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. Consequently, 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 techniques 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 such as aerospace, medicine or automotive, requirements tracing is required by standards such as IEC61508 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 including architecture description languages, state machines or business rule languages. Comments also refer to code (for example, a comment on 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 in software projects. These are different from code comments in that they look at the bigger picture and "tell a story"; they are not embedded in 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 are often 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 those 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, no syntax or type system checks are performed. 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 an integrated approach for handling prose in the context of model-driven engineering tools that solves the above challenges. 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 [8,9]. 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 change the AST *directly*. Projection rules render a concrete syntax from the AST. Consequently, MPS supports non-textual notations such as tables or mathematical symbols, and it also supports unconstrained language composition and extension – no parser ambiguities can ever result from combining languages (see [6] for details).

mbeddr is a set of languages implemented with MPS centered around an extensible implementation of C99 (see Fig. 1). On top of that, mbeddr ships with a library of reusable extensions relevant to embedded software, including test cases, components, state machines, decision tables and data types with physical units. For many of these extensions, mbeddr provides an integration with static verification tools (model checking state machines, verifying interface contracts or checking decision tables for consistency and completeness; see also [5]).

Users can use or build languages at any abstraction level and at any degree of rigor. For example, the C implementation is very rigorous and low level, the state machines are more abstract but just as rigorous, and the prose support discussed in this paper is not very rigorous at all. While C99 artifacts would probably be called *code*, state machines would likely be called *models*. Since both are tightly

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

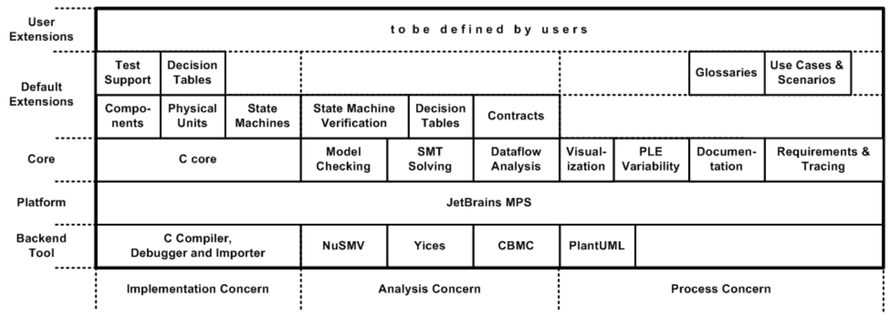

Fig. 1. The core building blocks of mbeddr; see Section 2 for details.

integrated in mbeddr, the distinction makes no sense. The terms *model* and *code* are used interchangeably.

Finally, mbeddr supports three important aspects of the software engineering process: requirements engineering and tracing [10], product line variability and documentation. All are implemented in a generic way that makes them reusable with any mbeddr-based language. The rest of this paper discusses the prose aspect of requirements, documentation and code comments.

Multiline Text Editing The projectional nature of the MPS editor has important advantages with regards to language extension and composition. However, traditionally, projectional editors have had usability challenges: they way users had to interact with the editor were very different from how users interact with text editors. MPS largely solves these problems. A detailed discussion of how MPS achieves this improved usability can be found in Section 7.3.1 of [7].

To support multiline text editing, developers can use the `mps-multiline`[3] MPS plugin, developed by Sascha Lisson. In addition, the `mps-richtext` plugin[4] 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, "flowing" 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*.

It is crucial to emphasize that embedded program nodes are not just simple tags; they can be *any* language concept, from simple tags (emphasize, bold) through complete, type-checked expressions or sophisticated textual or tabular structures. To make a language concept embeddable in text, it has to implement the `IWord` interface. For a developer who is familiar with MPS, implementing a Word takes only a few minutes. Note also, that MPS as a language workbench provides complete IDE support for any language developed with MPS. This is also true for embeddable Words. The code completion menu (available via `Ctrl-Space`) allows users to only add those Words that are compatible in the

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

current context. The next paragraphs show how to implement such a Word (the paragraph may be skipped, it is not essential for the rest of the paper.)

Implementing an Embeddable Word In MPS, language concepts have children, references and properties. They can also inherit from other concepts and implement concept interfaces. The multiline editor widget works with concepts that implement `IWord`; by implementing this interface, new language concepts can be plugged 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`. A reference to an argument is rendered as `@arg(argName)`, so an appropriate editor is defined:

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

The editor defines a list of cells `[- -]`. The list contains 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) }
```

The scope for the `arg` reference expresses that 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 it is sufficient to override a behavior method that returns the text string that should be used:

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

3 Integrating Prose with Code and Models

This section looks at various examples of integrating prose with code, addressing the challenges discussed in Section 1.

3.1 Requirements Engineering

As discussed in [10], 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

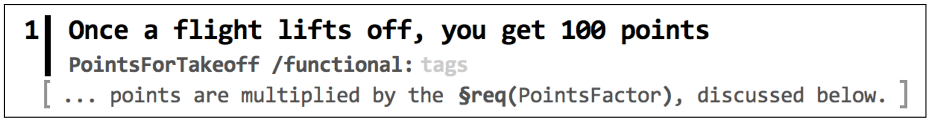

Fig. 2. Requirements descriptions can contain references to other requirements (the `§req` node in the text above), as well as references to actors, use cases and scenarios.

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. 2). 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.

3.2 Code Comments

In classical tools, a comment is just specially marked text in the code, often referring to program elements (such as module names or function arguments). This approach has two problems. First, in textual editors, the association of the comment with the commented element is only by proximity and convention – usually, a comment sits above the commented element. This can pose a problem to refactorings. 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 actually *attached* to the element it comments: structurally the comment is a child of the commented node, even though the editor may show it on top (Fig. 3); alternatively, based on the editor definition, it could also be shown on any other side of the of the state machine, or even in a separate view. 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. 3 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).

Some IDEs support real references in comments for a specific language (for example, Eclipse JDT renames argument names in JavaDoc method comments 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 any languages (C, the

```
// 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. 3. A state machine with a comment attached to it. Inside the comment, two of the states of the state machine are referenced.

```
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. 4. This document 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 output, the references are also formatted in code font.

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. Special `IWord`s are used 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 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. 4 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 output is generated, the actual source code is embedded either as text or as a screenshot (since non-textual notations such as tables cannot be sensibly embedded as text). Since the code is only embedded when the document is generated, the document 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

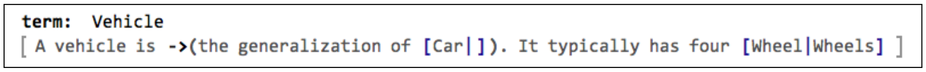

Fig. 5. A modular extension of the documentation language that supports the definition of glossary terms and the relationships between them.

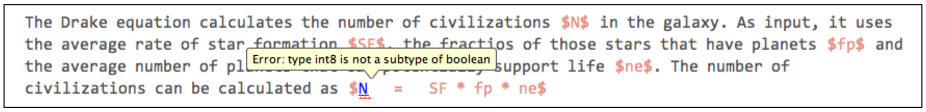

Fig. 6. An example where variable declarations and equations are integrated directly with prose. Since the expressions are real C expressions, they are type checked. To make this possible, the variables have types; these are specified in the properties view, which is not shown in the figure. To provoke the type error shown above, `boolean` has been defined as the type of the N variable.

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 languages are no exception. The mechanism based on 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 A glossary defines terms which can be referenced from other term definitions or from regular text paragraphs or even requirements or code comments. A term definition is a subconcepts of `AbstractParagraph`, so they can be plugged into regular documents. Fig. 5 shows an example of a term definition.

The term in Fig. 5 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. 6) which are exported to the math mode of the respective target formalism. However, the variables are actual symbols and the equations are C expressions. Both can be are checked for syntax and type correctness (see the red underline under N in Fig. 6). mbeddr's interpreter for C expressions can be plugged in to evaluate the formulas. By adding tables with test values for the expressions, users could even express tests for the formulas embedded in the prose code. Using the interpreter, these could be evaluated directly in the IDE.

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, variant markup can be added to documents such as user guides, configuration handbooks or software design documents. This way, it is easy to create variants of these documents along with variants of the software system they relate to.

Generating Documents Documents cannot just be written manually, they can also be generated from other artifacts, for example from requirements collections (introduced in Section 3.1) Such collections can be transformed to documents, and then, using the generators that come with the documentation language, they can be used 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 [4], where code fragments are embedded directly into documents; the code can be compiled and executed. A prototype of this approach has been built with mbeddr. However, it turned out that 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.

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. Only a few exceptions exist. One exception are Wiki-based tools such as Fitnesse (used for acceptance testing[5]). 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

[5] `http://fitnesse.org/`

arbitrary languages. Another exception is Mathematica[6], which supports mixing prose with mathematical expressions. It even supports sophisticated typesetting and WYSIWYG editing. 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]). 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[7]. 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 Conclusions and Future Work

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 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

[6] http://www.wolfram.com/mathematica/

[7] http://www.rascal-mpl.org/

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.

As part of our future work, we will integrate MPS' support for tabular and graphical notations with the support for prose, allowing users to embed prose paragraphs in table cells or graphical shapes.

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. I 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. ACM, 2009.
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, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering – Designing, Implementing and Using Domain-Specific Languages.* CreateSpace Publishing Platform, 2013.
8. 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.
9. 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.
10. 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.