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.

The remainder of this paper is structured as follows: 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 TEF, a framework for textual model editors that supports content assist. 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. Textual modelling can be applied to a whole language, i.e. the whole model is represented with a single cohesive text. In this case an according textual model editor is a standalone tool that is used to create and edit models. Textual modelling can also be applied to parts of a language, where the other parts are represented otherwise, e.g. graphically. In this case, textual model editors are integrated into another (e.g. graphical) editor. Examples for this are complex textual statements embedded into a graphical language, such as OCL expressions in UML diagrams. We will not further distinguish these types of textual model editing, because an embedded textual model editor is simply a model editor for a sub-language and both kinds of editors are realised in the same way.

Furthermore, categorising a language into modelling or programming language only refers to the pragmatics and semantics of the language and has nothing to do with how instances of this language are edited. To avoid confusion, we will stick to the term textual modelling language, but the presented techniques can also be applied to programming languages.

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 identifies the model corresponding to a syntactically correct string. Frameworks for developing textual modelling editors provide notation definition languages to define textual notations and can generate textual model editors from such notations. More detailed information on using grammars for notating models 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-left 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 right 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

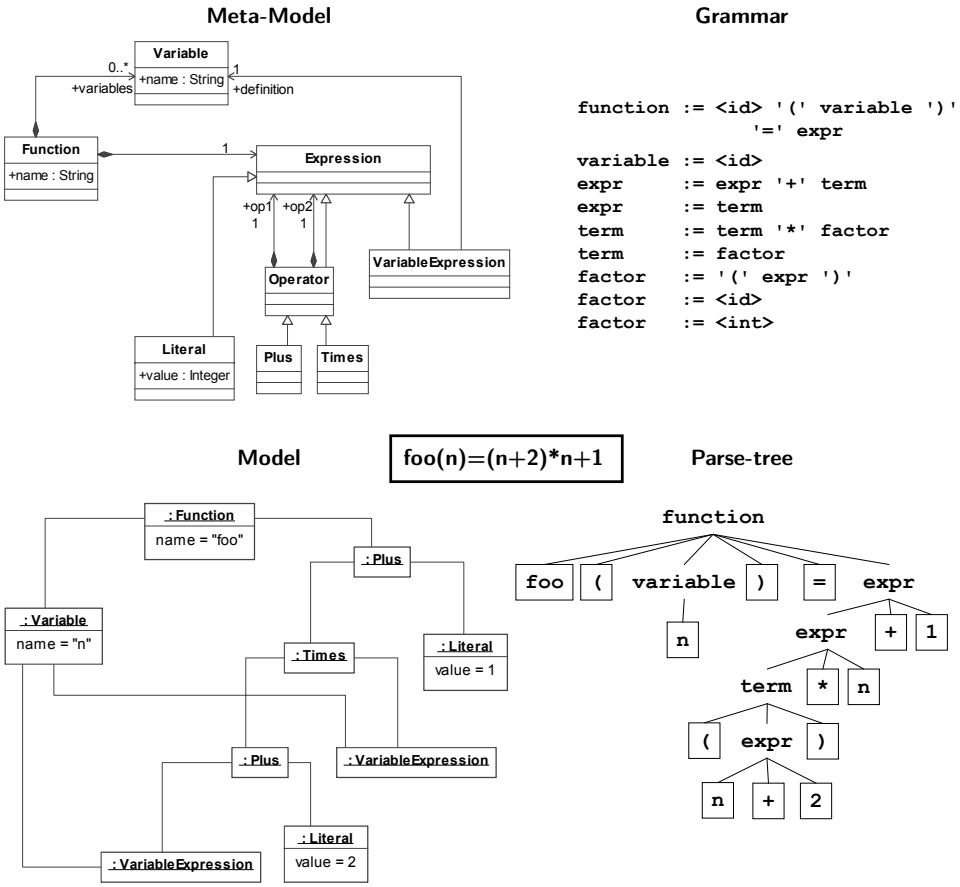

Figure 1: The differences between meta-models and grammars examplified based on a simple expression language.

in the model that are not represented in the parse-tree directly. These are links between instances of *VariableExpression* and *Varibale*. 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 references, creating a model from a parse-trees has to include *identifier resolution*.

Content assist in textual model editors. Content assist in general describes the text editor capability to provide a list of possible text fragments that form reasonable continuations for the text under the current cursor position.

References are the most important language constructs that content assist provides help for. One of the major problem that users have with editing textual models is to select identifiers and to spell them correctly. Content assist helps with references by providing a list of reasonable identifiers for the current cursor position; users can choose identifiers from that list instead of creating the according reference manually. The proposed identifiers are taken from the model that is created during a background parsing process. We call this intelligent, model-aware form of content assist *reference assist*.

Content assist can also help with simpler language constructs such as keywords or special characters. If a user is uncertain about what keywords or special characters are syntactically possible at the current cursor position, content assist provides a list of possible terminals to choose from. We call this simpler, only syntax dependent form of content assist *keyword assist*.

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 requests content assist, the editor determines for each content assist type if the current cursor position is located within 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.

3.1 Determining the Syntactical Context

The following pieces of information about the language are known to the editor engineer and are also programmed into the editor: the language's meta-model, the notation's grammar, and the mapping between grammar and meta-model. Based on this information the editor engineer has to define possible syntactical contexts. The editor furthermore knows the current text, a model based on a possibly earlier version of the text, and the cursor position. Based on this information, the editor has to determine if a syntactical context is active or not.

What is a syntactical context? A syntactical context is a specific point within the syntax of a language construct. A syntactical context can be used by an editor to determine all those positions within a text that describe this point in the instances of the according language construct. We say that these positions are *located* in this syntactical context. A syntactical context is active if the current cursor position is located in this context.

example grammar except from a simplyfied OCL grammar:

```
collection_op_call = expr "->" IDENTIFIER "(" variables "|" expr ")"
op_call = expr "." IDENTIFIER "(" arguments ")"
variable_access = IDENTIFIER
expr = collection_op_call | op_call | variable_access
```

syntactical context for a symbol content assist: IDENTIFIER

three syntactical contexts for single reduction content assists:
(the underline denotes the symbol that defines the syntactical context)

symbol suffix

1. `collection_op_call = expr "->" IDENTIFIER "(" variables "|" expr )"`

rule

2. `op_call = expr "." IDENTIFIER "(" arguments ")"`
3. `variable_access = expr "." IDENTIFIER`

syntactical context for a multiple reduction content assist:
(the underline denotes the position where the next rule is used, the last underlined symbol defines the syntactical context)

```
[ collection_op_call = expr "->" IDENTIFIER "(" variables "|" expr ")",
  expr = variable_access,
  variable_access = IDENTIFIER ]
```

Figure 2: Some examples for syntactical contexts.

How can we define a syntactical context? We distinguish between syntactical contexts of three complexity levels. The simplest syntactical context is defined by a single terminal or non-terminal symbol. An assist using such a context is called *symbol content assist*. A more complex syntactical context is defined by a symbol, used within a specific grammar rule. An assist using such a context is called *single reduction content assist*. *Multiple reduction content assists* use a symbol within a specific grammar rule, which again is used within a specific grammar rule, and so on. Syntactical contexts can be defined using the data structures: `SymbolCA`, `SingleReductionCA`, and `MultipleReductionCA` (in the top of figure 3). Figure 2 shows a few examples for syntactical contexts.

How can we determine if a syntactical context is active or not? We use LR-syntax analysis to determine whether a syntactical context is active. We emulate continued parsing as if the text following the cursor position is written according to the syntactical context. If the continued parsing is possible, the context is active; if the continued parsing causes a parse error, the context is not 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.

```
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.sufix 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].sufix 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.

3.2 Collecting Content Assist Proposals

Each content assist type includes instructions to collect proposals. The proposals have to 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. Thereby, *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, respectivly 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

The Textual Editing Framework and Content Assist 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 editors are based on already existing EMF meta-models. TEF provides a EBNF-like language, that allows to define a language's textual notation. Such a textual notation definition contains a context-free grammar and binds this grammar to a meta-model. Meta-model elements can be assigned to grammar elements: grammar rules can be assigned to meta-model classes or data-types; elements of a rule's right-hand side can be assigned to attributes and references. This information allows TEF editors to continuously create EMF models for the edited text. A TEF editor defined by such textual notation definitions already comprise a serious of editor features, like syntax-highlighting based on the used terminals or an outline-view that presents the currently edited EMF model.

TEF editors also provide basic content assist based on a textual notations definition only. The automatically generated content assist comprises *keyword assists* and *reference assists*. Keyword assists are generated for each keyword or special character in the notation. They use a symbol content assist type to define a syntactical context, and their proposal collection always provides the terminal itself as the only proposal. Reference assists are generated for each reference binding in the textual notation definition that refers to a meta-model reference that defines a non-containment reference. These reference assists use a single reduction content assist type to define a syntactical context, and their proposal collection provides a list of names from all those model elements that have the meta-type of the according reference.

There are two ways to customize the automatically generated content assist types. Firstly, a language engineer can implement callbacks that alter the proposal collection behaviour of the automatically generated reference assists. Giving parse-tree and model as input, these

methods can be used to realise the more advanced proposal quality levels. Secondly, a language engineer can additionally define own content assist types. Here, the engineer has to implement certain interfaces to define a syntactical context using Java data-structures similar to those in figure 3 and he needs to program a proposal collection.

Content Assist for OCL We realised model editors for several example toy languages, like the expression language in figure 1, and we created more significant editors for EMF's meta-modelling language ecore [BSM+03] models and OCL [Obj06] constraints.

The TEF generated OCL editor, for example, automatically provides a content assist for each operator. Since these keyword assist are defined for a certain syntactical context, operators are only proposed if this operator is syntactically allowed at the according position. However, these automatically generated assists only reflect OCL's syntax, and operators are proposed even if they are not allowed semantically, e.g. not allowed based on operand types.

OCL contains five kinds of references: references to local variables, i.e. *self*, operation parameters, and variables defined in *let*-statements; references to the properties of model elements; references to the operations of model elements; references to types; and references to operations of OCL's collection library. Content assists for all these reference kinds could be generated with TEF. However, the proposal collections had to be manually altered for two reasons. Firstly, most references do not reference elements within the OCL model, but within the model that the OCL is written for. Secondly, we only wanted to allow proposal that are semantically valid, e.g. only propose properties and operations defined in the according type. The first alteration was necessary, because TEF's automatically generated reference assists only rely on the edited model, and not on some external model. The second alteration is always necessary if the desired content assists have to reflect the language's static semantics.

Creating the OCL editor showed lead to the general conclusion that even though TEF provides reasonable syntactical contexts automatically, and therefore relieves the language engineer from navigating though abstract syntax trees, etc., there is still lots of manual programming necessary, if content assist has to be restricted to semantically reasonable proposals.

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 and became popular through the eclipse *Java Development Tools* (JDT). In JDT, for example, the content assist types for the different references in Java are realised as pieces of Java code. Each of this assist types triggers parsing of the document, analysis parser and parse tree to determine if its active and collects proposals from the

least recently created background Java model. This approach results in multiple instances of similar code, because many (but not all) content assist types can be realised using the concepts in this paper.

With model-driven development and domain specific modelling languages, frameworks where introduced that allowed to create textual model editors more efficiently. Example frameworks are TCS [JBK06], TCSSL [MFF+06], MontiCore [KRV07], or [Kle07]. However, realising code-completion in these frameworks, if possible at all, still requires manual work. Two frameworks provide support for content assists: Safari [CDF+06] and xText [oAW].

Safari is a framework to create tooling for context-free grammar-based textual domain specific languages. Safari allows manual implementation of content assist. This means that the editor engineer has to realises each content assist type as a piece of Java code, which takes an AST of the document and the current cursor position as input and produces a collection of proposals as output. It provides no automatic means for determining if a syntactical context is active or proposal collection.

The xText framework allows to define textual languages with context-free grammars. It creates a meta-model, and grammar-to-meta-model mapping from a language defining grammar, and provides background-parsing text editor based on this meta-model automatically. The xText framework provides automatic reference assists similar to those of TEF, and allows to manually implement content assist. To manually implement content assists the editor engineer has to provide pieces of Java code that take a *grammar-element* and the model that the edited text represents as input and provide a collection of proposals as output. The given grammar-element represents the point in the grammar that reflects the current cursor position. This information allows to realise symbol and single reduction content assists easily. However, there is no support to realise multiple reduction content assists.

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* (MPS) [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. An interesting subject for future work is content assist 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.

[CDF+06] Philippe Charles, Julian Dolby, Robert M. Fuhrer, Jr. Stanley M. Sutton, and Mandana Vaziri. SAFARI: a meta-tooling framework for generating language-specific IDE's. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 722–723, New York, NY, USA, 2006. ACM.

[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 Metamodel. In *Proceedings of the Third European Conference, ECMDA-FA 2007*, pages pp. 114–129, 2007.

[KRV07] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 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.