a visual representation (instance of a concrete syntax) which is one projection of the AST and not the tree itself.

However, none of the textual DSML frameworks supports an approach comparable to bottom-up modelling. All of them require a definition of the used DSML before a model can be created. Focusing this challenge, we present an approach on how an appropriate tool may support users in building a textual DSML the bottom-up way. With it, we aim to support users that may not be experts in defining languages but are used to structured languages like programming languages.

2. Example

In the following, we present an example which points out the core features a text editor for bottom-up modelling should support. Figure 1 shows an instance of a simplified and slightly adapted variant of Fowler's state machine DSML [8]. Analogous to the lexical analysis in the field of compiler construction, we break the text into tokens to identify their basic meanings in form of their types. The different types of tokens are highlighted with different colours (except delimiters).

```
events   START , WORK , STOP

state  Idle
   START  =>  Active
state  Active
   WORK  =>  Active
   STOP  =>  Idle
```

Figure 1. Example model of a state machine (free text)

In an ideal world, one simply enters this text and the editor recognizes the tokens and their respective types as well as the intended hierarchical structure. To do this, we first extract four different token types which are only complete regarding this example but not in general:

- the blue *keywords* `events` and `state`,
- the black *identifiers* for naming a specific construct (here the events `START`, `WORK`, and `STOP` and the states `Idle` and `Active`),
- the violet *references* (for referring to events and states via their particular name) and finally
- the *delimiters* (namely `,` and `=>`)

After the recognition of tokens, the structure between them has to be extracted since it acts as basis for the later induced abstract syntax. At first place, we would expect a comma-separated list of events which is indicated by the keyword `events`. Afterwards, the states are defined whereas each state is declared by typing a line wrap followed by the keyword `state`. Each state contains an arbitrary number of transitions that are represented by the indented lines below every state declaration. A transition is composed by a reference to an event and a reference to a target state.

Simply because of her/his experiences in formatting source code according to the code's meaning, most software developers understand this structure without deeper explanations. Unfortunately, a computer does not have such experiences and consequently, it cannot identify this structure out of the box. To support the bottom up modelling approach, this structure and the different token types need to be recognized automatically as far as possible. Tokens are the elementary parts of a language and the structure describes how they are combined. Hence, identifying this structure is an essential precondition for the later derivation of the abstract syntax.

Figure 4 shows a suggestion how the example model's hierarchical structure may look like. Thereby, we distinguish between two types of containers. Statements (marked by a dashed border) are intended to represent domain elements like events, states and transitions. They consist of arbitrary tokens and any number of blocks. Blocks (surrounded by a solid border) are intended to constitute containers for statements. Their pendant on domain side are containers within domain elements that contain other domain elements.

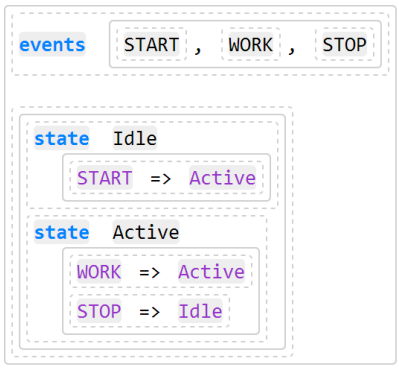

Figure 2. Example model of a state machine (structured text)

3. Requirements

During our research, we identified three requirements a textual editor should meet for supporting the bottom-up approach. They are formulated in a quite abstract manner because, at this place, we do not want to anticipate how the solution looks like.

First of all, the user experience of such an editor should feel like entering free text (**R1**). In more detail, it means that a user should not be hampered in her/his workflow when textually sketching a model. Otherwise, the simplicity benefit of the bottom-up modelling approach will be lowered and thus, the top-down procedure might be preferred.

Besides, developers are accustomed to the code editing support of modern IDEs. Especially features like syntax highlighting and content assist are popular and expected by the users. Therefore, these features should be provided as early as possible in the process of sketching textual models (**R2**). For this task, grammar inference [13] is not sufficient because appropriate tool support (primarily syntax highlighting and content assist) can only be supplied after a grammar was induced [14].

In order to not hamper the user's workflow (see **R1**), the extraction of tokens and structure mentioned in section 2 should be performed in an automatic way. In some cases, however, the user needs to directly influence the identification process (**R3**). The main reason would be that (s)he is not content with the produced result and thus, (s)he wants to manipulate it accordingly.

4. Solution approach

Owing to the different input method when modelling textually, tools need to support the user in a different way in comparison to the demonstration of graphical DSMLs. When working with diagrams, the primary interactions are mouse gestures that create, move, resize and connect shapes. Working with text, however, means occurring key events which are applied to the text at the cursor's current position. Usually, the character associated with the pressed key is inserted at this place. This is also true for projectional editors but they only allow for entering text at specific places within the document. Elsewhere inputted characters are simply ignored by the editor. Because of a better user experience, the input behaviour of projectional editors corresponds more and more to the one of free text editors [15].

Consequently, tools supporting textual bottom-up modelling should be able to accept key strokes as input. During input, the characters are directly analyzed and tokens as well as their particular types are identified. For texts written in a formal language, it is common practice to represent them as a tree structure. In available DSMLs and also general purpose programming languages, this tree structure primarily is an AST [1, 8]. However, when writing free textual models the abstract syntax is not yet defined. At this point, only the concrete syntax is given in form of tokens. Hence, we adopt the general idea and build up a so called concrete syntax tree (CST) out of these tokens. Beyond the identification of tokens, this is also performed automatically (as far as possible) while the user inputs text.

Motivated by requirement **R3**, we mainly adopt the projectional editing paradigm because then the entered text is directly available as a hierarchical structure. With it, the user can simply modify the tree's structure and its content which directly affects the intended meaning. This is important for a later step where the abstract syntax is inferred [16].

In the following, some prerequisites are described which need to be fulfilled. After that, we specify the building blocks of the CST and present a basic strategy for recognizing its overall structure.

4.1 Prerequisites

Textual bottom-up modelling requires a fundamental assumption on textual DSMLs. Otherwise, automatic recognition of particular structures within the entered character sequence is impossible. This assumption bases on the semantic of certain tokens for structuring text expected by users. Typical representatives for structure-bearing tokens are punctuation symbols, combinations of such symbols and whitespaces. Further on, we call them delimiter tokens. With them, there are several syntactic patterns which recur in many formal computer languages. For instance, elements are enumerated by means of commas or common elements are clasped using curly braces. In place of inserting curly braces for declaring blocks, syntactic indentation is an increasingly used pattern due to its reduction of syntactic noise (namely the braces) [8]. In doing so, a new block is defined by a line break and one further indentation. Such patterns should be simultaneously applied to the current CST while the respective tokens are identified.

4.2 Building blocks of the CST

As shown in the example and mentioned at the beginning of section 4, the main structure can be described as a tree which predominantly consists of tokens and containers. The root `Container` is stated by a `Document` which always represents the entry point of a CST (Figure 3). Both, tokens and containers, have a common base class `Cell` that provides some basic functionality (e.g. the possibility to re-parent a cell). Except of the root one, each cell is part of a parent container.

To determine a token's type, instances of the class `TokenType` need to be assigned to the particular token. This task is performed automatically in the background during the type recognition process (section 4.3). Since `TokenType` is abstract only sub classes can be used for instantiation. To provide a new token type developers need to implement such a sub class and register it at the document's `TypeRegistry`. Every type has to implement the `conforms()` method which states to a given text whether this text is valid or not. Currently, we distinguish the following types in form of separate sub classes (not shown in Figure 3).

- *Keyword*: a reserved word which often has a typing or annotating purposes (e.g. see `state` and `events` in the state machine example).

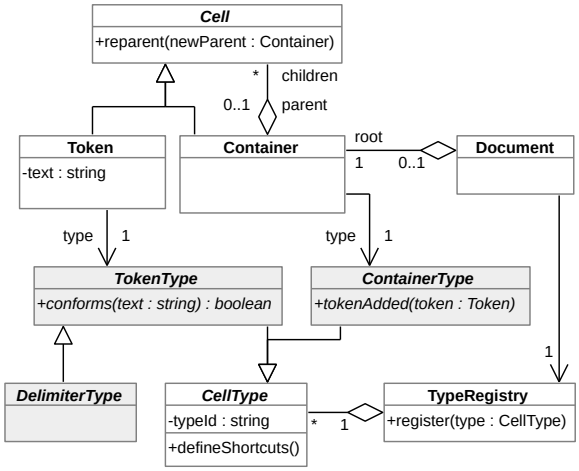

Figure 3. Meta model for concrete syntax tree

- *Identifier*: specifies a statement's unique name. Hence, each statement should only consist of one identifier token at a max.
- *Reference*: it refers to another statement via its name that was specified using an identifier. In contrast to tokens of other languages, identifiers and references already need to be distinguished within the CST to support bottom-up modelling. Otherwise, the abstract syntax inference engine cannot differentiate between both, leading to unexpected results.
- *String*, *integer* and *float literal*: indicate literal values, well-known from other languages.
- *Comment*: a not further considered note, but needed by users for documentation purposes.
- *Delimiter*: typical representatives are whitespaces, commas, assignment operators like "=" and ":=", comparison operators like "==" and "<", arithmetic operators like "+" and "-" etc.. Since delimiters often affect the CST's structure and thus take a special position, the particular sub class `DelimiterType` of `TokenType` is predefined for all delimiter types. Hence, delimiters can be further subdivided according to their impact on the tree structure. These subdivisions are listed in section 4.3 because they are part of the recognition strategy.

Analogous to tokens, a container's type is determined using sub classes of `ContainerType`. Every time a new token has been added to a container, the associated type is notified by invoking `tokenAdded()`. With it, the container type gets the opportunity to perform any further adaptions on the CST. By virtue of Friedman [9] and various programming language specifications [10, 12, 18], the following three container types should be distinguished.

- *Statement*: only may contain tokens, blocks and an expression because it is intended to represent a domain model element.
- *Block*: merely consist of statements. It is always part of a statement and hence, constitutes a container for sub-elements of the wrapping statement's domain element. In Figure 2, for instance, the statement representing the `Active` state includes a block which contains two other statements. These ones are intended to represent transitions within the domain.
- *Expression:* some kind of calculation rules, i.e. rather similar to statements. Hence, an expression may be composed of tokens. In addition, it may also contain further expressions which would break with the semantic of a statement because they are not self-containable. Consequently, it is reasonable to treat expressions as a further container type.

Both, token and container types have a common base class `CellType` and are managed by a `TypeRegistry`. With this base class, each type is able to define custom shortcuts for specific operations on the CST (e.g. constituting a token's type or creating a block). This feature addresses requirement **R3**. Moreover, the abstract classes `TokenType`, `DelimiterType` and `ContainerType` (highlighted grey) are intended for subclassing to extend the CST's meta model about additional cell types.

4.3 Recognition strategy

As mentioned earlier, the recognition of tokens, their types and the hierarchical structure of the text is directly performed after the user has pressed a key. Accordingly, the user can simply input text as how (s)he would do with free text editors. By this means, requirement **R1** is fulfilled.

Before extracting any structural information, the tokens need to be identified. In order to detect a token's type, regular expressions can be applied to the entered text (within an implemented

`conforms()` method). This is sufficient for string, integer and float literals. Unfortunately, distinguishing between the token types identifier, reference and keyword is not as easy since they are valid in accordance to the same regular expression. Thus, user interaction is required to determine the token type in those ambiguous cases. It can be provided via shortcuts defined within an overridden `defineShortcuts()` method. According to the particular types, each token can be highlighted which solves one part of requirement **R2**. The content assist part can be addressed by providing code completion for reference tokens. Permitted values are, for instance, identifiers declared within the text entered so far.

The structure and hence the containers with their respective types are automatically created when delimiters are recognized. Delimiters can be categorized according to their impact on the CST's structure.

- *Token separator*: it is the default category of each delimiter which is not classified any further. Entering such a separator will simply lead to a new token within the current statement. (typical representative: space character)
- *Statement separator*: such delimiters separate sequenced statements from each other. We assume that different representatives of this category have a different meaning (similar to the assumption stated in section 4.1). Consequently, the behaviour when entering such a separator depends on the previous separator used in the current block. If there is a separator and it is equal to the one entered then a new statement is added to the block. Otherwise a new block is inserted into the current statement. For instance, imagine a block with three statements that are separated by means of a comma. Then if a line break is added to the last statement, this statement is extended by a block at this place. In case one wants to separate child statements with the same delimiter as used for the parent statements (e.g., it happens in the example in Figure 2 for separating states and their contained transitions) those delimiters need to be distinguished in a different way (e.g., by simultaneously pressing a modifier key like Ctrl or Alt while entering the separator). Owing to this feature, requirement **R3** is covered. Typical representatives are commas, semicolons and line breaks.
- *Opening brace*: when entering an opening brace, a new block is inserted in every case. Beyond that, the corresponding closing brace is added automatically as well which is a familiar feature of modern IDEs and thus addresses requirement **R2**. In Figure 4, there are four examples for opening braces, but the second round one is empty and hence no container is visualized. Typical representatives are "(", "[" and "{".
- *Operator*: entering an operator leads to a new expression that is added to the current statement or an already existing expression. Currently, we only support binary operators and so infix expressions and also do not consider the different priorities of various operators. How an exemplary complex expression structure may look like can be seen in the `return` statement of Figure 4. Typical representatives are ".", "+", "-", "*" and "/".

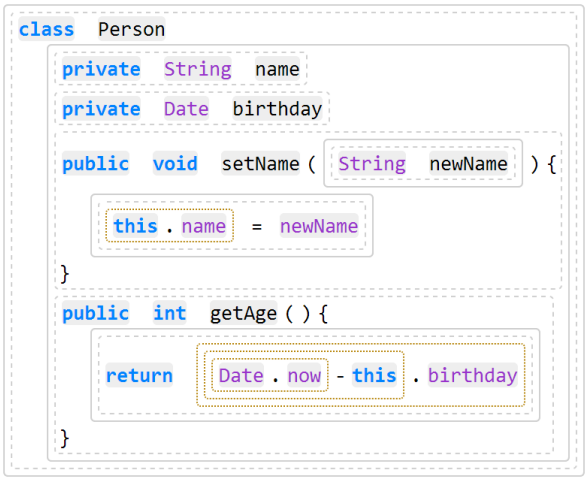

Figure 4. Example of a sketched programming language snippet (similar but not identical to Java [10])

The recognition of where the current token ends and a new token starts is performed comparably to the lexical analysis of compilers [1]. By doing this, the end of a token is encountered if the previous complying regular expression is not valid any more.

5. Proof of concept

For evaluating the presented approach, we developed a prototypical editor using HTML5 technologies. The reason for choosing HTML5 is that modern browsers may enable users to modify a web page in the manner of WYSIWYG by setting an element's `contenteditable` attribute to true [11]. During the input, the underlying DOM is directly updated which we can interpret and utilize as CST. The implemented editor offers each feature described in section 4. For this reason, it addresses all requirements stated in section 3, especially supporting some user interventions to cover the third requirement in greater depth.

One problem we have tackled during the development, is the implementation of proper syntax highlighting of keywords, identifiers and references. Unfortunately, it cannot be done automatically and thus, the user has to indicate the type of such tokens using shortcuts (requirement **R3**). By default, each token is an identifier. However, one time a token is classified as a keyword this information can be reused for each further extracted token. For instance, it is the case for the keyword `state` in Figure 2. Therein, the keyword was defined manually for state `Idle` and automatically recognized for state `Active`.

Again, requirement **R3** is addressed by the user's ability to select successive tokens and wrap them in a new container by means of a shortcut. The specific shortcut determines the type of the container (predominantly block or statement). In addition, a container's type can be changed afterwards if desired.

Beyond the quite simple example presented in section 2, with the editor it is also possible to sketch sophisticated textual models. For that purpose, Figure 4 shows a class in a fictional object-oriented programming language (similar to Java) created with the editor. The most noticeable parts compared to the above example are references to elements that are not declared within the same model, the usage of two different brace types and the presence of expressions (marked by dotted orange borders).

6. Conclusion and future challenges

As mentioned above, our research is in an early state and thus, the presented approach raises no claims of being complete. However, it already shows a promising solution of how an editor for enabling textual bottom-up modelling may look like and function. We already gathered some successful validation results concerning the practicability by means of a prototypical implementation (section 5). With it, we recreated several textual models by using the syntax of existing DSMLs (e.g., a state machine in Figure 2) and even general purpose programming languages (e.g. a class described by a language similar to Java in Figure 4).

There is still plenty of research to be done in the field of textual bottom-up (meta) modelling. While different people worked with

our editor, we found out that it would be expedient that the editor's behaviour is highly customizable. The reason for that lies in the different habits and preferences of various users regarding the design of languages. For instance, a user would like to use colons for assignments, whereas another one wants to utilize them as division operators. One further issue which could be covered by means of configurability is the prioritisation of operators. Nevertheless, it still needs to be discovered how the structure has to be modified according to the particular operator's priority.

Another major task is exploring how the editor's user experience can be further improved (requirement **R2**). That principally can be achieved by extending the content assist support, e.g., for recurring statements. In Figure 4, two fields are declared which are prefaced with the modifier `private`. So, if the user inputs a "p" in the third line the editor could provide the proposal "private" based on the information gained from the statement before. It might also be possible to provide content assist for the structure of whole statements (in the form of templates). Instead of only inserting "private", a reference and an identifier placeholder can be additionally inserted. To support content assist for more than one type of statement (e.g., in the given class beyond fields there are methods as well), those types have to be identified first. In the graphical bottom-up domain the authors of [2] already proposed a solution to derive the syntactic building blocks of diagrams which could be evaluated and (if appropriate) adapted in respect of textual models.

What we did not address until now is how an abstract syntax is derived from textual example models. According to the presented approach, however, the user needs to ensure that the CST is structured in a way that fits the semantic model in mind. In future research, a mapping has to be determined that specifies the method of inferring an abstract syntax from a given set of CSTs. Among other things, it needs to be explored how meaningful names can be recognized for derived constructs of the abstract syntax.

Acknowledgments

This research paper was authored in the context of the project "Kompetenzzentrum für praktisches Prozess- und Qualitätsmanagement" (KpPQ) funded by "Europäischer Fonds für regionale Entwicklung" (EFRE). So, we thank this institution which has kindly facilitated our work.

References

[1] Aho, A. V., Lam, M.S., Sethi, R. and Ullman, J.D. 2008. *Compiler*. Pearson Studium.

[2] Anaby-Tavor, A., Amid, D., Fisher, A., Ossher, H., Bellamy, R., Callery, M., Desmond, M., Krasikov, S., Roth, T., Simmonds, I. and de Vries, J. 2009. An algorithm for identifying the abstract syntax of graph-based diagrams. *IEEE Symposium on Visual Languages and Human-Centric Computing* (Corvallis, OR, Sep. 2009), 193–196.

[3] Cho, H. 2011. A demonstration-based approach for designing domain-specific modeling languages. *Proceedings of SPLASH 2011* (New York, NY, 2011), 51–54.

[4] Cho, H. 2013. *A Demonstration-Based Approach for Domain-Specific Modeling Language Creation*. University of Alabama.

[5] Clark, T., Sammut, P. and Willans, J. 2008. Applied metamodelling: a foundation for language driven development. *CETEVA*. (2008).

[6] Desmond, M., Ossher, H., Simmonds, I., Amid, D., Anaby-Tavor, A., Callery, M. and Krasikov, S. 2010. Towards smart office tools. *SPLASH 2010 Workshop on Flexible Modeling Tools* (Reno, Nevada, 2010).

[7] Dmitriev, S. 2004. Language oriented programming: The next programming paradigm. *JetBrains onBoard*. (2004).

[8] Fowler, M. 2011. *Domain-Specific Languages*. Addison-Wesley.

[9] Friedman, D.P. and Wand, M. 2008. *Essentials of Programming Languages*. MIT Press.

[10] Gosling, J., Joy, B., Steele, G. and Bracha, G. 2005. *The Java Language Specification*. Addison-Wesley.

[11] HTML5 - Editing: 2013. *http://www.w3.org/TR/2008/WD-html5-20080610/editing.html*. Accessed: 2013-08-09.

[12] Kernighan, B.W. and Ritchie, D.M. 1988. *The C programming Language*. Prentice Hall.

[13] King-Sun, F. and Booth, T.L. 1986. Grammatical Inference: Introduction and Survey - Part I. *IEEE transactions on pattern analysis and machine intelligence*. 8, 3 (Mar. 1986), 343–359.

[14] Mernik, M., Hrncic, D., Bryant, B.R., Sprague, A.P., Gray, J., Liu, Q. and Javed, F. 2009. Grammar inference algorithms and applications in software engineering. *Proceedings of the 9th International Colloquium on Grammatical Inference* (2009), 1–7.

[15] MPS public roadmap: 2013. *http://confluence.jetbrains.com/display/MPS/MPS+public+roadmap*. Accessed: 2013-08-08.

[16] Roth, B., Jahn, M. and Jablonski, S. 2013. A Method for Directly Deriving a Concise Meta Model from Example Models. *Proceedings of PATTERNS 2013* (2013), 52–58.

[17] Sánchez-Cuadrado, J., Lara, J. De and Guerra, E. 2012. Bottom-Up Meta-Modelling: An Interactive Approach. *Proceedings of the 15th International Conference on MODELS* (2012), 3–19.

[18] The Python Language Reference: 2013. *http://docs.python.org/3/reference/index.html*. Accessed: 2013-07-26.