

Embedding Formal Models into Latex Documents

Jan Kubovy

Institute for Application Oriented Knowledge Processing
Johannes Kepler University in Linz
Altenbergerstrasse 69, 4040 Linz, Austria
Email: jan.kubovy@faw.jku.at

Josef Küng

Institute for Application Oriented Knowledge Processing
Johannes Kepler University in Linz
Altenbergerstrasse 69, 4040 Linz, Austria
Email: josef.kueng@faw.jku.at

Abstract—Formal methods proved themselves to support provable and unambiguous models. Still a lot of manual work is needed to assure consistency during the modeling process. In this paper we propose a support tool in form of a latex package helping a modeler to embed formal code and related description into a specification document. This package helps to assure relative consistency between abstraction levels, structuring the formal ground model, and improving the reuse of parts of the ground model in different stages of the specification cycle.

I. INTRODUCTION

Development of a system is a common task in different technical disciplines, during which a ground model or a blueprint of it is developed before own implementation. This topic is targeted by many resulting into best practices such as [1] or formal approaches, e.g., [2], [3]. Typically a development cycle of a system consists of requirement capture, technical specification, design and implementation. Those stages involve documents targeting different audience, where such documents serve as a communication basis between involved parties at certain stage of the development cycle [2]. The usage of formal methods is motivated by the expectation that they contribute to the correctness, robustness, and reliability of the design. The need to use such methods increases with the growing complexity of the developed systems. To improve the requirement capture technique a set of convention is usually introduced to help facilitate reasoning [3]. For this purpose we developed a supporting latex package, which helps to ensuring relative consistency between abstraction levels and supports reuse of the developed ground model or its parts.

In this paper we describe the ground model latex package developed as a support tool for writing documents with embedded formal code. The problem lies in the fact that signatures, descriptions and also implementations of rules and functions are usually written directly into the document or are manually copy-pasted from one document to another. Such an approach tends to result in inconsistent code, where changes require much manual effort as soon as such specifications grows over certain, size and it is very hard to keep all relevant documents up-to-date with each other. The basic idea was to separate the formal code of the ground model from the rest of the informal text of a specification. This allows also the use of other tools developing or simulating the code. Such a formal code can be directly embedded into the latex document, and can be possibly reused in more documents and a lot of time and effort spent on the code maintenance, especially in case of changes in the ground model, can be saved. The user will be able to see and manage changes in both the documents and the ground model

separately. The signatures will be consistent and unified over the whole document. Description of a concrete function or rule can be reused in all the documents if necessary. Referring to a rule or a function is easy as calling a latex referencing commands targeting the auto-generated labels.

This paper is structured as follows. In section II we will introduce the directory structure of the ground model. In section III the file structure holding the formal code with its parts and conventions will be described. In section IV the provided commands by this package will be shown and their usage will be discussed.

II. DIRECTORY STRUCTURE AND FILE NAMING CONVENTIONS

For better orientation a package-based directory structure is used to help preventing name conflicts [4]. This means that the ground model may be logically separated into different packages where every package is represented by its own directory in the root directory. The root of the ground model can be specified using the `\asmCodePrefix{[path]}` command taking one parameter representing the relative path to the root directory of the ground model. The directory structure in every package directory follows the same rules and is based on separating rules from functions and different function types from each other. The default directory structure configuration is shown in Fig. 1.

```
-- [ground model root]
|  |-- [package A]
|  |  |-- functions
|  |  |  |-- basic
|  |  |  |  |-- dynamic
|  |  |  |  |  |-- controlled
|  |  |  |  |  |-- monitored
|  |  |  |  |  |-- out
|  |  |  |  |  |-- shared
|  |  |  |-- derived
|  |  |-- rules
|  |  ...
|  |-- [package X]
|  |  |-- functions
|  |  |  ...
|  |  |-- rules
```

Fig. 1. Directory structure.

There is a directory for functions and a directory for rules. The function directory sub-tree is based on the ASM function/relation/location classification [2]. For cases such separation is to fine, the directory structure is configurable.

E.g., in case of small ground models or if a certain function type does not occur often enough it makes sense to flatten the directory structure a bit. For this purpose the `\asmPath{[type]}{[path]}` command is available. This command takes two parameters, first defines the function type or rule, which path should be modified and the second defines the relative path from the ground model package root where the files with the rules or function type will reside.

This file structure makes it easy to maintain in a file version system like Subversion [5] or GIT [6]. It reduces the amount of conflicts since the code is separated into packages, and functions and rules into separate files so users may make changes to only one package or file. If sub-packages are needed, e.g. in case of a more complex ground model, the parameter defining the package in the latex commands will need to contain the whole subpath of the package (see section IV).

Every function or rule has to be stored in a separate file. The name of the file is the name of the function or rule without input or output parameters and has to be stored in given directory based on the directory structure described in section II. The file names are case-sensitive. In case a function or rule has multiple different input/output parameter configuration but same name, than all such configurations will be stored in one file. The first function or rule in a file is the default one and the order of the rest is arbitrary. The default file extension is `cas`, but it may be changed using the `\asmCodeExt{[extension]}` command. The only restriction is that all source code files should have the same file extension.

III. FILE STRUCTURE

A. Documentation block

Every function or rule must be documented. Even if the function or rule description will be empty the empty documentation block has to be present in the source code file. The documentation was inspired by JavaDoc [7]. It has to begin with “`/**`” token and end with “`*/`” token on separate lines. The token “`/**`” is not allowed to be used anywhere else than for the function or rule documentation begin. In case of a comment block only “`/*`” token may be used.

There are three parts of the documentation block recognized by the package commands: description, private description and business signatures.

The *description* should describe the functionality of the function or rule. It should be written in a way that it can be read without the code being present, i.e., avoiding relative reference expressions, e.g., “... this function ...”. In case a function or rule has to be referenced the signature or the implementation figure can be referenced using the `\ref` command or similar. For auto-generated labels see section IV-A. Latex commands may be used in the *description*. However, all commands defined in this package, except of the `\asmFormat` would cause an unexpected behavior and must not be used inside a *description* block.

The *private description* can be written after the *description* separated with a “`* --`” line. This *private description* is ignored by the package commands and serves only the purpose of notes inside the source code file.

The *business signatures* is an optional list of higher level signatures, which can be used in the latex document, referenced by their absolute position in the list. The first one is treated as the default one and the rest should have an arbitrary but fixed order.

Additional annotations in the documentation block may be used, e.g., `@author`. Such annotated lines in the documentation block are ignored. The different parts can be separated with an empty line from each other for better readability. Empty line in a documentation block is a line containing only a space and a star (“ `*`”) and does not generates a new paragraph if used in the *description*. An example is shown in Fig. 2.

```

1 /**
2  * The derived function \asmFormat{exampleFunction}
3  * shown in signature
4  * \ref{sig:asm:monitored exampleFunction : A -> R}
5  * receives one parameters from the universe
6  * \asmFormat{A} and returns a value from the
7  * universe \asmFormat{R}.
8  *
9  * —
10 *
11 * Private description ignored by the package
12 * commands.
13 *
14 * @business exampleFunctionOfA
15 * @author John Doe <john.doe@example.com>
16 */
17 monitored exampleFunction : A → R

```

Fig. 2. Documentation example.

B. Signatures

There are two types of signatures: the *declaration signature* and the *definition signature*. Every, basic function, derived function and rule has a declaration signature, while only non-abstract derived functions and rules have a definition signature. The difference is that a declaration signature defines the type, name and universes for all input and output parameters and definition signature defines type, name and names of all input parameter names and is followed by the implementation of the derived function or rule. The declaration signature must be defined after the documentation block. The definition signature, if present, must be defined after the declaration signature. The function or rule declaration signatures are built in the following way:

- 1) The keyword `abstract` (at the beginning of the signature) belongs to a derived function or rule if such derived function or rule is abstract.
- 2) *The next step differs for functions and rules:*
 - Rules:
 - In case the rule in question is a main rule the keyword `main` will follow.
 - The keyword `rule` will follow.
 - Functions:
 - One of `controlled`, `monitored`, `out`, `shared`, `static` or `derived` keywords will follow depending on the function type.
- 3) Followed by the name of the function or rule. Naming convention differ for functions and rules: both, function and rule names are written in camel-case.

But, rules begin with a upper-case letter and functions begin with a lower-case letter.

- 4) If and only if a function or rule has one or more input parameters:
 - a) A colon follows the function or rule name surrounded by one space on both sides.
 - b) A list of universes of the parameters follows, separated using the symbol: “X” (upper-case “x”) surrounded by one space from both sides.
- 5) If and only if a function or rule have one or more output parameters:
 - a) The string: “->” follows (“slash/minus” sign followed by “greater than” sign) surrounded by one space at both sides.
 - b) The list of universes of the output parameters follows, separated using the symbol: “X” (upper case “x”) surrounded by one space at both sides as in the case of input parameters.

In case there are two or more functions or rules with the same name but different parameters they will all be defined in the same file. All different signatures needs their own documentation block. The first function or rule in a file is treated as *default* one. The rest may have arbitrary ordering. A signature has to be always on one line.

C. Implementation

Non-abstract derived functions and rules must contain an, even empty, implementation. An example of an implementation part can be seen in Fig. 3. Every implementation begins with a definition signature, which is built in the same way as the declaration signature described in section III-B till step 4. Then it continues as follows:

- 4 If and only if a function or rule has one or more input parameters:
 - a) An open parenthesis (“(”) follows the function or rule name without any surrounding spaces.
 - b) Followed by a comma separated list of parameter names.
 - c) A close parenthesis (“)”) follows without any surrounding spaces.
- 5 Signature ends with a space and equal sign (“=”).

The definition signature has to be always on one line. After the definition signature the own implementation follows, where the defined parameter names may be used, and every line of the implementation has to be indented.

```

1 derived exampleFunction(paramA, paramB) =
2   return res in
3     // own implementation
4     res ← someOperation(paramA, paramB)

```

Fig. 3. Implementation Examples

IV. LATEX COMMANDS PROVIDED BY THE PACKAGE

To unify formatting in a document the package provides the `\asmFormating` and `\asmFormat` commands.

The `\asmFormating` command takes exactly one required parameter, which is expected to be a declaration signature and reformats it in the following way:

- Replaces “->” (space, followed by slash/minus sign, followed by greater than sign, followed by space) with “→” symbol.
- Replaces “ X ” (space, followed by “X”, followed by space) with “×” symbol.
- Puts a *OK to hyphenate a word here* command (“\”) before every underscore (“_”) and between any case-change from lower-case to upper-case.

The `\asmFormat` command takes two parameters. The first one is required and is expected to be a declaration signature and the second one is optional and if it takes the value “abstract” it enforces abstract formatting. Internally this command uses the `\asmFormating` command to format the signature first and then it uses different font face for abstract and non-abstract signatures. If the signature contains the keyword “abstract” or the second parameter takes the value “abstract” then the font face for abstract signatures will be used. Otherwise the font face for non-abstract signatures will be used.

A. Including code to documents

In addition the package enables commands shown in Fig. 4, which can be used in a latex document. Where `pkg` rep-

```

1 \asm{pkg}{type}{name}[params][options]
2 \asmDescription{pkg}{type}{name}[params]
3 \asmName{pkg}{type}{name}[params]
4 \asmBusiness{pkg}{type}{name}[params][number]
5 \asmSig{pkg}{type}{name}[params]
6 \asmSignature{pkg}{type}{name}[params]
7 \asmImplementation{pkg}{type}{name}[params][options]
8 \asmFloat{pkg}{type}{name}[params][options][search]
9 \asmFile{pkg}{type}{name}

```

Fig. 4. Provided commands by the package.

resenting the package name, `type` representing “rule” or function type, `name` representing the function or rule name refers to the file structure described in section II. The optional parameter `params` is the part of the signature followed after the colon containing both input and output parameters with all the separators as in the source code. This becomes useful in case more than one function or rule is defined in the same source file. In case this parameter is not set in the command, the first, *default*, function or rule in the source file will be used.

The latex command `\asmDescription` will read the corresponding file with the function or rule, extract the description and include it into the document. The description can contain latex commands but there are some things the writer has to pay more attention to. E.g., that any opening bracket “[” has to be closed “]” on the same line. Otherwise the parsing will fail. In case “{” and “}” brackets are used in the implementation part of a function or rule and there is a need of having them on different lines, they have to be replaced with “([” and “)]” strings.

The package will transform the “ ([” into a “{” and the “])” into a “}” in the output document.

The commands `\asmSignature{pkg}{type}{name}[params]`, `\asmSig{pkg}{type}{name}[params]`, `\asmBusiness{pkg}{type}{name}[params][number]` and `\asmName{pkg}{type}{name}[params]` will read the corresponding source file with the function or rule, extract the signature and include it in different ways into the document. The difference between `\asmSignature{pkg}{type}{name}[params]` and `\asmSig{pkg}{type}{name}[params]` is that `\asmSignature{pkg}{type}{name}[params]` includes the complete signature in a box taking the whole page width while the `\asmSig{pkg}{type}{name}[params]` is for including the signature in-line in the surrounding text. Additionally the `\asmSignature{pkg}{type}{name}[params]` will prepend the prefix defined using the `\asmSignaturePrefix{[prefix]}` command. Such signature box can be placed multiple times in a document. The first time it is placed for a concrete declaration signature it will be numbered with a distinct counter and a label will be created for it in order to enable the possibility to reference such signature. The generated label will have the following format: `sig:asm:[signature]`, where the `[signature]` is a full signature including all keywords and universes as it is on the whole line in the source code.

The `\asmName{pkg}{type}{name}[params]` command is meant for in-line use as the `\asmSig` command, but it prints out only the name of the function or rule without any parameter universes.

The `\asmBusiness{pkg}{type}{name}[params]` command searches the documentation block for `@business` annotated entries and prints them in-line to the surrounding text. This command takes a fifth optional parameter identifying the position of the *business signature*. By default the first is taken if this parameter is left out.

The command `\asmImplementation{pkg}{type}{name}[params][options]`, where the fifth optional parameter can take the value “implementation” (default value) or “interspersed”, prints the implementation of a rule or derived function. If this parameter takes the default value “implementation” or is left out the implementation will be printed into one listing environment. If this parameter takes the value “interspersed” the comment blocks — those, which begin with “/*” and end with “*/” — will be taken out and printed as a normal text between the code parts.

For the purpose of the implementation the package defines an *ASM* language for the `listings` package.

Additionally the package provides `\asmFloat{pkg}{type}{name}[params][options]` command taking the same set of parameters as the `\asmImplementation{package}{type}{name}[parameters][options]`, does the same extraction and additionally wraps the implementation into a float environment, where the caption is a formatted signature of the function or rule and the float will be labeled with auto-generated label in the following format: “`fig:asm:[signature]`”, where the `[signature]` is a full signature including all keywords and universes.

Last, the package provides also a command allowing full inclusion of a function or rule into a document. This can be done using the `\asm{pkg}{type}{name}[params][options]` command, where the parameters take same values as in case of `\asmImplementation{package}{type}{name}[parameters][options]`

described above. This command prints the description of the function or rule followed by its signature and followed by the implementation if such a function or rule has an implementation. The fifth optional parameter takes additionally a “float” value, which makes the implementation be wrapped into a float environment. In case the fifth parameter needs to take more than one value, those should be separated by a comma.

V. CONCLUSION

In this paper we showed and described a support tool for writing specification documents with embedded formal code, its options and limitations. Usually a development cycle involves more than one document and those tend to detach over time in later phases. This tool helps to structure the underlying ground model of a specification, helps to identify missing points and helps to ensure relative consistency of the document across the abstraction levels. We use this tool in our projects where it already proved itself to be a great support especially in cases when changes are introduced into an ongoing specification process to identify inconsistencies or parts to be refined.

REFERENCES

- [1] Office of Government Commerce (OGC), *ITIL Version 3: Service Design*. The Stationery Office, 2007.
- [2] E. Börger and R. F. Stärk, *Abstract State Machines - A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
- [3] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, jun 2010. [Online]. Available: <http://www.cambridge.org/us/academic/subjects/computer-science/programming-languages-and-applied-logic/modeling-event-b-system-and-software-engineering?format=HB>
- [4] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Upper Saddle River, NJ: Addison-Wesley Longman, 2005. [Online]. Available: <http://java.sun.com/docs/books/jls/>
- [5] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control with Subversion*, mar 2011, no. r4723, for Subversion 1.6. [Online]. Available: <http://svnbook.red-bean.com/en/1.6/svn-book.pdf>
- [6] J. Loeliger, *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*, 1st ed. O’Reilly, jun 2009. [Online]. Available: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0596520123>
- [7] D. Kramer, “API Documentation from Source Code Comments: A Case Study of Javadoc,” in *Proceedings of the 17th Annual International Conference on Computer Documentation*, ser. SIGDOC ’99, J. Johnson-Eilola and S. A. Selber, Eds. New York, NY, USA: Association for Computing Machinery (ACM), 1999, pp. 147–153. [Online]. Available: <http://doi.acm.org/10.1145/318372.318577;http://dblp.uni-trier.de/db/conf/sigdoc/sigdoc1999.html#Kramer99>