Integrating Prose as First-Class Citizens with Models and Code

Markus Voelter

independent/itemis, voelter@acm.org

Abstract. In programming and modeling we strive to express structures and behaviors as formally as possible to support tool-based processing. However, some aspects of systems cannot be described in a way that is suitable for tool-based consistency checking and analysis. Examples include code comments, requirements and software design documents. Because they can only be analyzed manually, they are often out-of-sync with the code and do not reflect the current state of the system. This paper demonstrates how language engineering based on language workbenches can help solve this problem by seamlessly mixing prose and program nodes. These program nodes can range from simple references to other elements over variables and formulas to embedded program fragments. The paper briefly explains the language engineering technology behind the approach as well as a number of prose-code integrated languages that are part of mbeddr, an integrated language and tool stack for embedded software engineering.

1 Introduction

Even though developers and systems engineers would love to get rid of prose as part of the development process and represent everything with machine-processable languages and formalisms, prose plays an important role.

In *requirements engineering*, prose is the starting point for all subsequent formalizations. Classical requirements engineering uses prose in Word documents or Doors databases, together with tables, figures and the occasional formula. Since these requirements are not versioned together with the code, it is hard to branch and tag them together with the implementation. In safety-critical domains, requirements tracing is used to connect the requirements to implementation artifacts. Traceability across tools is challenging in terms of tool integration.

During the implementation phase, developers add *comments* to the code. These comments must be associated with program elements expressed in various languages. For example, an architecture description language, a state machine modeling language or a business rule language are considered as part of the implementation. Comments also refer to code (for example, a comment that documents a function typically refers to the arguments of that function), and it is hard to keep these code references in sync with the actual code as it evolves.

Depending on the process, various *design documents* must be created during or after the implementation. These are different from code comments in that

they look at the bigger picture and "tell a story"; they are not inlined into the code, they are separate documents. Nonetheless they are tightly integrated with the code, for example, by referring to program elements or by embedding code fragments. Today, such documents are usually written in Latex, Docbook or Word – and synchronized manually with the implementation code.

Problem Prose is is often badly integrated with the artifacts it relates to. It cannot be checked for consistency with implementation artifacts. Mixing prose and code or models is hard: either they reside in separate files, or, if pseudo-code is embedded into a requirements document, it is not checked with regards to syntax and type system rules. No IDE support for the programming or modeling language is available. This leads to a lot of tedious and error-prone manual synchronization work.

Contribution This paper proposes a highly integrated approach for handling prose in the context of model-driven engineering tools that solves the challenges outlined above. The implementation behind the approach relies on language engineering and language workbenches, and an implementation has been developed as part of the mbeddr platform.

2 mbeddr and MPS

mbeddr[1] is an open source project supporting embedded software development based on incremental, modular domain-specific extension of C [7,8]. It also supports languages that address other aspects of software engineering such as requirements or documentation (which is what is discussed in this paper).

mbeddr Overview mbeddr builds on the JetBrains MPS language workbench[2], a tool that supports the definition, composition and use of general purpose or domain-specific languages. MPS uses a projectional editor, which means that, although a syntax may look textual, it is *not* represented as a sequence of characters which are transformed into an abstract syntax tree (AST) by a parser. Instead, a user's editing actions lead *directly* to changes in the AST. Projection rules render a concrete syntax from the AST. Consequently, MPS supports non-textual notations such as tables, and it also supports unconstrained language composition and extension – no parser ambiguities can ever result from combining languages (see [6] for details).

The next layer in mbeddr is an extensible implementation of the C99 programming language in MPS. On top of that, mbeddr ships with a library of reusable extensions relevant to embedded software. As a user writes a program, he can import language extensions from the library into his program. The main extensions include test cases, interfaces and components, state machines, decision tables and data types with physical units[3]. For many of these extensions, mbeddr provides an integration with static verification tools (model checking

[1] `http://mbeddr.com`

[2] `http://jetbrains.com/mps`

[3] I do not distinguish between *models* and *code*. While C99 artifacts would probably be called code, state machines would likely be called models. Since both are

state machines, verifying interface contracts or checking decision tables for consistency and completeness; see also [5]).

Finally, mbeddr supports three important aspects of the software engineering process: requirements engineering and tracing [9], product line variability and documentation. All are implemented in a generic way that makes them reusable with any mbeddr-based language. We discuss the prose aspect of requirements, documentation and code comments in the rest of this paper.

Multiline Text Editing The projectional nature of the MPS editor has important advantages with regards to extensibility of languages. However, it also means that the editor is a bit more rigid than a regular text editor. In particular, until recently, MPS did not support multiline strings with the familiar editing experience where pressing `Enter` creates a line break, pressing ↑ moves the cursor to the line above the current one, or deleting a few words on a line "pulls up" the text from the next line. However, the `mps-multiline`[4] MPS plugin, developed by Sascha Lisson, has enabled this behavior. In addition, an additional plugin[5] supports embedding program nodes into this multiline prose. At any location in the multiline text, a user can press `Ctrl-Space` and select from the code completion menu a language concept. An instance of this concept is then inserted at the current location. The program node "flows" with the rest of the text during edit operations. Other editing gestures can also be used to insert nodes. For example, an existing regular text word can be selected, and, using a quick fix, it can wrapped with an `emph(...)` node, to mark the word as *emphasized*.

The set of language concepts that can be embedded in prose text this way is extensible; the concept simply has to implement the `IWord` interface. For a developer who is familiar with MPS, the implementation takes only a few minutes.

Implementing an Embeddable Word In MPS, language elements (called *concepts*) have children, references and properties. They can also inherit from other concepts and implement concept interfaces such as `IWord`. The multiline prose editor widget works with instances of `IWord`, and by implementing this interface we can "plug in" new language concepts into the multiline editor. An example is `ArgRefWord` which can be embedded into function comments to reference an argument of that function:

```
concept ArgRefWord implements IWord
  references:              concept properties:
    Argument arg 1           transformKey = @arg
```

It states that the concept implements `IWord`, that it references one `Argument` (by the role name `arg`) and it uses the `@arg` transformation key: typing `@arg` in a comment, followed by `Ctrl-Space`, instantiates an `ArgRefWord`.

---

tightly integrated in mbeddr, the distinction makes no sense and I use the two terms interchangeably.

[4] `http://github.com/slisson/mps-multiline`

[5] `http://github.com/slisson/mps-richtext`

A reference to an argument should be rendered as `@arg(argName)`, so we have to define an appropriate editor:

```
[- @arg ( %arg%->{name} ) -]
```

The editor defines a list of cells `[- -]` inside which we define the constant `@arg`, followed by the `name` property of the referenced `Argument`, enclosed in parentheses. To restrict this `IWord` to comments of functions, a constraint is used:

```
can be child constraint for ArgRefWord {
  (node, parent, operationContext)->boolean {
    node<> comment = parent.ancestor<DocumentationComment>;
    node<> owner = comment.parent;
    return owner.isInstanceOf(Function) }
```

We also define the scope for the `arg` reference, since only those arguments owned by the function under which the documentation comment lives are valid targets:

```
link {arg} scope: (refNode, enclosingNode)->sequence<node<Argument>>) {
     enclosingNode.ancestor<Function>.arguments; }
```

Finally, a generator has to be defined that is used when HTML or LaTeXoutput is generated. In this case we simply override a behavior method that returns the text string that should be used:

```
public string toTextString() overrides IWord.toTextString {
  "@arg(" + this.arg.name + ")"; }
```

This completes the implementation. All in all, only 10 lines of code have to be written (the remaining ones shown above are IDE scaffolding)

3 Integrating Prose with Models

In this section we look at various examples of integrating prose with code, addressing the challenges discussed in Section 1.

3.1 Requirements Engineering

As discussed in [9], mbeddr's requirements engineering support builds on the following three pillars. First, requirements can be collected as part of mbeddr models and they are persisted along with any other code artifact. A requirement has an ID, a prose description, relationships to other requirements (`refines`, `conflicts with`) as well as child requirements. Second, the requirements language is extensible in the sense that arbitrary additional attributes (described with arbitrary DSLs) can be added to a requirement. Examples include business rules or use cases, actors and scenarios. The third pillar is traceability: trace links can be attached to any program element in any language.

In the context of this paper, the interesting aspect is that the prose description can contain additional nodes, such as references to other requirements (the `§req` nodes in Fig. 1). References to actors, use cases and scenarios are also supported. Since these are real references, they are automatically renamed if the target element is renamed. If the target element is deleted, the reference breaks and leads to an error. Referential integrity can easily be maintained.

```
1 Once a flight lifts off, you get 100 points
  PointsForTakeoff /functional: tags
  [ ... points are multiplied by the §req(PointsFactor), discussed below. ]
```

Fig. 1. Requirements descriptions can contain references to other requirements (the `§req` node in the text above), as well as references to actors, use cases and scenarios.

```
//[This state machine has separate states for the
  important flight phases, such as
  @child(beforeFlight) or @child(airborne).]
statemachine FlightAnalyzer initial = beforeFlight {
  state beforeFlight {
    on next [tp->alt > 0 m] -> airborne
    exit { points += TAKEOFF; }
  } state beforeFlight
  ...
```

Fig. 2. A state machine with a comment attached to it. Inside the comment, we reference two of the states of the state machine.

Note that the mainstream requirements management tool, DOORS, cannot embed references in the requirements description, they can only be added as a separate attribute, which is awkward in terms of the semantic connection between the text and the reference.

3.2 Code Comments

In classical tools, a comment is just specially marked text in the program code. As part of this text, program elements (such as module names or function arguments) are mentioned. We observe two problems with this approach. First, the association of the comment with the commented element is only by proximity and convention – usually, a comment is located above the commented element (this is true only in textual editors, graphical modeling tools usually do not have this problem). Second, references to other program elements are by name only – if the name changes, the reference is invalid. mbeddr improves on both counts.

First, a comment is not just associated by proximity with the commented program node, it is actually *attached* to it. Structurally the comment is a child of the commented node, even though the editor shows it on top (Fig. 2). If the element is moved, copied, cut, pasted or deleted, the comment always goes along with the commented element.

Second, comments can contain `IWord`s that refer to other program elements. For example, the comment on the state machine in Fig. 2 references two of the states in the state machine. Some of the words that can be used in comments can be used in any comment (such as those that reference other modules or functions), whereas others are restricted to comments for certain language concepts (references to states can only be used in comments on or under a state machine).

Note that some IDEs support real references in comments for a specific language (for example, Eclipse JDT renames argument names in JavaDoc comments

```
mbeddr supports physical units. For example, \code(struct) members can have
physical units in addition to their types. An example is the @cc(Trackpoint/)
in the @cm(DataStructures) module. Here is the \code(struct):
```

Fig. 3. This piece of document code uses `\code` tags to format parts of the text in code font. It also references C program elements (using the `cm` and `cc` tags). The references are actual, refactoring-safe references. In the generated output, these references are also formatted in code font.

for functions if an argument is renamed). mbeddr's support is more generic in that it automatically works for any kind of reference inside an `IWord`. This is important, since a cornerstone of mbeddr is the ability to extend all languages used in it (C, the state machine language or the requirements language). The commenting facility must be similarly generic.

3.3 Design Documents

mbeddr supports a documentation language. Like other languages for writing documents (such as LaTeX or Docbook), it supports nested sections, text paragraphs and images. We use special `IWord`s to mark parts of texts as emphasized, code-formatted or bold. Documents expressed in this language live inside MPS models, which means that they can be versioned together with any other mbeddr artifact. The language comes with generators to LaTeX and HTML, new ones (for example, to Docbook) can be added.

Referencing Code Importantly, the documentation language also supports tight integration with mbeddr languages, i.e. C, exiting C extensions or any other language developed on top of MPS. The simplest case is a reference to a program element. Fig. 3 shows an example.

Embedding Code Code can also be embedded into documents. In the document source, the to-be-embedded piece of code is referenced. When the document is generated to LaTeX or HTML, the actual source code is embedded either as text or as a screenshot of the notation in MPS (since MPS supports non-textual notations such as tables, not every program element can be sensibly embedded as text). Since the code is only embedded when the document is generated, the code is always automatically consistent with the actual implementation.

Visualizations A language concept that implements the `IVisualizable` interface can contribute visualizations, the context menu for instances of the element has a *Visualize* item that users can select to render a diagram in the IDE. The documentation language supports embedding these visualizations. As with embedding code, the document source references a visualizable element. During output generation, the diagram is rendered and embedded in the output.

4 Extensibility

A hallmark of mbeddr is that everything can be extended by end users (without invasively changing the extended languages), and the prose-oriented lan-

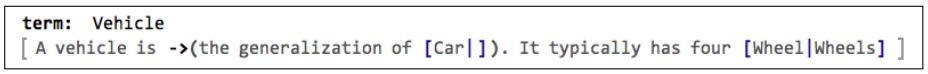

Fig. 4. A modular extension of the documentation language that supports the definition of glossary terms and the relationships between them. Terms can be referenced from any other prose, for example from comments or requirements.

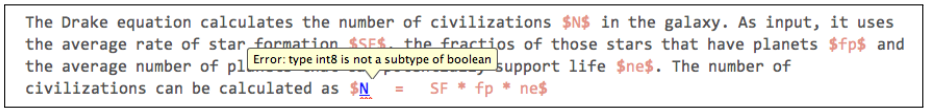

Fig. 5. An example where variable declarations and equations are integrated directly with prose. Since the expressions are real C expressions, they are type checked. To make this possible, the variables have types; these are specified in the properties view, which is not shown in the figure. To provoke the type error shown above, `boolean` has been defined as the type of the N variable.

guages can be extended as well. The extension mechanism that uses new language concepts that implement the `IWord` interface has already been discussed. This section discusses a few example of further extensions, particularly of the documentation language (Section 3.3).

Glossaries An obvious extension is support for glossaries. A glossary defines terms which can be referenced from other term definitions or from regular text paragraphs or even requirements or code comments. Such term definitions are subconcepts of `AbstractParagraph`, so they can be plugged into regular documents. Fig. 4 shows an example of a term definition.

The term in Fig. 4 also shows how other terms are referenced using the `[Term|Text]` notation (such references, like others, are generated to hyperlinks when outputting HTML). The first argument is a (refactoring-safe) reference to the target term. The optional second argument is the text that should be used when generating the output code; by default, it is the name of the referenced term. Terms can also express relationships to other terms using the `->(...)` notation, which creates a dependency graph between the terms in the glossary. A visualization is available that renders this graph as a diagram.

Formulas Another extension adds variable definitions and formulas to prose paragraphs (Fig. 5) which are exported to the math mode of the respective target formalism. However, the variables are actual referenceable symbols and the equations are C expressions. Because of this, the C type checker performs type checks for the equations (see the red underline under N in Fig. 5). mbeddr' interpreter for C expressions can be plugged in to evaluate the formulas. This way, live test cases could be integrated directly with prose.

Going Meta Section 3.3 has demonstrated how programs written in arbitrary languages can be integrated (by reference or by embedding) with documents written in the documents language. However, sometimes the *language definitions themselves* need to be documented, to explain how to develop languages in MPS/mbeddr. To make this possible, a modular extension of the documentation

language can be used to reference or embed language implementation artifacts. Similarly, documentation language documents can be embedded as well, to write documents that explain how to use the documentation language. The user guide for the documentation language[6] has been created this way.

Cross-Cutting Concerns mbeddr supports two cross-cutting concerns that can be applied to any language. Since the documentation language is *just another language*, it can be used together with these cross-cutting languages. In particular, the following two facilities are supported. First, requirements traces can be attached to parts of documents such as sections, figures or paragraphs. This way, requirements traceability can extend into, for example, software design documents. This is an important feature in safety-critical contexts. Second, mbeddr supports product line variability. In particular, static negative variability is supported generically. Using this facility, documents such as user guides, configuration handbooks or software design documents can be made variant-aware in the same way as any other product line implementation artifact.

Generating Documents Documents cannot just be written manually, they can also be generated from other artifacts. For example, mbeddr's requirements language supports generating reports, which contain the requirements themselves, the custom attributes (via specific transformations) and trace information. This feature is implemented by transforming requirements collections to documents, and then using the generators that come with the documentation language to generate the PDFs.

5 Related Work

The idea of more closely integrating code and text is not new. The most prominent example is probably Knuth's literate programming approach [4], where code fragments are embedded directly into documents; the code can be compiled and executed. While we have built a prototype with mbeddr that supports this approach, we have found referencing the code from documents (and generating it into the final PDF) more scalable and useful.

The closest related work is Racket's Scribble [2]. Following their paradigm of *documentation as code*, Scribble supports writing structured documentation (with Latex-style syntax) as part of Racket. Racket is an syntax-extensible version of Scheme, and this extensibility is exploited for Scribble. Scribble supports referencing program elements from prose, embedding scheme expressions (which are evaluated during document generation) and embedding prose into code (for JavaDoc-like comments). The obligatory literate programming example has also been implemented. The main differences between mbeddr's approach and Racket Scribble is that Scribble is implemented as Racket macros, whereas mbeddr's facility are based on projectional editing. Consequently, the range of document styles and syntactic extensions is wider in mbeddr. Also, mbeddr directly supports embedding figures and visualizations.

[6] `http://bit.ly/10gUs0q`

Essentially all mainstream tools (incl. modeling tools, requirements management tools or other engineering tools) treat prose as an opaque sequence of characters. None of the features discussed in this paper are supported. The only exception are Wiki-based tools, such as the Fitnesse tool for acceptance testing[7]. There, executable test cases are embedded in Wiki code. A big limitation is that there is no IDE support for the (formal) test case description language embedded into the Wiki markup. mbeddr provides this support for arbitrary languages.

One exception to the statement made above is Mathematica[8], which supports mixing prose with mathematical expressions. It even supports sophisticated type setting and WYSIWYG. Complete books, such as the Mathematica book itself, are written with Mathematica. mbeddr does not support WYSIWYG. However, mbeddr documents support integration with arbitrary MPS-based languages, whereas Mathematica has a fixed programming language.

One way of integrating program code and prose that is often used in book publishing are custom tool chains, typically based on LaTeX or Docbook. Program files are referenced by name from within the documents, and custom scripts copy in the program code as part of the generation of the output. mbeddr's approach is much more integrated and robust, since, for example, even the references to program fragments are actual references and not just names.

mbeddr's approach to integrating references (to, for example, text sections, figures or program nodes) into documents relies on user-supplied mark up: a reference must be inserted explicitly, either when creating the document, or using a refactoring later. mbeddr makes no attempt at automatically understanding, parsing or checking natural language (in contrast to some approaches in requirements engineering [1,3]). My experience is that such approaches are not yet reliable enough to be used in everyday work. However, it would be possible to add automatic text recognition to the system; an algorithm would examine existing text-only documents and introduce the corresponding nodes. We have built a prototype for the trivial case where a term is referenced from another term in the glossaries extension: by running a quick fix on a glossary document, plain-text references to terms are replaced by actual term references.

mbeddr relies on MPS, whose projectional editor is one of the core enablers for modular language extension. This means that arbitrary language constructs with arbitrary syntax can be embedded into prose blocks. I have seen a prototype of embedding program nodes into comments in Rascal[9]. However, at this point I do not understand in detail the limitations and trade-offs of this approach. However, one limitation is that the syntax is limited to parseable textual notations.

6 Conclusion

mbeddr is a scalable and practically usable tool stack for embedded software development. However, a secondary purpose of mbeddr is to serve as a convincing

[7] `http://fitnesse.org/`
[8] `http://www.wolfram.com/mathematica/`
[9] `http://www.rascal-mpl.org/`

demonstrator for the *generic tools, specific languages* paradigm, which emphasizes language engineering over tool engineering: instead of adapting a tool for a specific domain, this paradigm suggests to use generic language workbench tools and then use language engineering for all domain-specific adaptations.

As this paper shows, this approach can be extended to prose. Through the ability to embed program nodes into prose, prose can be checked for consistency with other artifacts. Of course, this does not address all aspects of prose. For example, consider a program element (such as a function) that is referenced from a prose document that explains the semantics of this program element. If the semantics changes (by, for example, changing the implementation of the function), the *explaining* prose does not automatically change. However, `Find Usages` can always be used to find all locations where in prose a program element is referenced. This simplifies the subsequent manual adaptations significantly.

Since prose is now edited with an IDE, some of the IDE services can be used when editing documents: go-to-definition, find usages, quick fixes, refactorings (to split paragraphs or to introduce term references in prose) or visualizations. Taken together with the direct integration with code artifacts, this leads to a very productive environment for managing requirements or writing documentation.

Acknowledgements I thank the mbeddr and MPS development teams for creating an incredibly powerful platform that can easily accommodate the features described in this paper. We also thank Sascha Lisson for building developing the `multiline` and `richtext` plugins for MPS.

References

1. V. Ambriola and V. Gervasi. Processing natural language requirements. In *Proceedings of the 12th IEEE Intl. Conf. on Automated Software Engineering, 1997*.
2. M. Flatt, E. Barzilay, and R. B. Findler. Scribble: closing the book on ad hoc documentation tools. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, New York, NY, USA, 2009. ACM.
3. V. Gervasi and B. Nuseibeh. Lightweight validation of natural language requirements. *Software: Practice and Experience*, 32(2):113–133, 2002.
4. D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
5. D. Ratiu, M. Voelter, B. Schaetz, and B. Kolb. Language Engineering as Enabler for Incrementally Defined Formal Analyses. In *FORMSERA'12*, 2012.
6. M. Voelter. Language and IDE Development, Modularization and Composition with MPS. In *4th Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2011)*, LNCS. Springer, 2011.
7. M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Journal of Automated Software Engineering*, 2013.
8. M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proc. of the 3rd conf. on Systems, programming, and applications: software for humanity*, SPLASH '12, pages 121–140, New York, NY, USA, 2012. ACM.
9. M. Voelter and F. Tomassetti. Requirements as first-class citizens: Tight integration between requirements and code. In *Proc. of the 2013 Dagstuhl Workshop on Model-Based Development of Embedded Software*, 2013.