RbG: A Documentation Generator for Scientific and Engineering Software

Michael Moser and Josef Pichler
Software Analytics and Evolution
Software Competence Center Hagenberg
4232 Hagenberg, Austria
Email: (michael.moser, josef.pichler)@scch.at

Günther Fleck and Michael Witlatschil
Siemens Transformers Austria
8160 Weiz, Austria
Email: (guenter.fleck, michael.witlatschil.ext)@siemens.com

***Abstract*—This paper demonstrates *RbG*, a new tool intended for the generation of high-quality documentation from source code of scientific and engineering applications. *RbG* extracts mathematical formulae and decision tables from program statements by means of static code analysis and generates corresponding documentation in the Open Document Format or LaTeX. Annotations in source code comments are used to define the structure of the generated documents, include additional textual and graphical descriptions, and control extraction of formulae on a fine-grained level. Furthermore, *RbG* provides an interpreter to generate function plots for extracted formulae. In this tool demonstration we briefly introduce the tool and show its usage for different scenarios such as reverse engineering and re-documentation of legacy code and documentation generation during development and maintenance of software.**

I. Introduction

Companies in industrial domains (e.g. electrical engineering and process engineering) run and maintain scientific and engineering software for a variety of tasks in construction, simulation, and interpretation of data. Software, containing complex and extensive computations, is often highly optimized with respect to running time and memory requirements and developed by domain experts over years or decades [1]. Decades ago Fortran (in particular, FORTRAN 77) dominated programming in science and engineering, especially when one of the numerous Fortran libraries was required [2]. Although the importance of C/C++ increased in all programming fields, and especially in science and engineering, Fortran is still relevant today. The Fortran language, as well as compilers, evolved towards a sophisticated tool including modern programming paradigms of object-orientation and parallel programming. While some companies moved from Fortran to C/C++, others where stimulated by these enhancements and still pursue software development with Fortran even for new software systems. Today, both Fortran and C/C++ are predominant for programming in science and engineering.

Maintenance and evolution of software requires extensive documentation of domain knowledge, including physical and mathematical description of the problem together with selected numeric algorithms to solve the problems. Furthermore, documentation serves also as a kind of specification that needs to be approved by a principal domain expert, without programming expertise [3]. The co-existence of source code and documentation implicates two challenges. First of all, maintaining both documentation and source code independently results in high writing effort and in a documentation which is likely to be inconsistent by itself and inconsistent with respect to the documented source code. Secondly, documentation may be out-of-date or non-existent at all for legacy code maintained and evolved over several years [4].

To tackle both challenges, we propose generating documentation from annotated source code. Therefore, we provide *RbG*, a tool that automatically *transforms* source code into high-level, user-readable documentation. *RbG* was originally designed and implemented to automatically extract technical documentation from Fortran 90 programs in the electrical engineering domain. In this scenario, a Fortran programmer annotates program code with special tags and uses *RbG* to generate the corresponding documentation from annotated source code. A previous paper [3] summarizes industrial experience from this scenario with an initial version of the *RbG* tool. In the meantime, we extended *RbG* in several dimensions:

1) An additional parsing front end for C/C++ was integrated resulting in a *multi-language* tool.
2) Additional high-level documentation fragments are generated, including decision tables and function plots.
3) *RbG* is used to reverse engineer and re-document legacy code in other domains.

In the remainder of this paper we present the architectural overview together with details on the implementation (Section II) followed by a more detailed discussion on its inner working by means of an example (Section III). A short evaluation and discussion on implementing and using the *RbG* tool is provided in Section IV. Furthermore, a demo video and tool description can be found on `www.scch.at/en/rbg-tool`.

II. The Rulebook Generator Tool

A. Architecture

The architecture of *RbG* supports the extension towards new programming languages, static analysis techniques and output document formats. Fig.1 aligns components along the three main phases an execution runs through; namely, parsing input sources, analysis of source code models, and documentation generation.

978-1-4799-8469-5/15/$31.00 © 2015 IEEE

The parsing phase is determined by two main components; the *source code parser* and a *comment parser*. To parse program sources *RbG* reuses parsing infrastructure from Metamorphosis [1], our toolkit based on open source parsers for analyzing legacy software systems. For *RbG*, the parsing front ends for Fortran and C/C++ are reused. The *source code parser* transforms program sources into an intermediate representation, the generic abstract syntax tree (GAST). The GAST is independent of a specific programming language and is built up either directly during parsing or by transformation of language dependent abstract syntax trees. Next to source code statements *RbG* specific annotations determine the content and structure of the generated documentation. The *comment parser* extracts these annotations from source code comments or from separate *symbol files*. We apply static code analysis [5] on the GAST to create an application specific *documentation model*. Static analysis is either directly performed upon the GAST or upon more specific representations (i.e. call graph and control flow graph) which better facilitate transformation of program code into the documentation model. We pursue an intraprocedural analysis of program code, whereas the processing order of functions and subroutines is determined by the call graph starting from the main program or the main entry point of a library.

Within a function or subroutine, a sequence of program statements is translated into formulae and decision tables. A comprehensive description on static analysis techniques applied by *RbG* can be found in [6]. If a fully automatic generation process is performed, i.e. no manual annotations are added to program sources, structure and content of the generated documentation largely resembles a mapping of assignment statements in source code to formulae and decision tables in the generated documentation. In order to control content and structure on a fine grained level, annotations can be used to translate, substitute or suppress program statements. *Translation* replaces identifiers of program variables and functions with symbol names defined by annotations. *Substitution* replaces the usage of a program variable within an expression statement by its last assigned expression and *suppression* removes all formulae and decision tables referring to a particular variable from the created documentation model. In a subsequent transformation step the documentation model is merged with structural information provided by the annotation of sections and paragraphs. The resulting *documentation model* is input for the document generation process which produces documentation in LaTex or Open Document Format.

B. Implementation

RbG is implemented in the Java programming language. For parsing Fortran and C/C++ open source parsers are reused. To parse Fortran source code we use the Open Fortran Parser (http://fortran-parser.sourceforge.net); for C/C++ the Eclipse CDT (https://eclipse.org/cdt/) parsing infrastructure for C/C++ is reused. To represent abstract syntax tree models created by these parsers in a language independent way the Abstract Syntax Tree Metamodel (ASTM) [7] is used. ASTM defines a specification for modeling elements to express abstract syntax trees (AST) in a representation that is sharable among multiple tools from different vendors. Among others, the ASTM specification defines the Generic Abstract Syntax Tree Metamodel (GASTM). GASTM is a generic set of language modeling elements common across numerous languages and establishes a common core for language modeling, called the Generic Abstract Syntax Trees (GAST). *RbG* uses an Java implementation of GAST generated by the Eclipse Modeling Framework (EMF) [8].

C. Availability

RbG can be executed as a command line tool, from within the *Visual Studio* development environment and from an online service. Major program options fall into the categories *input specification* (i.e. specification of symbol and source files), *reporting* (i.e. logging, verbose mode for tool and parser preprocessor), and output customization (i.e. template specification for Open Document Format, configuration of document styles). While the command line tool and *Visual Studio* integration provide full access to all configuration variants the online service can only be used with a predefined configuration. Unfortunately, the command line tool is not publicly available, however, the online service can found at `http://codeanalytics.scch.at`.

III. Rulebook Generator By Example

Given the code fragment in Listing 1 containing the two C functions `sine` and `i`, *RbG* generates the documentation fragment given in Fig. 2 in the Open Document Format.

RbG transforms the Taylor series for *sin(x)* implemented in the lines 7-13 into a closed formula shown at the top of Fig. 2. The left hand side of the resulting formula is specified by the `@symbol` annotation in the source code (line 2). The right hand side of the formula results from substitution of variables `p` and `term` in subsequent statements. Both occurrences of variable `p` in line 10 are substituted with the expression `2*n+1`. Afterwards, the variable `term` is substituted by the expression `pow(x,2*n+1)/F(2*n+1)` in line 11, and by the expression `pow(-1, n+1)*(pow(x, 2*n+1)/F(2*n+1))` in line 12. Furthermore, *RbG* detects the calculation of the sum over `term` implemented in line 12 within the `for` loop statement, resulting in the final formula. Note that the implementation of function `F` calculating the factorial of a non-negative integer n is omitted in the source code. Besides the symbol annotation (`@symbol`) and the annotation for variable substitution (`@substitute`) in line 3, the result can be achieved automatically. The annotation `@suppress` commands *RbG* to omit formulae for the annotated variable. In most cases, both annotations (`@substitute` and `@suppress`) appear together; however *RbG* supports substitution and suppression of formulas independent of each other.

The transformation of function `i` results in three formulae, one decision table, and a function plot (see Fig. 2). The transformation of program statements resulting in formulae follows the idea described above. However, the extraction of

the decision table for ω demonstrates two interesting internals of *RbG*. First, the `if` statements used to assign different values to variable `f` are transformed into a decision table (or tabular expression). Following the annotations for variable `f` (line 21), the resulting decision table is not contained in the generated documentation. Instead, *RbG* substitutes `f` in the statement in line 31 with the extracted decision table resulting in a decision table for variable `w`, i.e. symbol ω. Users may specify classical mathematical notation for decision tables by changing the annotation in line 20 to `@symbol[decbracket] \omega` resulting in following representation:

$$\omega = \begin{cases} 2 \cdot \pi \cdot 50 & \alpha < 1.57 \wedge t < 0.1 \\ 2 \cdot \pi \cdot 30 & \alpha < 1.57 \wedge t < 0.2 \\ 2 \cdot \pi \cdot 20 & \alpha < 1.57 \\ 2 \cdot \pi \cdot 50 & \end{cases}$$

While small (in terms of different cases) decision tables are more comprehensible in mathematical notation, tables containing hundreds of entries are more comprehensible in table notation. In particular, the splitting of conditions into table columns according to contained symbols facilitates readability.

The annotations in lines 39-41 command *RbG* to generate a function plot for the C function `i`. For this, the user has to provide values for function parameters, whereas one parameter is mapped to the x-axis of the diagram requiring the specification of a value range. *RbG* generates the data series for the function plot by means of an interpreter that interprets extracted formulae and decision tables (instead of interpreting the original source code). To be more precise, the interpreter works on the *Documentation Model* shown in Fig. 1.

Furthermore, the example shows differences between program code and generated formulae on the level of typesetting. The extraction requires the following transformations of *identifiers* in program code into *symbols* used in the generated document: `RL` $\rightarrow \frac{R}{L}$, `pi` $\rightarrow \pi$, `I_max` $\rightarrow I_{max}$, `w` $\rightarrow \omega$, and `alpha` $\rightarrow \alpha$. All other identifiers appear in the extracted formula as written in program code. Program annotations in lines 19 and 20 show how users can explicitly specify this transformation. Annotation syntax follows LaTeX notation, in the same way as the built-in transformation of variable identifiers containing Greek letters (`pi` and `alpha`) or underscore character, which is interpreted as subscript (e.g. `I_max` $\rightarrow I_{max}$).

Besides the transformation of program statements into documentation, *RbG* also processes annotations used to include texts and images. The example code given in Listing 1 contains several `@p` annotations that are transformed into text paragraphs and inserted in the resulting documentation model.

IV. Evaluation and Discussion

A. Evaluation

In this section we briefly report on industrial application of *RbG* and answer to what extent manual adaption of program sources is required to generate an optimal documentation. Table I lists proprietary engineering and scientific software from industry for which *RbG* is used to generate documentation. For all programs, number of source files, SLOC, size of documentation in pages, and measured duration in milliseconds are given. Evaluations have been performed on a PC with a Core i7-3770 CPU with 3.4 GHz and 8GB RAM.

TABLE I
List of selected programs

Program	Files	SLOC	Language	Pages	Performance
CoreGrad	5	89	Fortran 90	9	2520 ms
WindGrad	8	253	Fortran 90	3	1794 ms
IronLoss	1	496	Fortran 90	9	2285 ms
ContactSurf	1	511	C++	6	2071 ms
MaxIT	16	1467	Fortran 77	7	1904 ms
CCLoss	22	1592	Fortran 77	56	4008 ms
AsmCost	27	4318	Fortran 77	78	7135 ms

Programs are written in C++ and Fortran. The programs heavily vary in terms of source lines of code (SLOC), size of the generated documentation and run-time performance. This is due to the fact that programs *CoreGrad*, *WindGrad*, *IronLoss*, and *ContactSurf* comprise only source code containing calculations relevant for technical documentation. For all other programs relevant calculations are spread over all source files, which hampers separation of files of interest from the rest. Therefore, all source files are input for documentation generation. If *RbG* is applied without prior manual adaption of program sources, structure and content of the generated documentation largely resemble a mapping of assignment statements in source code to formulae and decision tables in generated documentation. Table II lists annotations, which are used in order to generate an optimal documentation with respect to content and structure. The optimum is thereby defined by engineers who originally developed the programs.

TABLE II
Usage of annotations in selected programs

	Symbol Info.		Ext. Control		Structure	
Program	Total	Coverage	Sub.	Supp.	Par.	Sections
CoreGrad	33	67,4%	11	2	9	5
WindGrad	108	91,5%	2	0	21	11
IronLoss	19	67,9%	14	4	10	1
ContactSurf	57	93,5%	0	0	0	0
MaxIT	64	12,3%	10	1	12	6
CCLoss	48	1,6%	72	1	90	66
AsmCost	244	13,04%	1	31	150	54

Table II shows that in order to achieve an optimum in terms of content and structure of the generated documentation adaption mechanisms are applied to control the *RbG*. For programs which comprise source code containing relevant calculations only, more than two thirds of program variables are annotated with `@symbol` information (see column *Coverage*). Similar results are observed for the substitution and suppression of program statements. Except of *ContactSurf* all programs use `@p` (column *Par.*) and `@section` (column

Sections) annotations to provide additional descriptions and structure for the final documentation.

B. User Feedback

The following feedback was collected from users at *Siemens Weiz*. Currently, *RbG* is used in 10 Fortran software projects to generate up-to-date documentation for every build of the software. The introduction of *RbG* is an ongoing process and will be extended to more programs within the *Siemens Weiz* build server environment. For writing *new* code, *RbG* is used to immediately create user-readable documentation in a good quality. Having symbols in the documentation, which are commonly used by the engineers, enormously increases readability. The benefit of the plot feature is twofold. First, it makes documentation more comprehensible and second, it is a good tool during development to see if code behaves as expected. For *existing* code, *RbG* is used to create documentation for selected algorithms. Users value the limited effort of adding tags to source code over analyzing code and writing documentation manually. Important for user acceptance was the integration in the IDE (Visual Studio). For software developers it took a very short time to be able to use *RbG*. Furthermore, stability of *RbG* seems to be a major criterion for user acceptance.

C. Experience from Tool Development

1) Multi-Language Analysis: A principal design decision of *RbG* is to use a *language independent* intermediate representation of program source code based on the ASTM standard [7]. The standard provides modeling elements for almost all language constructs of considered programming languages. Only exotic (legacy) language features such as the *computed goto* statement of Fortran are not covered by the standard. In such a case extension of, or mapping to existing modeling elements is required. To implement static program analysis in a language-independent way additional constraints are required which are not provided by the standard. For instance, the standard makes no assumption about the semantics of switch statements, which is different in C/C++ (*fallthrough*) and Fortran (no *fallthrough*).

2) Open source parsers: The decision for re-using open source parser was, in retrospective, appropriate. C/C++ parsers provided by Eclipse CDT are very robust and accurate. A comparison of seven open source parsers conducted by Janes et al. [9] also concludes with this result. The Open Fortran Parser (OFP) manages parsing of Fortran 90 but has some problems parsing legacy code implemented in Fortran 77 and older. A comparison of OFP with Photran (http://www.eclipse.org/photran/) on industrial legacy code showed that with respect to legacy code issues, Photran has problems on a similar scale.

3) Incremental Tool Development: The development of *RbG* was driven by analysing real programs from the very beginning. Starting with a single language (Fortran 90) and small programs, the feature set was initially rather limited. Based on user feedback provided by domain experts on extracted documentation and the usage and style of annotations, the next iteration was planned and executed. In a typical iteration, we had to add some further language constructs, and, unfortunately, adopt and extend all subsequent analysis steps. However, the advantage was early feedback from domain experts.

V. Conclusion

In this paper we presented *RbG*, a documentation generator for scientific and engineering software. The tool has been applied in different industrial domains such as electrical engineering (e.g. design of power transformers, cost calculation for asynchronous machines) and steel making. The application of *RbG* in these industrial settings showed that *RbG* can be used in different application scenarios, such as reverse engineering and re-documentation of existing (legacy) systems as well as for documentation generation for new software.

Future work is driven by one main challenge. We want to reduce the overall manual effort required (e.g. annotation of program sources) to control the content of generated documentation. We will investigate into heuristics to automatically apply substitution and suppression of program variables and thereby, increase the quality of automatically generated documentation.

Acknowledgment

The authors would like to thank Wolfgang Beer, Walter Hargassner, Christian Huber, Christa Illibauer, Claus Klammer, and Michael Pfeiffer for their contribution and help in developing the *RbG* tool.

References

[1] C. Klammer and J. Pichler, "Towards tool support for analyzing legacy systems in technical domains," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6*, 2014, pp. 371–374.

[2] B. Mösli, "A Comparison of C++, Fortran 90 and Oberon-2 for Scientific Programming," in *GISI 95*, ser. Informatik aktuell, F. Huber-Wäschle, H. Schauer, and P. Widmayer, Eds. Springer Berlin Heidelberg, 1995, pp. 740–748.

[3] J. Pichler, "Extraction of documentation from Fortran 90 source code: An industrial experience," in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8*, 2013, pp. 399–402.

[4] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet, "Extracting business rules from cobol: A model-based framework," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, October 2013, pp. 409–416.

[5] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, Dec. 2010.

[6] M. Moser and J. Pichler, "Static code analysis for domain knowledge extraction from Fortran and C++ software," Software Competence Center Hagenberg, Tech. Rep. SCCH-TR-1467, November 2014.

[7] OMG. (2011, January) Architecture-driven modernization: abstract syntax tree metamodel (ASTM), version 1.0. OMG. [Online]. Available: http://www.omg.org/spec/ASTM/1.0/

[8] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[9] A. Janes, D. Piatov, A. Sillitti, and G. Succi, "How to Calculate Software Metrics for Multiple Languages Using Open Source Parsers," in *Open Source Software: Quality Verification*, ser. IFIP Advances in Information and Communication Technology, E. Petrinja, G. Succi, N. El Ioini, and A. Sillitti, Eds. Springer Berlin Heidelberg, 2013, vol. 404, pp. 264–270.

```c
double sine(double x){
   double sine;    // @symbol sin(x)
   double p, term; // @substitute @suppress
   int N = 100;

// @p The Taylor series for the \emph{sin} function:
   sine = 0.0;
   for(int n=1; n < N; n++){
      p = 2*n + 1;
      term = pow(x, p) / F(p);
      term = pow(-1, n+1)*term;
      sine = sine + term;
   }

   return sine;
}

double i(double I_max, double alpha, double t){
  double RL = 34;         // @symbol \frac{R}{L}
  double w;               // @symbol \omega
  double f, i_a, i_c;     // @substitute @suppress
  double i;
  double pi = 3.141592;
  if (alpha < 1.57) {
    if (t < 0.1) f = 50;
    else if (t < 0.2) f = 30;
    else f = 20;
  }
  else f = 50;
// @p Angular frequency \omega:
  w = 2*pi*f;

// @p Short circuit current:
  i_a = I_max*sin(w*t + alpha);
  i_c = I_max*sin(alpha)*exp(-RL*t);
  i   = i_a + i_c;
  return i;

// @begindiagram Demo Function Plot i
// @plot(I_max=42, alpha=0.89, t=0:0.3)[color=black]{i}
// @enddiagram
}
```

Listing 1. Example program code containing *RbG* annotations.

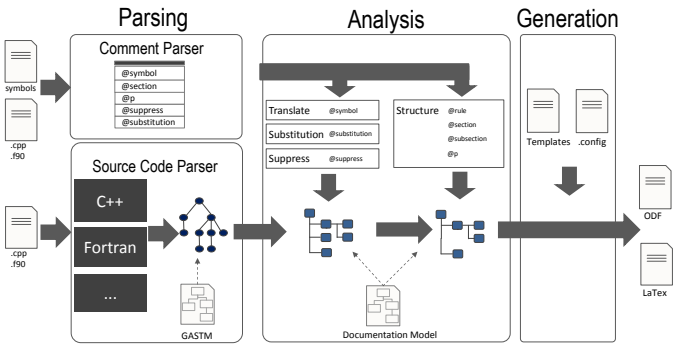

Fig. 1. Architectural overview of *RbG*.

The Taylor series for the *sin* function:

$$\sin(x) = \sum_{n=1}^{n<N} \left(-1^{(n+1)} \cdot \frac{x^{(2 \cdot n+1)}}{F(2 \cdot n+1)}\right)$$

$$\frac{R}{L} = 34$$

$$\pi = 3{,}141592$$

Angular frequency ω:

t	α	ω
$t < 0{,}1$	$\alpha < 1{,}57$	$2 \cdot \pi \cdot 50$
$t < 0{,}2$	$\alpha < 1{,}57$	$2 \cdot \pi \cdot 30$
-	$\alpha < 1{,}57$	$2 \cdot \pi \cdot 20$
-	-	$2 \cdot \pi \cdot 50$

Short circuit current:

$$i = I_{\max} \cdot \sin(\omega \cdot t + \alpha) + I_{\max} \cdot \sin(\alpha) \cdot e^{\left(-\frac{R}{L}t\right)}$$

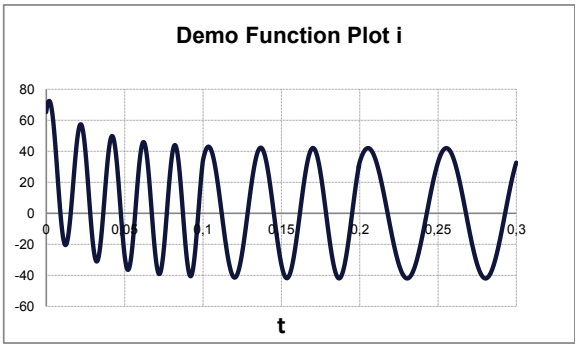

Figure 1: Demo Function Plot i ($I_{\max} = 42.0$; alpha = 0.89)

Fig. 2. Document generated from program code in Listing 1.