eknows: Platform for Multi-Language Reverse Engineering and Documentation Generation

Michael Moser
Software Analytics and Evolution
Software Competence Center Hagenberg
Hagenberg, Austria
michael.moser@scch.at

Josef Pichler
Software Engineering
University of Applied Sciences Upper Austria
Hagenberg, Austria
josef.pichler@fh-hagenberg.at

***Abstract*—Software documentation is an asset for many activities in maintenance and evolution of software. To alleviate the problem of outdated or lost documentation and to improve program understanding, reverse engineering tools and documentation generators have been developed and used during development, maintenance, and evolution of software systems. In this paper, we report on the development of six tools that address different programming languages (e.g. COBOL, Java, and C), problem domains (e.g. banking, insurance, and engineering), and SE activities (e.g. maintenance and migration). While the development of these tools was driven by domain-specific requirements, an architecture that supports reuse of components for the analysis and visualization of software from different programming languages was pursued from the beginning. This resulted in a polyglot software platform for building reverse engineering tools and documentation generators. The software platform provides a modular set of programming language front-ends, static program analysis components and document/diagram generators. To evaluate effectiveness of the platform for tool creation, we conducted an industrial case study and discuss results on reuse potential, adoption of new languages and usage of a generic intermediate representation.**

***Index Terms*—reverse engineering, documentation generation**

I. Introduction

Program comprehension is a core activity in maintenance and evolution of software. To improve program comprehension, reverse engineering tools and documentation generators are used during development, maintenance and evolution of software systems. Reverse engineering tools are software tools and components that support the process of analyzing a system to identify the system's components and their interrelationships and of creating representations of the system in another form or at a high level of abstraction [1]. Similar, documentation generators support the redocumentation of a system, this is the creation or revision of a semantically equivalent representation within the same relative abstraction level [1].

Tools analyse source code statically and dynamically monitor program execution. Many tools and approaches [2] [3] [4] [5] [6] [7] have been proposed by the scientific community and successfully implemented in industry. Obviously, the high number of tools comes from different technology stacks to be supported and the specific information needs stakeholders of a software system have.

Effort to provide tooling for different programming languages is significant. The main reason is due to the various languages, dialects, and compiler-specific extensions or locally developed constructs used in (legacy) software systems [8]. Furthermore, custom, focused, domain-specific solutions are required to support program understanding and evolution of software through software documentation.

To efficiently create reverse engineering tools we propose the software platform *eknows* that supports the creation of these tools, by reuse of prefabricated parsing, analysis and visualization components. Cornerstone of the platform implementation is a generic programming language independent representation (ASTM) of source code that can be reused across analysis tools. The design and development of the *eknows* software platform was driven by several projects that developed tools to support software maintenance and evolution. Six of these tools form the basis for the case study presented in this paper.

It is of interest to what extent language and technology independent representations facilitate reuse of analysis components and ease the creation of tools. Therefore, we raise research questions which address the reuse potential of platform components (RQ-1), suitability of the chosen generic intermediate representations (RQ-2), the construction of new programming language front-ends (RQ-3), and categorize additional effort in tool construction (RQ-4).

To answer these questions we summarize application of *eknows* in six different application contexts. The different cases have been conducted over the course of five years and document the application of *eknows* in various domains, i.e. power engineering, finance, public health, metallurgy, scientific software, mechanical engineering and automation industry, targeting different technology stacks, including Java, .NET, Fortran, C++, COBOL, and ST (IEC 61131-3 Structured Text). Results show that *eknows* facilitates reuse of analysis components across technology stacks and use cases.

Section II presents related work compared to *eknows*, which is introduced in Section III. Section IV describes the research design of our case study, followed by the description of six cases in Section V. In Section VI we evaluate our case study and answer the research questions. Sections VII concludes this paper.

II. Related Work

eknows as a multi-language platform for reverse engineering and documentation generation is related to work on reverse engineering tools (A), documentation generators (B), and multi-language parsing facilities (C).

A. Reverse Engineering Tools

The development of reverse engineering tools was already an active research field decades ago from which many tools (e.g. Rigi [6], GUPRO [4], Moose [5], and MoDisco [7]) have emerged. Rigi can be considered as one of the first reverse engineering tools with high impact both in the research community as well as in the industrial adoption of such tools. The Rigi environment is a mature research tool that provides functionality to reverse engineer software systems. With Rigi large systems can be analyzed, interactively explored, summarized, and documented [6]. GUPRO is an integrated reverse engineering workbench supporting multiple program analysis techniques [4]. Moose is a reverse engineering environment that supports multiple programming languages and focuses software analysis and visualization. Moose provides a generic, extendable user interface and has been applied in different research projects. MoDisco is an Eclipse-based framework implementing a model-driven approach for reverse engineering.

Tools provide generic reusable components and user interfaces, which support users in the task of program comprehension. Users are typically experts in software development, i.e. software engineers or software architects. All these tools have in common that they follow the *Extract-Abstract-View-Metaphor* [3] that can be considered as reference architecture for these reverse engineering tools.

For source code parsing, these tools use language-specific parsers built traditionally with compiler-generator tools. For instance, Rigi's C and COBOL extractors are parsers built with the help of a the parser generator Yacc. *eknows* is implemented in the Java programming language and reuses existing language parsers if available as Java library. If no parser library is available, we generate parsers based on compiler-generators CoCo/R [9] and ANTLR [10]. Generated parser, however, cannot be considered language complete but were implemented with a level of detail driven by the initial reverse engineering project. All these parser-based approaches have in common that they are brittle in the sense that they easily break in the face of code anomalies such as syntax errors, dialects, and embedded languages [11].

A central point of each of these tools is the generic representation of parsed content, which follows different approaches. Rigi and GUPRO for instance implement a generic graph-based representation facilitating graph-theory not only for the representation but also for graph queries, graph transformations, and graph editors / visualizations. For instance, the graph editor is the heart of the Rigi system. MoDisco follows a standard-based approach and leverages OMG standards such as KDM and ASTM [12] meta-model for the representation of source code. *eknows* also implements ASTM for the representation of source code in form of a generic abstract syntax tree.

B. Documentation Generators

Classic documentation tools for generating documents from (annotated) source code were also developed and used decades ago. For instance, SOFT-REDOC [2] is a tool to support isolating and extracting business rules from COBOL source code. Not only does it extract business rules but it also generates a data dictionary with the references to each data item, a procedure tree that depicts the structure of the component parts of a program, and a decision tree that represents the conditional logic of each program. The tool Meta [8] was built for the redocumentation of a legacy banking system and specially tailored to the organization's need.

Tools such as JavaDoc and Doxygen are well-known and widely used for generating API documentation based on annotated source code. JavaDoc is part of the Java Development Kit[1] (JDK) and a representative of a language-specific tool. Doxygen[2] is the defacto standard tool for generating documentation from annotated C++ source code, but also supports other popular programming languages such as C, C#, Java, Python, and many more. In addition, Doxygen supports the generation of diagrams (e.g. call graphs, include graphs, UML class diagrams) beyond pure API documentation. Current approaches in the scientific community turn towards on-demand-documenation (ODD) [13] generation and try to generate documentation from various data sources, e.g. system implementation, issue tracking systems, test cases, or online discussion forums, as response to user a query.

C. Multi-Language Parsing

Although reverse engineering tools and documentation generator tools may include multi-language parsing facilities, we also find multi-language parsing as stand-alone tools. For instance, srcML [14] facilitates the analysis and manipulation of source code via an XML format for source code. The provided multi-language parsing tool converts source code into srcML format for Java, C#, C++ and C. *srcML* provides a one to one mapping of source elements of a given language into *srcML* elements. For instance, C# attributes and Java annotations are mapped to different *srcML* elements (i.e. <annotation> and <attribute>). This clearly, differentiates *srcML* from the presented approach.

III. Software Platform *eknows*

eknows is a Java-based software platform to build reverse engineering tools and documentation generators. The platform provides a modular extensible set of software components, cf. Fig. 1, that facilitate the rapid development of tools in program comprehension, documentation generation and software reverse engineering. Support for multiple programming languages in terms of language-specific extraction components and language-independent analysis is a key feature of the

[1]https://www.oracle.com/java/technologies/

[2]https://www.doxygen.nl/index.html

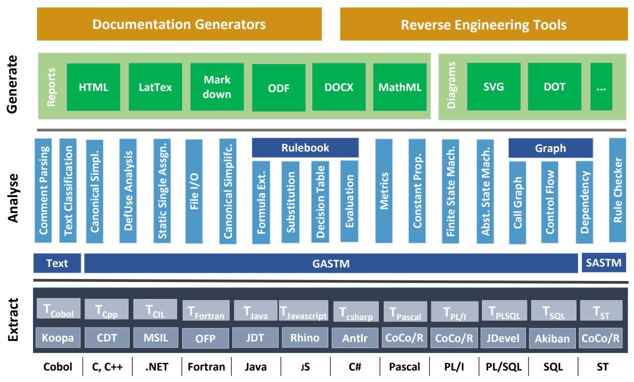


Fig. 1. Overview *eknows* platform components

TABLE I
LANGUAGES SUPPORTED BY EKNOWS.

Language	Version	Parser Tech.	SLOC
C, C++	C++17	Eclipse CDT[3]	3 348
C#	C# 7.0	ANTLR[4]	3 618
CI/.NET	.NET 4.5	MSIL	3 233
COBOL	COBOL 85	Koopa[5]	4 683
Fortran	77/2003	Open Fortran[6]	3 969
JCL	JCL z/OS 2.2	CoCo/R[7]	1 560
Java	Java 8	Eclipse JDT[8]	2 748
Javascript	ES 6	Mozilla Rhino[9]	1 618
Natural	v 4.2.6	CoCo/R	2 345
Pascal	Pascal 7.0	CoCo/R	989
PL/I	z/OS 4.1	CoCo/R	2 450
PL/SQL	9.1	JDeveloper/Akiban[10]	781
SQL	MySQL 5.6	Akiban	1 500
ST	IEC 61131-3	CoCo/R	3 729

platform. Tools based on *eknows* may target different or even multiple programming languages.

The platform comprises 350K source lines of code (SLOC), cf. Table X, and provides reusable components that facilitate (1) language parsing and transformation of source code into a generic abstract syntax tree (AST), (2) structural and behavioral analysis of software, (3) reporting and visualization of analysis results. Tools built on top of *eknows* integrate required software components as is and add functionality required for a specific use cases. *eknows* provides support for main reverse engineering activities [15] extraction, transformation, analysis and visualization as depicted in Fig. 2.

A. Exraction Components

Extraction components extract information from software systems and provide structured access via abstract syntax trees and data structures used in analysis. *eknows* extraction components, also referred to as language front-ends, target program source code and source code comments. A language front-end fulfills the following tasks: (1) parse program source code and source code comments and (2) transform parse trees into abstract syntax trees (AST). To reuse analysis components across different programming languages and technology stacks a language independent, generic abstract syntax tree model is used. Language front-ends are created for each programming language supported by *eknows*. Table I lists all programming languages supported.

To provide robust and update-to-date parsing infrastructure *eknows* reuses freely available parsing components as far as possible. For instance to parse Java or C++ parsing, Eclipse JDT and Eclipse CDT are used. If no ready-to-use source code parser is available, parser generators (i.e. ANTLR and CoCo/R) are used to generate parsing infrastructure from context-free grammar specifications. Reused parsing technology significantly differ in completeness and parsing strategies.

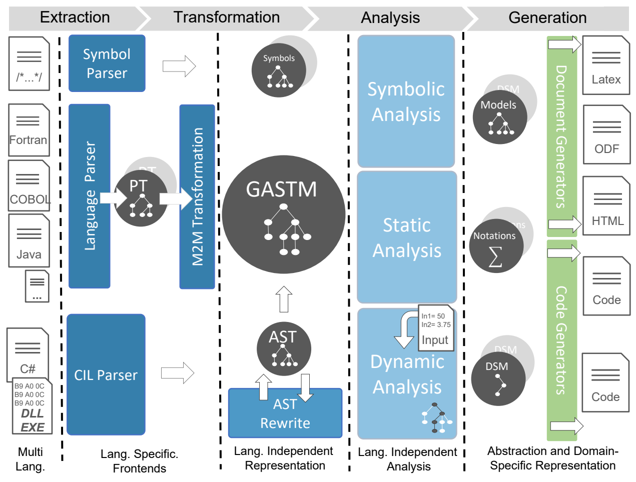


Fig. 2. Reverse engineering activities supported by *eknows*.

B. Language-independent Representation

eknows is a polyglot software platform, meaning that it supports the creation of tools targeting different implementation technologies. To reuse analysis components across different technology stacks *eknows* builds upon the Abstract Syntax Tree Metamodel (ASTM) [12]. The standard provided by the Object Management Group (OMG) is used as common intermediate representation and as exchange format. The ASTM establishes a specification for abstract syntax tree models. To provide for uniformity as well as a universal framework for extension, ASTM is composed of the Generic Abstract Syntax Tree Metamodel (GASTM) and a set of complementary, language-specific specifications, called the Specialized Abstract Syntax Tree Metamodels (SASTM). GASTM implementation used by *eknows* is generated using the Eclipse Modeling Framework (EMF).

[3]https://www.eclipse.org/cdt
[4]https://www.antlr.org/
[5]https://github.com/krisds/koopa
[6]https://github.com/OpenFortranProject/open-fortran-parser
[7]https://ssw.jku.at/Research/Projects/Coco
[8]https://www.eclipse.org/jdt
[9]https://github.com/mozilla/rhino
[10]https://github.com/brunoribeiro/sql-parser

TABLE II
ANALYSIS COMPONENTS PROVIDED BY EKNOWS.

Description	Input Rep.	Lang spec.	PLs
Call graph analysis	GASTM	generic	all
Canonical rep. of GASTM statements	GASTM	generic	all
Canonical simplification of formulas	Specific	generic	all
Cobol-based file IO analysis	Specific	specific	COBOL
Code comment classification	Text	generic	all
Constant propagation through code	GASTM	generic	all
Control-flow graph analysis	GASTM	generic	all
Data flow analysis for db updates	Specific	specific	(PL/)SQL
Database read/write analysis	GASTM	specific	Java
Decision table construction	Specific	generic	all
Elimination of GOTO statements	Specific	specific	Fortran
File input/output analysis	Specific	specific	C++
Finite state machine for statements	GASTM	generic	all
Function inlining	GASTM	generic	all
GASTM based metric calculation	GASTM	generic	all
Grammar inference engine	GASTM	generic	all
Java Persistence API usage analysis	Specific	specific	Java
Math. formula extraction from code	GASTM	generic	all
Parameter data flow analysis	GASTM	generic	all
Rule-based code consistency checks	Specific	specific	ST
SMT-based array access checks	GASTM	generic	all
Static single assignment	GASTM	generic	all
Statistical AST evaluation	GASTM	generic	all
Symbolic execution engine	specific	generic	all
Symbolic variable substitution	GASTM	generic	all
Text extraction from code	Text	generic	all
Usage analysis on AST nodes	GASTM	generic	all
Value ranges analysis	GASTM	generic	all

eknows strives to reuse analysis components as far as possible. Since language-specific extensions need to be reflected in analysis components, usage of language specializations, in the sense of SASTM extensions, are kept to minimum. Therefore, we aim to create abstract syntax trees on GASTM only, i.e. language-independent models. This requires mapping of different language elements to language-neutral elements, which is challenging in cases when GASTM does not provide corresponding elements.

C. Analysis Components

Analysis components provided by *eknows* are listed in Table II. Components mostly use GASTM as intermediate representation. Additional analysis data structures building on GASTM are frequently used (specific) for more complex analysis, e.g. symbolic execution, or base analysis on text only (e.g. comment extraction). Generally, analysis components are generic in the sense that analysis can be applied to source code of all supported languages. To a certain extent analysis components are provided to selected languages only. For instance, to analyse technology specific APIs, e.g file I/O or database manipulations, analysis are intended/implemented for specific languages only.

D. Visualization Components

eknows provides a set of components (see Table III) to create visualizations and reports from analysis results. Visualizations generate text documents and graphical result presentations. Generated elements fall into the categories text, mathematical formula, tables, charts, or graphs and are aggregated to reports and documents. Input for visualization components are ASTM data structures or specific result data structures created by analysis components. These are basically generic definitions of graphs (e.g. vertices and edges) or abstractions over data used in charting components. Graphs are output in intermediate format, e.g. Graphviz DOT. To generate documentation a format independent data structure is used to specify document structure and content (e.g. sections, paragraphs, formulas, figures or tables). Document specifications can be output as LaTeX, HTML, Markdown, and Open Document Format.

TABLE III
VISUALIZATION COMPONENTS AND REPORT GENERATORS.

Description	Input	Formats
Word document generation	rulebook	odt, docx, pdf
Latex document generation	rulebook	tex, pdf
MathML formulae rendering	rulebook	mml, html
HTML document generation	rulebook	hmtl
Markdown document generation	rulebook	md
Call graph export writer	graph-ds	dot
Control-flow graph export writer	graph-ds	dot
Syntax tree export writer	graph-ds	dot
Metric table export writer	astm	csv

eknows does not provide ready-to-use viewer components integrated within an analysis workbench. In this sense, *eknows* differs significantly from related platforms like Rigi or Moose. Generated visualizations present information in static fashion, interactivity is mostly added by a specific tool which integrate generated artefacts and result data into custom documentation environments (e.g. web applications) and reports.

IV. RESEARCH DESIGN

We conduct a multiple industrial case study to investigate capability of the eknows platform across different domains, technologies and use cases. It is the overall objective of the study to evaluate reuse potential across different technology stacks and the platforms suitability to build new reverse engineering tools. To answer these questions the study follows a research process as shown in Runson et al. [16].

A. Research Questions

The following research questions have been used to guide this study. Generally, questions address the reuse potential of platform components, the construction of new programming language front-ends, suitability of the chosen intermediate representations, and additional effort for tool construction.

- *RQ-1:* Reuse: Does the platform facilitate reuse of components across domains and technology stacks? Which components are reused?
- *RQ-2:* Intermediate representation: Does ASTM provide a sufficiently rich basis for language reuse? Which extensions are typically made when new languages are adopted?
- *RQ-3:* Parsing front-end construction: To what extent are *complete* parsing front-ends required and how rich are created GASTM models?

- *RQ-4:* Tool construction: How much tool-specific code is required? What is it needed for?

B. Selection of Cases and Units of Analysis

The cases in this study are reverse engineering tools, which were built on top of the *eknows* platform. Selection of cases of analysis was mainly driven by the following criteria: (1) tools were developed for an industrial application context, (2) main purpose of tools is reverse engineering or documentation generation, (3) full access to source code of tools and analyzed program sources determining development of tools. Units of analysis are latest versions of tool implementations and the *eknows* platform.

C. Data Collection

Data was collected from project and source code repositories. Both, the tool implementations and *eknows* platform source code were collected to answer research questions. Although, tools were constructed over a time span of five years, analysis presented in this paper uses data from current *eknows* version (march, 2021) and latest release versions of tools. To address research question related to the evolution of *eknows* (e.g. modification of ASTM) project data from issue management systems and versioning systems was evaluated.

V. Cases

In the following we describe six reverse engineering tools and documentation generators which are the cases for this industrial case study. Table IV lists tool details covered in this report.

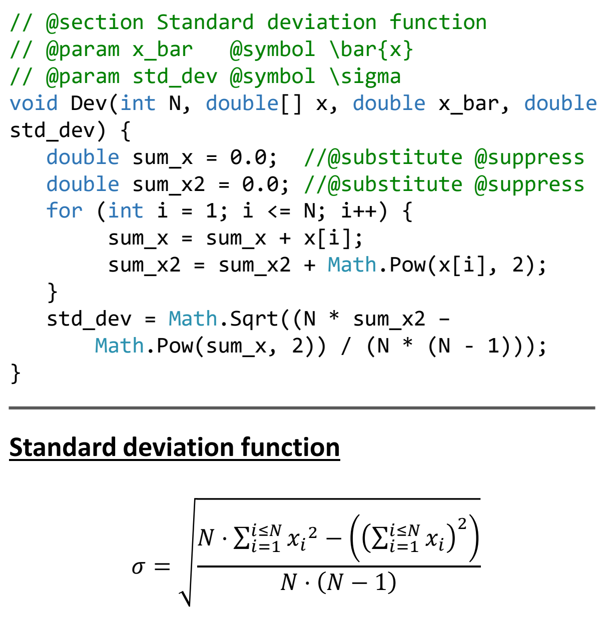
A. Case: RbG

1) Tool Summary: RbG [17] generates high quality engineering documentation, so called *rulebooks*, from Fortran, C++ and C# source code. Generated documentation contains mathematical formulae, decision tables, and function plots, which are automatically extracted from annotated program sources.

2) Problem Statement: Maintenance and evolution of software in engineering domains requires extensive documentation of domain knowledge. Domain knowledge often includes descriptions of mathematical or physical ground theories together with numeric algorithms that solve this problem. Solutions applied in source code reflect ground theories, however adapt formulae for a specific context. Moreover, documentation is regarded as the specification of program behaviour that needs to be approved by principal engineers. Domain experts, which do not necessarily have programming expertise, use the documentation to understand and reproduce program behavior. The co-existence of source code and documentation implicates high writing effort and inconsistency by itself and with respect to documented sources.

TABLE IV
Overview of industrial application of *eknows* tools.

Name	Domain	Languages	Activity
RbG	Power Transformer	Fortran, C++, C#	Maint.
Varan	Steelmaking	C++	Maint.
ReDoc	Health Care	COBOL, Java, SQL	Migration
PlSql-Doc	Steelmaking	PL-SQL	Migration
Trustscore	Machine Building	C, Structured Text	Maint.
Finance-C	Finance/Insurance	COBOL, PLI, Natural	Maint.



```
// @section Standard deviation function
// @param x_bar   @symbol \bar{x}
// @param std_dev @symbol \sigma
void Dev(int N, double[] x, double x_bar, double
std_dev) {
   double sum_x = 0.0;  //@substitute @suppress
   double sum_x2 = 0.0; //@substitute @suppress
   for (int i = 1; i <= N; i++) {
        sum_x = sum_x + x[i];
        sum_x2 = sum_x2 + Math.Pow(x[i], 2);
   }
   std_dev = Math.Sqrt((N * sum_x2 -
       Math.Pow(sum_x, 2)) / (N * (N - 1)));
}
```

Standard deviation function

$$\sigma = \sqrt{\frac{N \cdot \sum_{i=1}^{i \leq N} {x_i}^2 - \left(\left(\sum_{i=1}^{i \leq N} x_i\right)^2\right)}{N \cdot (N-1)}}$$

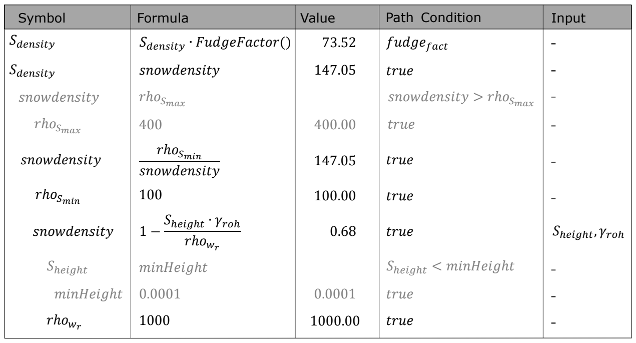
Fig. 3. RbG: Formula extraction from C# source code.

3) Solution: To break up separation between source code and documentation *RbG* automates the generation of documentation directly from program sources. The tool transforms source code into mathematical formulas, decision tabels, and function plots. The tool provides a set of annotations that are used in source code comments to control the extractions of formulae on a fine-grained level and to define documentation structure and content within source code files. Fig. 3 shows the usage of `@subsitute` to replace usage of variables with its last definition, `@suppress` to remove statements from documentation, `@symbol` to replace variable identifiers with symbol names, and `@section` to structure documentation.

B. Case: VARAN

1) Tool Summary: VARAN [18] supports program comprehension of process models used to control desulfurization in the steel making process. By integration of static and dynamic analysis methods the tool generates documentation that shows the step-wise computation of relevant model results.

2) Problem Statement: Implementation of models used to control steelmaking processes are complex and complicated to comprehend. If the effect of model input values on model results is not clearly documented any change to source code becomes critical. Stakeholders need to know (1) which model results are produced from a given entry method, (2) what formulas are applied to calculate the value and (3) which input parameter affect which model result.



Symbol	Formula	Value	Path Condition	Input
$S_{density}$	$S_{density} \cdot FudgeFactor()$	73.52	$fudge_{fact}$	-
$S_{density}$	$snowdensity$	147.05	$true$	-
$snowdensity$	$rho_{S_{max}}$		$snowdensity > rho_{S_{max}}$	-
$rho_{S_{max}}$	400	400.00	$true$	-
$snowdensity$	$\frac{rho_{S_{min}}}{snowdensity}$	147.05	$true$	-
$rho_{S_{min}}$	100	100.00	$true$	-
$snowdensity$	$1 - \frac{S_{height} \cdot \gamma_{roh}}{rho_{w_r}}$	0.68	$true$	S_{height}, γ_{roh}
S_{height}	$minHeight$		$S_{height} < minHeight$	-
$minHeight$	0.0001	0.0001	$true$	-
rho_{w_r}	1000	1000.00	$true$	-

Fig. 4. VARAN: Dynamic result table for step-wise computation of model result.

3) Solution: Essential output of the generated documentation are tables covering the step-wise computation of a model result. For each model result, the documentation contains one table with table rows for all assignment statements which may influence the computation of a model result. To construct these tables *VARAN* uses static analysis to create use-definition chains from an interprocedural control-flow-graph. To construct the graph, call relations between functions are analyzed and new sub-control-flow graphs are created for each function definition called. To create documentation focused on certain use cases, static analysis is integrated with dynamic analysis that performs flow-sensitive program interpretation. From a given set of concrete input values model executions are calculated and merged with statically computed use-definition chains. By this *VARAN* generates compact documentation focusing only executed program paths as shown in Fig. 4.

C. Case: ReDoc

1) Tool Summary: ReDoc [19] supports the automatic re-documentation of large batch- and online-processing architectures used in public-health industry. The tool facilitates system understanding by documenting inter-procedural control-flow and data-flow on configurable levels of detail of Java and COBOL sources.

2) Problem Statement: Much needed documentation of legacy software is notoriously outdated or missing at all. Upcoming software migration projects, intensify the need for accurate and comprehensible software documentation. However, manual re-documentation of large code bases prior migration is often not feasible and automation the only possibility. The development of *ReDoc* was driven by the following requirements: (1) Fully automated documentation generation, no manual redocumentation on source level. (2) Moving abstraction level of the generated documentation from source code towards business scenarios. (3) Support for different stakeholders, i.e. developers, project managers, business analysts. (4) Document actual implementation state (i.e. COBOL) and target implementation (Java).

3) Solution: Fig. 5 shows HTML documentation generated by *ReDoc*. *ReDoc* generates listings for batch programs, online

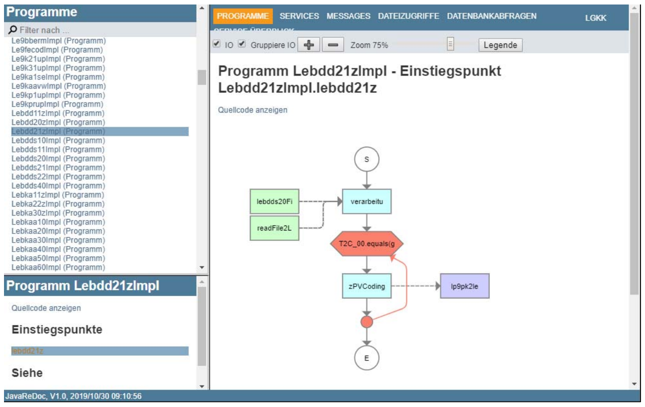


Fig. 5. ReDoc: Control-flow documentation generated by *ReDoc*.

services, messages exchanged between services, and access to files and databases from batch programs.

Behavior of batch programs and services is further described by flow diagrams, cf. Fig. 5. The rigorous transformation from COBOL/Java source code to flowchart diagrams results in flowcharts with thousands nodes for one batch program. Therefore, smart heuristics were implemented which control filtering of nodes. Flow diagrams show control-flow starting from a defined entry-point method and highlight nodes which perform read or write operations to database tables, files, or messages. These I/O operations are aligned in separate swim lanes on the left (input operation) and right (output operation) of program control-flow. Moreover, documentation of messages and file access list usage information and provide description on the precise data formats used. Access to database tables is documented by SQL statements recovered from program sources and XML definitions.

D. Case: PLSQL-DOC

1) Tool Summary: PLSQL-DOC [20] is used to reverse engineer PL/SQL source code into abstract and comprehensive representations, that allow stakeholders to trace data-flow from Oracle reports back to original data sources.

2) Problem Statement: Stored procedures and database triggers often contain substantial amount of procedural logic that capture core business knowledge. New or updated business cases are easily added to a code base. However, logic of outdated business cases is hard to detect. Execution logic is spread across different procedures, triggered by database update operations and may include hundreds of SQL statements performed in sequence with hundreds of auxiliary tables involved. In the project for which *PLSQL-DOC* was originally developed, users need to see the original data source for tables used in Oracle reports and associated verification logic from PL/SQL sources. Moreover, verification logic needed to be resolved so that variables in reports refer to original data sources rather than auxiliary tables.

3) Solution: PLSQL-DOC performs data-flow analysis and symbolic substitution to trace report data. Compact graph representations show the flow of data from original database

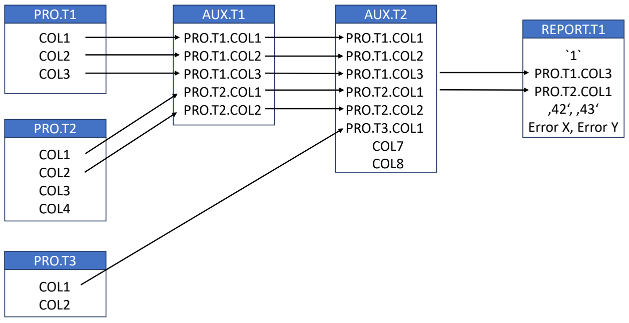


Fig. 6. PLSQL-DOC: Data-flow graph generated by *PLSQL-DOC*.

tables, via auxiliary tables, to reports. Graph representations can be configured to replace names of database tables with names from data-flow analysis as shown in Fig. 6, or to hide auxiliary tables at all. Moreover, listings of error codes and conditions producing the error codes are output using names from original database tables only.

E. *Finance-C*

1) Tool Summary: Finance-C was developed for a financial service provider and generates technical and functional documentation that satisfies regulatory obligations setup by national financial supervision authorities.

2) Problem Statement: Companies in financial service industries are required to create and maintain system documentation that meet requirements defined by financial supervision authorities. Documentation must reflect systems at different levels including user documentation and training material, functional documentation that contains business rules, and technical documentation. Providing up-to-date documentation of business rules and technical implementation takes significant effort. The initial project context *Finance-C* was originally developed for, is a COBOL/IMS/DB2 legacy system for leasing with several million lines of code. Manual documentation effort to fulfill regulative requirements was estimated to be > 25K PT.

3) Solution: Finance-C supports development departments in the creation of documentation of business rules and technical system implementation. Fig. 7 shows an excerpt of business rules which contain the computation of domain concepts, e.g. $BASE_{SV}$, together with conditions under which a computation is applied. Business rules are extracted from source code statements by means of formulae extraction and symbolic execution already shown in *RbG* [17]. A major asset of Finance-C are a set of heuristics and machine learning algorithms that are used to classify text extracted from source code comments. Classified comments are used to structure documentation and to further improve readability [21].

F. *Case: Trustscore*

1) Tool Summary: Trustscore generates documentation for PLC programs written in Structured Text (ST) and C programming language. Integral part of documentation are finite state charts derived from switch statements found in source code.

2) Problem Statement: Programmable logic controllers (PLC) are applied in various industrial domains and machinery. PLC execution is based on input and output variables, which map to sensors (input) and actuators (outputs) of the environment the PLC is connected to. PLCs apply a cyclic execution model and read inputs periodically, perform computation and write outputs variable. Comprehension of PLC programs is hard in the sense that computations of output variables in a specific execution cycle influences computation in a subsequent cycles.

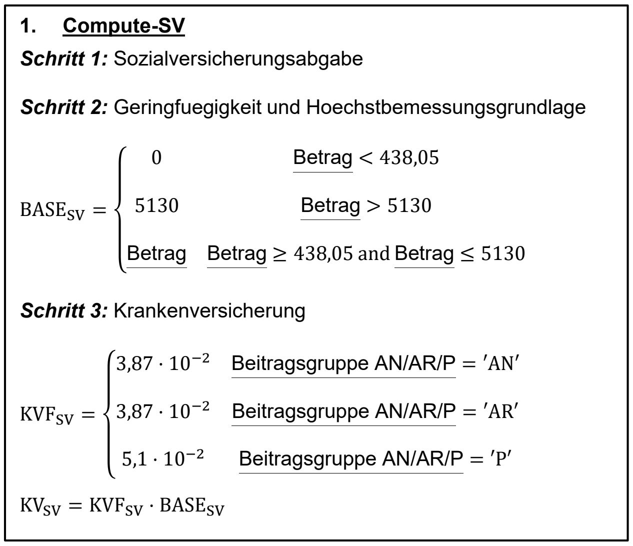
3) Solution: To facilitate comprehension of PLC programs *Trustscore* extracts finite state charts from program sources. The tool analyses source code patterns of switch statements in structured text and C functions. Assignments to input or output variables used in switch expression are transformed to finite state charts. The cases in a switch-statement are interpreted as states in a FSM, assignments to variables are considered as state transitions. The tool records conditions within case-statements and lists these within documentation. For each pattern detected within code state diagrams are generated and aggregated with an HTML documentation, cf. Fig. 8.

VI. Evaluation

The overall goal of this study is to investigate the ability of the *eknows* platform to support tool creation across different problem domains, technology stacks, and use cases, cf. Table V. Therefore, we discuss below (1) the reuse of platform components by tools, (2) the suitability of generic abstract representations for the tasks presented, (3) the construction of language front-ends for a polyglot software platform, (4) and the additional effort required for tool construction in general.

A. *RQ-1 Reuse Potential*

We study to which degree *eknows* components are reused across tools. Table V groups *eknows* components by main



1. <u>Compute-SV</u>

Schritt 1: Sozialversicherungsabgabe

Schritt 2: Geringfuegigkeit und Hoechstbemessungsgrundlage

$$\mathrm{BASE_{SV}} = \begin{cases} 0 & \underline{\text{Betrag}} < 438{,}05 \\ 5130 & \underline{\text{Betrag}} > 5130 \\ \underline{\text{Betrag}} & \underline{\text{Betrag}} \geq 438{,}05 \text{ and } \underline{\text{Betrag}} \leq 5130 \end{cases}$$

Schritt 3: Krankenversicherung

$$\mathrm{KVF_{SV}} = \begin{cases} 3{,}87 \cdot 10^{-2} & \underline{\text{Beitragsgruppe AN/AR/P}} = \text{'AN'} \\ 3{,}87 \cdot 10^{-2} & \underline{\text{Beitragsgruppe AN/AR/P}} = \text{'AR'} \\ 5{,}1 \cdot 10^{-2} & \underline{\text{Beitragsgruppe AN/AR/P}} = \text{'P'} \end{cases}$$

$$\mathrm{KV_{SV}} = \mathrm{KVF_{SV}} \cdot \mathrm{BASE_{SV}}$$

Fig. 7. Finance-C: Documentation of domain concept "Krankenversicherung" by *Finance-C*.

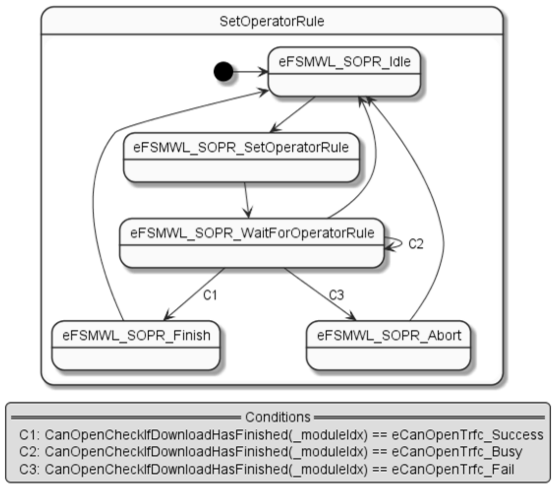

SetOperatorRule
eFSMWL_SOPR_Idle
eFSMWL_SOPR_SetOperatorRule
eFSMWL_SOPR_WaitForOperatorRule
C2
C1
C3
eFSMWL_SOPR_Finish
eFSMWL_SOPR_Abort
Conditions
C1: CanOpenCheckIfDownloadHasFinished(_moduleIdx) == eCanOpenTrfc_Success
C2: CanOpenCheckIfDownloadHasFinished(_moduleIdx) == eCanOpenTrfc_Busy
C3: CanOpenCheckIfDownloadHasFinished(_moduleIdx) == eCanOpenTrfc_Fail

Fig. 8. Trustscore: State chart diagram extracted from C and Structured Text.

reverse engineering activities and checks usages of single components by tools of the presented cases. Table VI aggregates the single usages and gives the percentage of components reused per reverse engineering phase. Obviously, *eknows* platform reuse is given by all tools, but to varying degrees. While tools like *Finance-C* or *RbG* reuse many components, *Plsql-Doc* uses just 4 components.

Reuse of components addressing *analysis* phase is significantly higher, compared to extraction and generation components. Within analysis components we see, that the more general a component is the more likely it is to be reused, e.g. call graph analysis or dependency analysis compared to the elimination of goto statements. All six tools require at least two extraction components of the platform. *Finance-C* which address the (legacy) mainframe technology stack used in finance sector, even six extraction components.

It is the general observation, that high reuse is triggered by tool requirements with respect to output and analysis, rather than requirements on supported programming languages. For instance, both *Finance-C* and *RbG* have similar requirements, e.g. documentation of source code on fine-grained level using mathematical notation and decision tables. The cases show that reuse of analysis components is possible, even if different programming language, with the exception of C/C++, are addressed. However, high reuse of analysis components does not always yield high reuse of components on generation level. This is due to specific requirements on result visualisation which is highly determined by the domain-specific notations and use-cases. For instance, *Varan* requires similar analysis components than *RbG*, however to fulfill requirements and show step-wise computation of input variables a tool-specific presentation layer needed to be implemented.

B. RQ-2: Generic Intermediate Representation

As *eknows* is built on OMG's ASTM, the question is not only whether an AST representation is suitable for representing multi-language software but also whether the specific OMG meta-model is suitable for this task. Even this standard exists for years and is used by some approaches (e.g. MoDisco), the answer to this question remains open so far. Even though ASTM foresees language-specific extensions from the very beginning (by separation of GASTM and SASTM), *eknows* was built with the principal design decision

TABLE V
REUSE OF EKNOWS COMPONENTS ACROSS TOOLS

	Component	Fin-C	RbG	ReDoc	Plsql-Doc	Varan	Trustscore
Analysis	Canonical simplification of GASTM	✓	✓			✓	
	Code comment classification	✓					
	Control-flow graph	✓	✓			✓	✓
	Call-graph analysis	✓	✓			✓	✓
	COBOL-based file IO analysis	✓					
	Constant propagation through code	✓		✓			
	Canonical simplification of formulas	✓	✓			✓	
	DFA read/write analysis			✓			
	Decision table construction	✓	✓				
	File input/output analysis	✓				✓	
	Finite state machine for statements						✓
	Function inlining refactoring		✓				
	Elimination of GOTO statements		✓				
	Java Persistenc API usage analysis			✓			
	GASTM based metric calculation	✓					✓
	Math. formula extraction from code	✓	✓			✓	
	Dependency analysis	✓	✓			✓	✓
	Parameter data flow analysis	✓	✓			✓	✓
	Rule-based consistency checks						✓
	Data flow analysis DB2 updates				✓		
	Static single assignment	✓	✓	✓			
	Statistical AST evaluation	✓	✓	✓			
	Symbolic execution engine					✓	
	Text extraction from code	✓		✓			
	Analysis of value ranges	✓	✓				
	Symbolic variable substitution	✓	✓			✓	
Extraction	COBOL	✓		✓			
	C, C++	✓	✓			✓	✓
	C#		✓				
	.NET CIL		✓				
	Fortran		✓				
	Java			✓			
	Job Control language	✓					
	Natural	✓					
	PL/I	✓					
	PL/SQL				✓		
	SQL	✓		✓	✓	✓	
	Structured Text						✓
Generation	Word document report generation	✓	✓				
	Latex document report generation	✓	✓				
	MathML formulae rendering	✓	✓			✓	
	HTML document report generation	✓	✓				
	Markdown document generation	✓	✓				
	Call graph export writer	✓	✓		✓		✓
	Metric table export writer		✓	✓			✓

TABLE VI
REUSE FOR REVERSE ENGINEERING PHASES AND TOOLS

Reuse	Fin-C %	RbG %	ReDoc %	Plsql Doc %	Varan %	Trust-score %
Analysis	64,3	50,0	21,4	3,6	32,1	25
Extraction	42,9	28,6	21,4	14,3	14,3	14,3
Generation	66,7	77,8	11,1	11,1	11,1	22,2

to minimize the number of language-specific extensions.

Table VIII shows language elements that cannot be mapped to a corresponding model element. Instead, we map language-specific constructs to available GASTM elements. The most important category of such mappings (normalization) concerns language elements that contribute to the control-flow of a program. The Fortran *arithmetic IF* statement that allows conditional jumps depending on the sign of an integer number is one example. Further examples are the *computed GOTO* statement, the COBOL *EVALUATE* and *TIMES* statement, and the C# *foreach* statement. A second category of required normalization includes specific language elements that need no specific treatment by an analysis algorithms but as single statement (or expression) without side effects. Such mappings are used for most programming languages (element *Function-Call* in column *Target Element*) but in particular for specific language constructs of the legacy languages COBOL (16), Natural (36), and PL/I (20). A third category is built from modern language constructs such as C# properties and events. The rest are only minor mappings such as the COBOL *MOVE* statement that is transformed into a binary expression with operator type *Assign*. However, due to normalizations, it is not possible to reconstruct the original statement or expression. For reverse engineering and documentation generation tasks, fortunately, this is not a requirement. All these normalizations cause effort during tool construction but are clearly indicators that ASTM is suitable to represent multi-language source code.

Table VII shows aggregated numbers of language constructs that are not normalized but represented by language-specific ASTM extensions. Such extensions are either additional fields of an ASTM model element (e.g. C++ function attributes) or new model elements derived from existing ones (e.g. for Java lambda expressions). Such extensions must be considered harmful because analysis algorithms may treat such extensions as well. Every time when we extend the representation we have to check all analysis algorithms whether the algorithm must be extended as well. With a total of 15 element extensions and 4 field extensions based on 182 total GASTM model elements, we consider this as manageable and, consequently, ASTM suitable to represent multi-language source code.

C. RQ-3: Front-end Construction

Major asset of a polyglot software platform are the front-ends towards different programming languages. Effort for the creation of front-ends is significant and different strategies are applied for construction. Reuse of existing parsing technology as listed in Table I is one cornerstone to limit effort. A second cornerstone is to support programming language features only as far as required by use-cases triggering the creation or evolution of front-ends. This raises the question to which degree feature complete language front-ends are required to fulfill requirements raised by industrial use cases.

TABLE VII
GASTM MODIFICATIONS TRIGGERED BY LANGUAGE ADOPTIONS

Lang.	Kind	No.	Description
C#	add	5	Properties, Logical Def. OR, Access kinds
C++	field	3	Function attributes default, delete pureVirtual
Fortran	add	6	Binary operators (e.g. Power), WithStatement
Java	add	2	Lambdas, MethodReferencExpression
JS	field	1	Name field added to Lambda
ST	add	2	Universal/Exclusive

TABLE VIII
NORMALIZATION: MAPPING LANGUAGE FEATURES TO GASTM ELEMENTS

Lang.	Source Elements	Target Element
C#	Event declaration	ProperteyDefinition
C#	Foreach	ForStatment
C#	Interface definition	ClassType
C#	Null coalescing operator	BinaryExpression
C#	Prop. accessor, delegate, destructor	FunctionDefinition
C#	sizeof, nameof	FunctionCall
Cobol	Accept, Cancel, Close ... (total 16)	FunctionCall
Cobol	Evaluate statement	IfStatement
Cobol	Move	BinaryExpression
Cobol	SubstractionFormat, AddtionFormat	BinaryExpression
Cobol	Times statement	ForStatment
Fortran	Arithmetic IF, Computed Goto	Switch
Fortran	Complex	Literal
Fortran	Cycle	ContinueStatement
Fortran	Pause, Write, Null	FunctionCall
Java	Annotation declaration	NamedTypeDefintion
Java	Interface definition	ClassType
Java	super, assert, synchronized	FunctionCall
Java	this expression	IdentifierExpression
Natural	Abs, Find, Input, ... (total 36)	FunctionCall
PL/I	Bin, Close, Cut, Date ... (total 20)	FunctionCall

TABLE IX
COMPLETENESS OF LANGUAGE FRONT-ENDS

Frontend	Transf. Type	Parser Hooks	Hooks Impl.	Parser Cov. %	Gastm Cov. %
COBOL	MT	272	83	30,5	37,1
C, C++	MT	127	96	75,6	60,2
C#	MT	241	188	78,0	66,7
.NET CIL	Hook	178	103	57,9	30,1
Fortran	Hook	534	133	24,9	46,8
Java	MT	100	76	76,0	47,3
Javascript	MT	165	81	49,1	40,9
Natural	MT	144	73	50,7	32,8
PL/I	MT	331	65	19,6	36,0
PL/SQL	MT	104	21	20,2	29,0
SQL	MT	527	54	10,2	9,2

Table I lists front-ends and sloc of code required to implement transformation of language-specific parse trees to GASTM data structures. In Table IX we further detail GASTM construction from language-specifics ASTs. Column parser coverage shows to which extent parser/language specific transformations are required to build GASTM. For instance, the Open Fortran Parser provides 534 hook methods called during parsing. Currently, 25% of hook methods are implemented in the respective front-end. Coverage across front-ends ranges between 10% for SQL/DB2 syntax to 78% in the case of C#. Absolute numbers of parser hooks greatly differ between parser technology. It is interesting to see, to which extent language coverage relates to coverage of GASTM elements. Table IX gives the percentage of how many GASTM syntax elements are used in transformations. For instance, from a total of 182 concrete GASTM syntax elements, including additions described before, the C++ front-end creates GASTM using 112 concrete GASTM elements. In general, the study confirms the assumption that 100% language coverage is not necessary for the use-cases described above.

D. RQ-4: Tool construction

As shown above, tools reuse platform components. Moreover, it is interesting to see how much additional tool-specific code is required and what code is needed for. Table X lists SLOC of tool-specific code. Source code is mapped to different reverse engineering activities. Remaining code, mostly used for configuration and wiring of *eknows* components, is stated as *Other*. Low numbers of tool-specific code for extraction are not surprising only *Finance-C* provides additional extraction components to reconstruct tables definition from DDL. Additional effort to provide generation components for result presentation is not surprising either. Generator components are kept generic and require major extension if integrated in final reports/visualizations. The only exception here is *RbG*. Reports generated by the tool are basis for report generation in other tools, therefore report generation is completely provided by *eknows* and only limited glue-code is required by *RbG*. Interestingly tools require additional analysis not provided by the platform. Reasons for this are: (1) Tools tweak platform analysis by post-processing results and (2) analysis targeting company-specific code patterns with no reuse potential.

TABLE X
OVERVIEW OF SLOC OF TOOLS AND EKNOWS PER PHASE

Phase	FinC	RbG	ReDoc	Plsql Doc	Varan	Trust score	eknows
Analysis	5238	4425	3589	3887	5002	1353	93453
Extract	3615	0	0	0	0	0	65263
Gen.	5292	0	7202	579	4563	256	44301
Other	6976	318	7649	734	1685	350	144524
Total	21121	4743	18440	5200	11250	1959	347541

VII. CONCLUSION

In this paper we present the polyglot software platform *eknows*. *eknows* facilitates the rapid development of documentation generators and reverse engineering tools target different programming languages. We conducted an industrial case study covering six tools based on the platform to evaluate its effectiveness. Results show that cross-language reuse of platform components can be achieved for all reverse engineering activities. On average the studied tools reuse 32% of analysis components, 22% of extraction components, and 33% of view/generation components. Moreover, we show that the chosen intermediate representation (ASTM) is a sufficiently rich basis for tools comparable to those of our study. Effort of interfacing generic representations with additional programming languages is manageable, as reuse of open-source parsing is feasible and 100% language coverage is not needed.

REFERENCES

[1] E. Chikofsky and J. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[2] H. Sneed and K. Erdos, "Extracting business rules from source code," in *WPC '96. 4th Workshop on Program Comprehension*, 1996, pp. 240–247.

[3] C. Lange, H. Sneed, and A. Winter, "Comparing graph-based program comprehension tools to relational database-based tools," in *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, 2001, pp. 209–218.

[4] J. Ebert, B. Kullbach, V. Riediger, and A. Winter, "GUPRO - generic understanding of programs an overview," *Electronic Notes in Theoretical Computer Science*, vol. 72, no. 2, pp. 47–56, nov 2002.

[5] O. Nierstrasz, S. Ducasse, and T. Gundefinedrba, "The story of moose: An agile reengineering environment." Association for Computing Machinery, 2005.

[6] H. M. Kienle and H. A. Müller, "Rigi-an environment for software reverse engineering, exploration, visualization, and redocumentation," vol. 75, no. 4, 2010.

[7] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "Modisco: A generic and extensible framework for model driven reverse engineering." New York, NY, USA: Association for Computing Machinery, 2010.

[8] J. Van Geet, P. Ebraert, and S. Demeyer, "Redocumentation of a legacy banking system: An experience report," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ser. IWPSE-EVOL '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 33–41.

[9] H. Mössenböck, "A generator for production quality compilers," in *Compiler Compilers*, D. Hammer, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 42–55.

[10] T. Parr, *The Definitive ANTLR 4 Reference*. The Pragmatic Programmer, 2013.

[11] S. Ducasse and S. Tichelaar, "Dimensions of reengineering environment infrastructures," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 15, no. 5, pp. 345–373, 2003. [Online]. Available: https://doi.org/10.1002/smr.279

[12] S. R. T. EDS, IBM, "Architecture-driven modernization: Abstract syntax tree metamodel (astm), v1.0," *OMG Document*, 2011.

[13] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, G. C. Murphy, L. Moreno, D. Shepherd, and E. Wong, "On-demand developer documentation," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 479–483.

[14] M. L. Collard and J. I. Maletic, "srcml 1.0: Explore, analyze, and manipulate source code," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 649–649.

[15] "Chapter 5 - the tools perspective on software reverse engineering: Requirements, construction, and evaluation," ser. Advances in Computers. Elsevier, 2010, vol. 79, pp. 189–290.

[16] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*, 1st ed. Wiley Publishing, 2012.

[17] M. Moser, J. Pichler, G. Fleck, and M. Witlatschil, "Rbg: A documentation generator for scientific and engineering software," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 464–468.

[18] W. Kirchmayr, M. Moser, L. Nocke, J. Pichler, and R. Tober, "Integration of static and dynamic code analysis for understanding legacy source code," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 543–552.

[19] B. Dorninger, M. Moser, and J. Pichler, "Multi-language re-documentation to support a cobol to java migration project," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 536–540.

[20] M. Habringer, M. Moser, and J. Pichler, "Reverse engineering pl/sql legacy code: An experience report," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 553–556.

[21] V. Geist, M. Moser, J. Pichler, R. Santos, and V. Wieser, "Leveraging machine learning for software redocumentation—a comprehensive comparison of methods in practice," *Software: Practice and Experience*, vol. 51, no. 4, pp. 798–823, 2021.