

Integrating Content Assist into Textual Modelling Editors

Markus Scheidgen*
scheidge@informatik.hu-berlin.de

Abstract: Intelligent, context sensitive content assist (also known as code completion) plays an important role in the effectiveness of model editors. This is not only true for textual language notations, but also for graphical notations that often contain a significant amount of textual elements. This paper presents techniques to describe content assists for meta-model based textual model editors. We show that these techniques help to automate the development of editors with content assist, a process that requires extensive manual work otherwise.

1 Introduction

Modern integrated development environments for programming languages, have accustomed us to editors with capabilities that increase productivity by magnitudes. Extensive knowledge about language syntax and semantics programmed into these editors, allows them to offer context sensitive assistance to the editor users by presenting them with a list of meaningful continuations at the current cursor position. This is known as *content assist* or *code completion*.

Unfortunately, development of such editors is extensive and therefore has only been done for popular programming languages like Java. New editor development frameworks for domain specific modelling languages allow to describe language notations efficiently and generate feature rich editors from these descriptions automatically. This possibly facilitates the same tool quality for textual modelling languages with less development efforts.

While existing frameworks for textual modelling notations succeed in offering basic model editing capabilities, like transforming input text into models and vice versa, they struggle to achieve some of the advanced editing features, especially content assist. This paper introduces high level description techniques for content assist. Such descriptions can be used to generate textual model editors with content assist automatically. Our work focuses on content assist for keywords and named references.

Section 2 gives an introduction to textual modelling and textual editing frameworks. With this knowledge we present content assist techniques in section 3. In section 4, we report about experiences in realising content assist in OCL [Obj06] and ecore [BSM⁺03] editors, which we developed as case-studies, using the textual editing framework TEF [tef]. We close with related work and conclusions in sections 5 and 6.

*This work is part of the Graduiertenkolleg METRIK, founded by the Deutsche Forschungs Gemeinschaft.

2 Background – Textual Modelling

Textual modelling, as opposed to graphical modelling, uses text to represent models. Textual model editors allow users to write models in a language specific syntax. In this paper we concentrate on languages with an abstract syntax defined as a meta-model and concrete notations defined separate from this meta-model. A notation, suitable for textual modelling, consists of a context-free grammar, and a mapping between that grammar and the language meta-model. The grammar defines a set of syntactical correct strings. The mapping describes what syntactical correct string represents which model. Frameworks for developing textual modelling editors provide template languages to define textual notations and can generate textual model editors from such notations. More detailed information can be found in: [AP04], [MFF⁺06], and [WK06].

The background parsing process. Most existing textual model editors use *background parsing*. Background parsing consists of the three steps: (1) the user edits text using a normal text editor, (2) the inserted text is parsed according to a given grammar, (3) a model is created from the resulting parse-tree based on a given meta-model and grammar to meta-model mapping. This process is repeated continuously to give the impression that the user edits the model directly.

Creating models from parse-trees. Figure 1 shows a meta-model for a simple expression language in the top-right corner. Below this meta-model we see a model, an instance of the meta-model, that represents the expression $foo(n) = (n + 2) * n + 1$. On the left side of this figure we see a grammar that could be used to define a notation for that expression language. The rules in this context-free grammar can be used to create the parse-tree below, which is a parse-tree for the string $foo(n) = (n + 2) * n + 1$.

The example parse-tree and model are very similar. Basically, we can map symbols and terminals to objects and their attributes, and we can map child-of relations between nodes to corresponding links. However, there is one important difference. There are some links in the model that are not represented in the parse-tree directly. These are links between instances of *VariableExpression* and *Variable*. The fundamental difference between meta-models and context-free grammars is that meta-models describe graphs, while grammars describe trees. Therefore, this link (which causes a circle in the model graph) has to be represented indirectly within the parse-tree. This is a typical problem for notations defined with context-free grammars: they usually use some form of identifier to describe a reference between a variable usage and its definition of that variable. Due to these identifier, creating a model from a parse-trees has to include *identifier resolution*.

The reason for content assist. Identifier resolution is also the reason for content assist: editors struggle to resolve references, users have troubles writing references. Content assist helps users by providing lists of possible resolvable identifiers. This means, for example, to provide a list of all variable names, when code-assist is requested at a position that would syntactically allow a named reference to a variable.

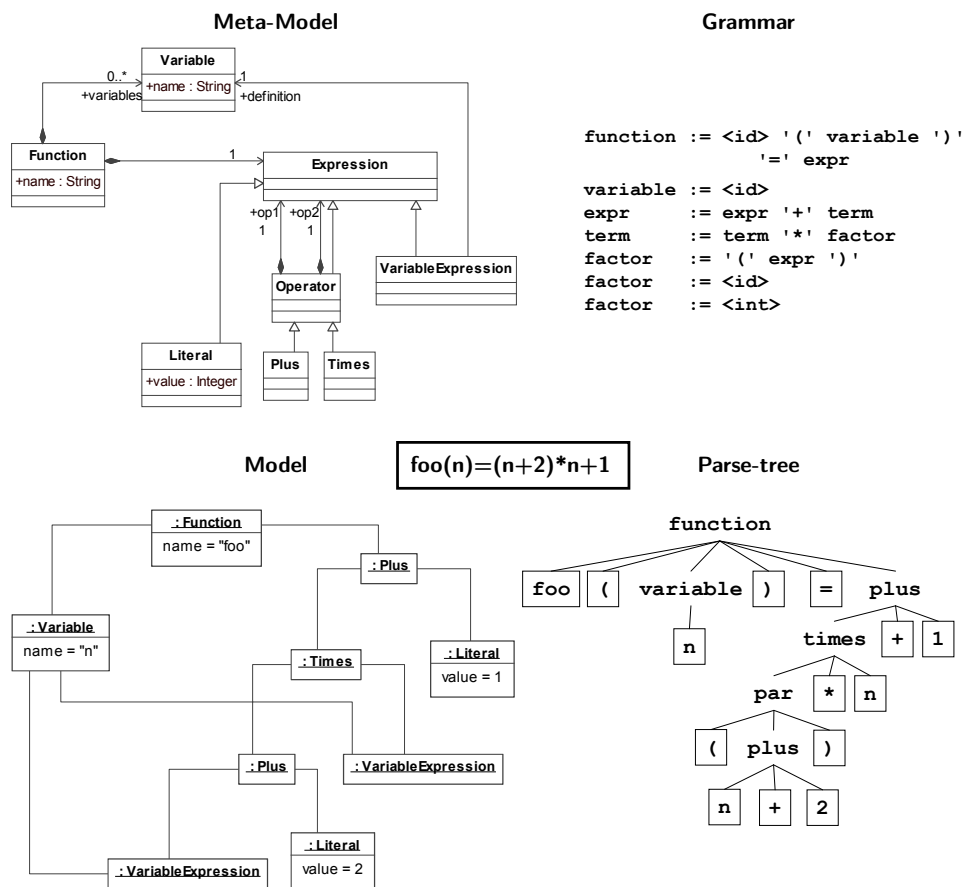


Figure 1: The differences between meta-models and grammars exemplified based on a simple expression language.

3 Content Assist

We introduce the notation of a *content assist type*. The engineer, who develops the textual model editor, can define multiple content assist types as part of a notation definition. The editor uses these content assist types to offer content assist. A content assist type defines a *syntactical context* and instructions to collect *content assist proposals*. When the user of the editor request content assist, the editor determines for each content assist type if the current cursor position is located in the syntactical context that is defined in the content assist type. Thereby, the editor selects a set of *active* content assist types. Content assist proposals are collected for all active content assist types based on the instructions given in these content assist types. The current model, text, and cursor position is used as input for these instructions. All the collected proposals are finally presented to the user.


```

definitions
datatype Symbol
array Symbol[] Rule
struct SymbolCA { symbol: Symbol }
struct SingleReductionCA { rule: Rule, symbol: Symbol, suffix : array Symbol[] }
array MultipleReductionCA SingleReductionCA[0..n]

boolean shift(Symbol) // shifts the symbol on the parse stack if possible
boolean reduce(Rule) // reduces the parse stack using the given rule if there is
// a follow up symbol allowing the reduction
boolean reduce(Rule, Symbol) // reduces the parse stack using the given rule if
// possible using the given follow up symbol

algorithm
parse the document using the notation's grammar rules and LR-syntax analysis
stop parsing at the current cursor position
switch type of ca
  SymbolCA:
    return shift(ca.symbol)
  SingleReductionCA:
    if not shift(ca.symbol) then return false
    for symbol in ca.suffix do
      if not shift(symbol) return false
    if reduce(ca.rule) return true
  MultiReductionCA:
    for i = 0; i < ca.length; i++ do
      if not shift(ca[i].symbol) return false
      for symbol in ca[i].suffix do
        if not shift(symbol) return false
      if (i+1 < ca.length)
        if not reduce(ca[i].rule, ca[i+1].symbol) return false
      else
        if not reduce(ca[i].rule) return false
    return true;

```

Figure 3: An algorithm that determines if an input content assist (ca) is active in pseudo-code.

How can we determine if a syntactical context is active or not? We use LR-syntax analysis, which has two properties that are important for this. Firstly, once a symbol is shifted onto the parse-stack, there cannot be a syntax error in front of that symbol. Secondly, reductions only happen on the top of the parse-stack. We use the following algorithm: the document is LR-parsed up to the cursor position. Now, we emulate continued parsing as if the text following the cursor position is written according to the syntactical context. If that is possible, the context is active. For symbol content assists, this means we try to shift the symbol onto the parse-stack. If that is possible, the context is active. For a single reduction content assist, we shift the symbol, then shift the symbols in the rule suffix, and then try to reduce with the context's grammar rule. If this is all possible, the context is active. For a multi reduction content assist, we do as in a single reduction content assist, but after reduction, we try to shift the suffix of the next rule, reduce with this rule, then continue with the next rule, etc. If we can do so for all parts of the multi reduction content assist, it is active. Figure 3 shows pseudo-code for this algorithm.

3.2 Collecting Content Assist Proposals

Each content assist type includes instructions to collect proposals. The proposals can be collected from the following information: the validity of the syntactical context of the content assist, the parse-tree created during analysing the syntactical context, and the current model. We distinguish between three different *quality levels* for proposals.

For the lowest proposal quality, we only use the syntactical context without any additional instructions. From the syntactical context, we know which type of element can be inserted at the cursor position and we simply offer all instances of the according type. Take the OCL example in figure 2 and the single reduction assist number 2: we propose all identifiers that reference an operation. Therefore, we collect all operations in the given model. The problem is that all the proposals are valid based on the abstract syntax of the language, but not necessarily its static semantics. In the example, we are not constraining the set of operations to those allowed based on the expression type that the operation is called upon.

For a higher quality level, we again use the syntactical context, but now also allow additional constraints based on the parse-tree. Starting at the node located in the syntactical context, one can visit all the *containers* of the syntactical context by navigating the parse-tree towards its root. Whereby, *container* refers to containment as defined by *composition* in the meta-model. Take the multi reduction content assist in figure 2: navigating from the variable access to the collection operation call (two containment relations out, respectively two parse-tree nodes up), we can access the variables of the collection operation call, and these are the variables we want to propose.

For the highest proposal quality, we allow proposal constraints based on parse-tree and model. Previously, using only the syntactical context, we proposed all operations in the first single reduction content assists of the OCL example in figure 2. But based on parse-tree and model, we can determine the expression that the operation is called on. We can determine the expression's type and all operations allowed for this type. It is now possible to constrain the set of operations to those allowed by OCL's operation call semantics.

4 Realisation and Case-studies

We created the *Textual Editing Framework* (TEF) [tef], a programming framework based on eclipse, that allows to create textual model editors based on the background parsing strategy. TEF uses a template language (similar to the one used in [JBK06]) to define textual notations based on a meta-model.

TEF allows several ways to define content assist types as part of templates, all with different levels of automation. You can define syntactical contexts manually, using data-structures similar to those in figure 3. TEF also provides syntactical contexts automatically based on templates: a symbol content assist type (*keyword assist*) is created for each keyword in the notation, and a default single reduction content assist type (*reference assist*) is created for each reference in the notation (single reduction is enough to distinguish different reference types in most language that we examined).

Proposal collection instructions are automatically generated for keyword and reference assists. For keyword assists there is, of course, always only one proposal that is the keyword itself. For reference assists the syntactical context is used to provide all instances of the according reference type as proposals. However, TEF defines callback methods that allow editor engineers to constrain the automatic reference assist proposals with arbitrary Java code. Giving parse-tree and model as input, these methods can be used to realise the more advanced proposal quality levels.

We realised model editors for several example toy languages, like the expression language in figure 1, and we created more significant editors for ecore models and OCL constraints. These case-studies showed that we can create high quality content assist for such language, while we only have to manually provide syntactical contexts in some seldom cases, and manual proposal collection instructions are only necessary to cover aspects of static semantics. We already discussed some representative examples from the OCL editor.

5 Related Work

Content assist has a long history, starting from simple language independent approaches such as *hippo completion*. Hippo completion proposes any word in a text document regardless of the syntactical context of the cursor. State of the art content assist is based on the languages syntax and static semantics, in its full potential firstly introduced in the *intelliJ* Java IDE.

With model-driven development and domain specific modelling languages, frameworks were introduced that allowed to create textual model editors more efficiently. Example frameworks are xText [oAW] and TCS [JBK06]; there are other research projects like [Kle07] or [MFF⁺06]. However, realising code-completion in these frameworks, if possible at all, still requires manual work.

Besides background parsing, editors can use the *model view controller pattern*. Such editors don't allow users to type arbitrary text, but to use predefined commands to insert language construct instances. Content assist plays an integral role in these editors, since models are edited by selecting commands from a list determined by the current syntactical context. But content assist also works intrinsically different, because the text is not edited by the user, but created by the editor. Frameworks for creating model view controller editors are *intentional programming* [Sim95] and the *meta programming system* [Dmi04].

6 Conclusions

We presented techniques to integrate content assist into meta-model based textual model editors. We showed that it is possible to describe content assists for keywords and references and that editors with content assist can be generated from these descriptions automatically. Beyond that, it is possible to even generate the content assist descriptions for

a lower quality of content assist based on notation descriptions, which have to be written anyway. Only for higher quality content assist, manual implementation is necessary to constrain content assist proposals to those allowed by language specific static semantics.

This paper mainly dealt with content assist for keywords or references. A interesting subject for future work is content assists for arbitrary syntactical constructs. These could allow users to insert whole instances of constructs (such as if or loop statements in programming languages) opposed to assists for keywords or named references. Another open point are descriptions to constrain content assist proposals. It should not be necessary for editor engineers to create these, if the information about language constraints is already part of the language's meta-model. It should be possible to use OCL-constraints in the meta-model to derive constraints for content assist proposals automatically.

References

- [AP04] Marcus Alanen and Ivan Porres. A Relation between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, TUCS, 2004.
- [BSM⁺03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework (The Eclipse Series)*. Addison-Wesley Professional, August 2003.
- [Dmi04] Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm. *onBoard*, (1), November 2004.
- [JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS:: a DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254, New York, NY, USA, 2006. ACM Press.
- [Kle07] Anneke Kleppe. Towards the Generation of a Text-Based IDE from a Language Meta-model. In *Proceedings of the Third European Conference, ECMDA-FA 2007*, pages pp. 114–129, 2007.
- [MFF⁺06] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michael Hassenforder, Rémi Schneckenburger, Sébantien Gérard, and Jean-Marc Jézéquel. Model-Driven Analysis and Synthesis of Concrete Syntax. In *Proceedings of the 9th International Conference, MoDELS 2006*, pages pp. 98–110, 2006.
- [oAW] openArchitectureWare. See <http://www.openarchitectureware.org>.
- [Obj06] Object Management Group. *Object Constraint Language Specification, version 2.0*, May 2006.
- [Sim95] Charles Simonyi. The death of computer languages, the birth of Intentional Programming. Technical report, Microsoft Research, 1995.
- [tef] Textual Editing Framework (TEF). See <http://www.informatik.hu-berlin.de/sam/meta-tools/tef>.
- [WK06] Manuel Wimmer and Gerhard Kramler. Bridging Grammarware and Modelware. In *Satellite Events at the MoDELS 2005 Conference*, pages 159–168, 2006.