2016 IEEE International Conference on Software Maintenance and Evolution

Integration of Static and Dynamic Code Analysis for Understanding Legacy Source Code

Wilhelm Kirchmayr*, Michael Moser†, Ludwig Nocke*, Josef Pichler† and Rudolf Tober*
*Voestalpine Stahl GmbH, Linz, Austria
Email: (wilhelm.kirchmayr, ludwig.nocke, rudolf.tober)@voestalpine.com

†Software Competence Center Hagenberg GmbH, Hagenberg, Austria
Email: (michael.moser, josef.pichler)@scch.at

***Abstract*—In software development we are faced with the problem to comprehend and take over source code from other developers. The key challenge is to understand the underlying specification implemented by the software system. Regaining this understanding is more difficult when the source code is the only reliable source of information, documentation is outdated or only present in fragments, and original developers are not available anymore. Unfortunately, we encounter such situations frequently for scientific and engineering software systems, developed in industry. For instance, process models in the steelmaking domain are developed and maintained over decades by single engineers. If such an engineer leaves the company, he/she literally leaves behind a legacy system for another person (or team). We propose tool support combining static and dynamic program analysis to tackle this challenge. Using static program analysis we extract the input/output behavior from program source code and present the extracted information besides the analyzed source code, providing seamless navigation between both views. Dynamic program analysis allows developers to examine input/output behavior for single program executions and thereby gain insight into standard behavior and exceptional cases. In this paper we present requirements on tool support integrating static and dynamic code analysis, briefly describe the implementation of the tool and report on its application to a C++ program source in the industry. Furthermore, we discuss challenges in the present implementation as well as the potential and limitations of using the tool in general.**

I. Introduction

The engineering and production in the steelmaking domain is controlled by process control software. Process control software embeds so called process models which are continuously adapted, improved and enhanced. Process models are implemented in Fortran, C, and C++ and maintained over decades since the seventies and eighties of the last century. Process models are typically developed and maintained by a small team, often by a single engineer only. If this person retires, we literally encounter a legacy code problem, where a different engineer has to take over a model implementation he/she has (in worst case) never worked with before. Frequently, the source code is the only reliable documentation of the process model due to outdated manual documentation. If only source code is available, it is hard to comprehend the implemented specification, e.g. the mathematical core model.

One approach to tackle this challenge is to try to extract the implemented specification from the program sources. Today static and dynamic analysis techniques from reverse engineering [1], business rule extraction [2], domain knowledge extraction [3], etc. are available that help to extract specific aspects from a software system. Each type of technique has its advantages and limitations. Static techniques analyze an entire program and generate abstract representations (i.e. models) that describe the behavior of the entire program code. In contrast, dynamic analysis techniques examine single program executions and generate models that describe the behavior of single or, after merging, multiple program runs. Hence, dynamic analysis faces problems with guaranteeing that generated models fully capture the behavior of a software system [4].

Developed methods and tools are very specific with respect to the information extracted from source code and the selection of method and tools largely depends on the characteristics of the examined software system. For instance, the *RbG* [5] tool was specifically developed for scientific and engineering software such as the before mentioned process models.

For understanding process models, typical questions are: *What are the specific results of a process model?*, *Which input values and parameters affect which model result?*, and *What is the specific formula for a model result?* The *RbG* tool facilitates answering these questions by extracting formulae and decision tables from the source code. *RbG* was developed and successfully used in the electrical engineering domain [5][6]. However, the adoption of the tool in other domains was not successful [7]. Initial experiments showed that generating up-to-date documentation for a process model from the steelmaking domain using the *RbG* tool improves comprehension of legacy code. Nevertheless, the experiments also revealed needed improvements with respect to the generated documentation.

The first improvement identified was to specifically show the stepwise computation of every single model result. The second improvement was to include values from model executions into the documentation of the stepwise computation of model results. For this, we integrated existing static analysis techniques of *RbG* with dynamic analysis implemented by an interpreter approach. The dynamic analysis not only improves program comprehension by providing concrete values for extracted computations but also reduces complexity of models

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.70

generated by static analysis. Based on these ideas, we started the development of tool support called VARAN that generates documentation based on the idea of *RbG* but with the before mentioned improvements.

The contributions of this paper are:

1) We characterize (legacy) software in process control domain and present requirements for documentation generators aimed at process model implementations.
2) We introduce an approach and tooling that integrates static and dynamic code analysis to generate documentation facilitating the comprehension of process models in general and for specific model executions.
3) We generate documentation for a real-world legacy process model from the steelmaking domain.

The structure of this paper is as follows: in Section II we give background information on the domain and formulate requirements for our tool support. In Section III we give an overview of our approach and, in particular, describe the content of generated documentation based on integration of results from static and dynamic analysis. This is followed by detailed description of the applied static analysis (Section IV) and dynamic analysis (Section V). Section VI contains a result report based on the application of our approach in the steelmaking domain complemented with discussion on the approach in Section VII.

II. Background and Requirements

A. Background

In this section we characterize the software to be analyzed by means of tool support presented in this paper. Even though the presented tool support was designed, implemented, and used for one specific process model only, we emphasize aspects that are typical for the process industry domain.

1) The Process Industry Domain: The engineering and production in the steelmaking domain is controlled by process control software. Foundational part of this software are computational models, also referred to as process models [8]. Process models are a core asset of a company and are engineered and implemented in software according to the state-of-the-art and afterwards, continuously improved, adapted, and enhanced. Maintenance activities are triggered mainly from the feedback of the process model in production. For instance, when the quality of the resulting product is not sufficient, an engineer has to advance the process model in order to generate better model results that control the production process. Cost optimization is also a frequent trigger for model adaptions, i.e. to achieve the same quality with less energy or resources. Of course, occasionally also bug fixes are necessary. Bugs to be fixed are typically introduced during model enhancements, since most preexisting bugs were already detected and fixed due to the long operation of the software system.

2) Process Model Implementation: Today we encounter process models that are implemented in the programming languages Fortran, C and C++ and are maintained over decades since the seventies and eighties of the last century. Some programs were migrated from older technology (e.g. Fortran) to newer ones (e.g. Java), others were never migrated and are still available in the original technology. Procedural models are either used online during production or offline for simulation purposes. In any case, process models need data exchange with databases containing process data (online) or simulation data (offline). Hence, the implementation performs some kind of database access. Ideally access to databases is abstracted from specific database technology, in practice, however, we encounter tight coupling between the model implementation and database access.

3) Software Development: Scientific and engineering software is typically developed and maintained by a small team, often by a single engineer only. The engineer is responsible for a couple of mathematical or process models which he/she maintains over a long time period, sometimes even over decades. If this person retires, we encounter a legacy code problem, where another engineer, who (in the worst case) has never worked with the implementation of the process model before, has to take over. Due to the high mathematical character of such models comprehension is difficult even for small programs ($<$ 10 kLOC). Moreover, the source code is accompanied by documentation that was accurate for the initial implementation but by now is long outdated and cannot be considered as a reliable source of information. Therefore, the source code is often the only documentation of a process model.

B. The Desulphurization Process Model

The process model for which documentation generation was desired follows the subsequent characterization. The characterization is schematically depicted in Figure 1. The process model for *desulphurization* (called *DeMo*) is implemented in the C++ programming language and comprises 5400 source lines of code. The process model is mainly implemented in a procedural way containing classes and methods but without exploiting core object-oriented techniques such as polymorphic variables and dynamic method dispatch. The core of this process model was implemented by a third party company. After the initial development, a process engineer maintained the C++ source code over decades until he retired in 2015. After retirement and, unfortunately, without intensive know-how transfer, two other process engineers had to take over this implementation.

Today, the process model computes about twenty model outputs based on twenty model inputs. The structure of the process model changed over time because the initial core was hardly ever changed. Instead, necessary adaptations were implemented by computations *before* and *after* the core model computations. For instance, by fixing certain input values passed to the core model's entry point function F2, it was ensured that a desired branch within the core model gets executed instead of changing corresponding conditional statements within the core implementation. Moreover, values calculated by the core model are sometimes updated by some fudge factor before returned by process model's main (entry

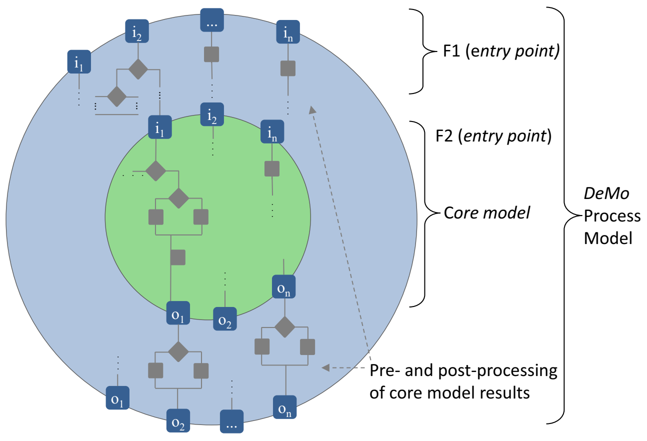


Fig. 1. Multi-shell structure of process model for desulphurization.

point) function F1. This metamorphosis of the process model from the initial core model into a multi-shell process model, with the core model in the middle and extensions around the core, complicates maintenance tasks considerably.

C. Requirements and Goals

The need for documentation generation was identified and requested by two process engineers after an initial attempt to regain the understanding of the process model failed. Both persons are experts in the steelmaking domain and have an overall understanding of the process model for desulphurization, but lack detailed understanding of required steps and their dependencies. Following requirements for documentation generation were identified:

1) Simplify maintenance and optimization: Documentation generation must simplify maintenance and optimization of process models. Maintenance of process models is triggered by improvements of metallurgical foundations, by feedback on the process model in production, or by cost optimizations, i.e. to achieve the same quality with less energy or resources in general. Occasionally, also bug fixes trigger maintenance work. All maintenance activities require a deep understanding, not only of the software structure and specific functions, but as well of the general principle of a process model, its individual steps, and the transitive dependencies between model inputs and model outputs.

2) Offline documentation: It must be possible to generate documentation in form of a set of files which can be archived in a repository. The specific document format was not specified from the very beginning. Interactively browsing documentation with easy navigation between documentation and source code was requested. As the generated documentation is typically consumed by a domain expert, who has to change source code in an IDE (e.g. Microsoft Visual Studio), integration within an IDE (e.g. as an additional view component) seemed obvious at first place.

However, besides the additional effort for plug-in development and dependency on a specific IDE, main reason to not pursue IDE integration was the requirement to easily bundle documentation together with a specific version of a process model.

3) Flexible clipping of model source code: Process models in general and the desulphurization model in particular are part of a model server that provides integration of computational models with the process data infrastructure. Depending on the architecture of the model server and the data exchange, computational models are not always encapsulated as isolated modules but are part of the overall model server. It must be possible to specify the entry point of a computational model by its corresponding C++ method together with input and output data structures. In contrast to the entry point, it must be also possible to specify exit points that denote the boundary of a computational model. For instance, model implementation may access databases of sub systems at any point in execution. These aspects are of no relevance to documentation generation and should be excluded.

4) Hierarchical model structure: Process models are typically hierarchically structured into sub models, which may be also shared between different process models. Generated documentation must reflect this hierarchical structure in a way that either the overall model or just an individual sub model can be explored.

III. Overall Approach

To tackle the challenges and requirements mentioned in the previous section, we pursue an approach for documentation generation that integrates standard methods from static and dynamic program analysis. Our approach together with the corresponding tool implementation was mainly driven by the industrial use case described in Section II. Nevertheless, general applicability for scientific and engineering software was considered from the very beginning. This section provides an overview of our approach including selected analysis techniques as well as the selected documentation output. Details on the applied analysis are discussed in subsequent sections.

The following source code listing is used as running example for the detailed discussion of static and dynamic analysis in subsequent sections as well as for the presentation of the desired documentation output in this section. For the sake of brevity we use a synthetic example. However, it contains the most interesting and most noteworthy aspects of the described industrial use case.

The fictive process model depicted in the source code listing with entry point function `SnowAnalysis` computes a single model output (parameter field `out->snow_density`) from a set of input values given by parameter `in`. The function verifies input values using the auxiliary function `CheckInput` and triggers the actual computation of snow density in method `SnowDensity::Calculate`. Afterwards, the calculated snow density is optionally adapted by a factor retrieved from a database table (in function `FudgeFactor`). The `Main` function is not considered as a part of the process model and represents the overall execution infrastructure into which a process models is typically embedded.

```
//Start Main.cpp
int Main() {
  input in = input{};
  output out = output{};
  snowAnalysis(in, &out);
  return 0;
}

void CheckInput(input *i) {
  double minHeight = 0.0001;
  if (i->snow_height < minHeight) {
    i->snow_height = minHeight;
  }
  return 0;
}

double FudgeFactor(input in) {
  double factor;
  SqlExecuteSR("SELECT ", "...","...", "f ",
       &factor, "WHERE", "H", in.snow_height);
  return factor;
}

void SnowAnalysis(input in, output *out) {
  CheckInput(&in);
  SnowDensity *density = new SnowDensity();
  density->Calculate(&in, out);

  if (in.useFudgeFactor) {
    out->snow_density = out->snow_density *
         FudgeFactor(in);
  }
}
//End Main.cpp

//Start SnowDensity.cpp
SnowDensity::SnowDensity() {
  rho_w_r = 1000.0;
  rho_S_min = 100.0;
  rho_S_max = 400.0;
}

void SnowDensity::Calculate(input *in, output *out){
  double snowdensity = 1.0 - in->snow_height *
       in->gamma_rho_S / rho_w_r;
  snowdensity = rho_S_min / snowdensity;

   if (snowdensity > rho_S_max) {
    snowdensity = rho_S_max;
  }
  out->snow_density = snowdensity;
} //End SnowDensity.cpp

//Start Commons.h
typedef struct {
   double snow_density;
} output;

typedef struct {
   double snow_height;
   double gamma_rho_S;
   bool useFudgeFactor;
} input;
//End Commons.h
```

The generated documentation represents the analyzed source code on various levels of abstraction (e.g. function level and statement level) and different slices (e.g. a single model executions). Generated documentation consists of following representations of process models providing easy navigation between different views.

1) It provides a *Big Picture* that shows the essential program structures and boundaries of the model implementation and serves as an entry point for the user of the generated documentation.
2) For every output of the process model, the documentation contains a list of formulae that specify the stepwise computation of a model output from model inputs.
3) A list of formulae specifying the stepwise computation for an output computed from a specific model execution and extended by computed result values.

The source code of the analyzed model implementation is part of the generated documentation as well. To see analysis results side by side with original program statements was an explicit request from model experts. The generated documentation links documentation elements, e.g. elements of the *Big Picture* and single formulae from the stepwise computation, to the corresponding source code lines. Since the source code is packaged together with the actual documentation, it is always the source code version the documentation was generated from, which is displayed to readers.

In detail, the different representations of generated documentation can be described as follows.

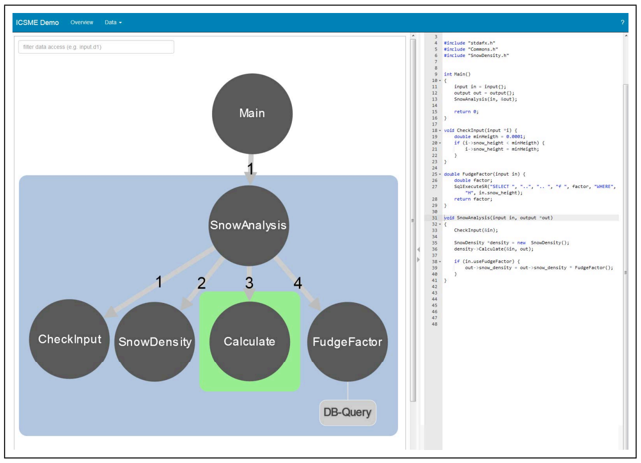
A. Big Picture

To facilitate quick overview, the software system is decomposed into major program structures. The generated *Big Picture* provides a hierarchical graphical representation, that satisfies a key requirement of stakeholders, namely to see major program structures and dependencies on a ”single sheet of paper”. Figure 2 shows a screen dump of the generated documentation for the source code of our running example with the *Big Picture* on the left side. The apparent structure of the model implementation is given by C++ methods[1] and functions together with ordered call relations between functions. The order of method/function calls are labeled on the call relation. For process models that are divided into sub models, as in our use case, the C++ methods and functions are grouped into sub models based on specified entry points of a sub model (e.g. method `Calculate`). Furthermore, the generated diagram also contains database access from model implementation appearing at any C++ method or function (e.g. in C++ function `FudgeFactor`).

B. Stepwise Computation of a Model Output

Core output of the generated documentation are tables covering the stepwise computation of a model output. For each model output, the documentation contains one table with table rows for all assignment statements which may influence the computation of a model output. Figure 3 shows a schematic representation of such a Table for the model output $S_{density}$ (field `out.snow_density`). Each table row gives the variable name, the assignment to the variable in mathematical notation, the path condition, and whether model inputs are

[1]The class name of C++ methods is omitted in the generated output for sake of simplicity.



```
#include "stdafx.h"
#include "Commons.h"
#include "SnowDensity.h"


int Main()
{
    input in = input{};
    output out = output{};
    SnowAnalysis(in, &out);

    return 0;
}

void CheckInput(input *i) {
    double minHeigth = 0.0001;
    if (i->snow_height < minHeigth) {
        i->snow_height = minHeigth;
    }
}

double FudgeFactor(input in) {
    double factor;
    SqlExecuteSR("SELECT ", "..", ".. ", "f ", factor, "WHERE",
        "H", in.snow_height);
    return factor;
}

void SnowAnalysis(input in, output *out)
{
    CheckInput(&in);

    SnowDensity *density = new  SnowDensity();
    density->Calculate(&in, out);

    if (in.useFudgeFactor) {
        out->snow_density = out->snow_density * FudgeFactor();
    }
}
```

Fig. 2. Screen dump of generated documentation for running example including ordered call graph (left) and source code view (right). The model boundaries are specified by C++ function `SnowAnalysis` and C++ method `SnowDensity::Calculate`.

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

Fig. 3. Static result table for stepwise computation of model output $S_{density}$ based on model inputs S_{height} and γ_{roh}.

accessed directly within the assignment. Definitions of any program variable used within an assignment statement are repeatedly resolved until definitions are derived from model inputs or literal values only. Starting with definitions of a model output all reaching definitions are added depth-first to the table. Indentation of variables indicates the distance of the current table-row to root definitions of model outputs.

Extraction and representation of mathematical formulae from program sources is a key requirement of documentation of scientific and engineering software [5]. Therefore, program statements are represented in mathematical notation. Besides the formulae used to compute the definition of a program variable, path conditions significantly contribute to the comprehension of program code. Identifiers of program variables which can be directly mapped to model inputs or outputs are replaced with corresponding symbol names.

C. Stepwise Computation of a Model Execution

To further improve understandability of program sources, specific values from model executions are included within the documentation. Therefore, static result tables are extended by specific values from model executions.

As Figure 4 shows, an additional table column *Value* is added containing the calculated values for each formulae. This not only facilitates comprehension of a formula of a single step but as well improves understandability of computation of an entire model output. Moreover, computation of examples by model execution can yield more compact and expressive documentation, since program paths which are not executed in a model execution can be hidden from readers. Generated documentation actually allows readers to collapse result tables to only show executed program paths. Documentation of an model execution lists all model inputs which influence the computation of a single model output.

D. Tool Support

The described analysis and documentation generators are implemented in the tool VARAN. The tool was specifically developed for the problem setting described in previous section. VARAN fulfills two main requirements. First, maintainers of the software system can use the tool to regenerate documentation after source code of the targeted software system is changed. Second, VARAN can be used to add new cases of model executions to existing documentation.

Figure 5 shows input data, main architectural aspects of the tool, and generated output artifacts. The tool is implemented in Java and is executed from the command line. To run a new analysis and generate documentation VARAN is parametrized with (1) the source location of the targeted software system, (2) a file path from which files containing input values are picked up, (3) and an output location where to generated documentation artifacts are written.

The tool reuses parsing infrastructure from the *Metamorphosis* toolkit [9]. *Metamorphosis* provides a set of parsing front ends (e.g. C++, Fortran, Cobol, PL/SQL) and transforms parse trees into generic abstract syntax trees (GAST) conforming to the abstract syntax tree meta-model (ASTM)

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. Dynamic result table for stepwise computation of model output $S_{density}$ with model input $S_{height} = 1.6$, $\gamma_{roh} = 200$, and $fugde_{fact} = true$.

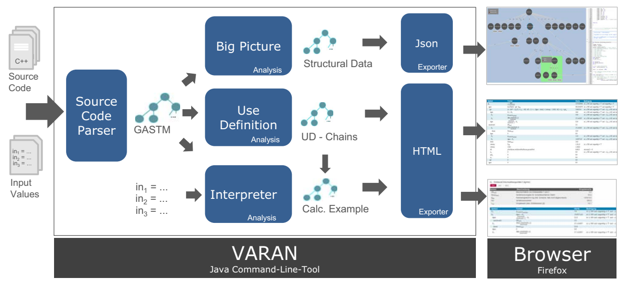


Fig. 5. VARAN - Architectural Overview

specification published by the Object Management Group (OMG) [10]. Main reason for this major design decision was the possibility to reuse preexisting analysis components, i.e. call-graph analysis and control-flow analysis. VARAN implements three major analysis components. First, structural analysis to compute the overview picture. Second, data-flow analysis to create interprocedural use-definitions, and third an interprocedural program interpreter which performs example calculations. All three components take GAST data structures as input. The computed result data structures are exported as JSON and HTML.

IV. Static Code Analysis

A. Analysis for Big Picture

The analysis for *Big Picture* extracts structural information from program sources in order to create an overview picture of the examined program structures as shown in Figure 2. Analysis performs the following steps.

1) Identification of program structures, i.e. functions, methods, and constructor definitions, from abstract syntax trees of parsed source code units.
2) Analysis of call relations between program structures. For this, a call-graph [11] is constructed from call relations between functions and constructors. The call-graph shows order of function calls, repetition of function calls (e.g. within loops), and path conditions under which function calls are executed.
3) Clustering of program structures into sub models builds upon the already constructed call-graph and groups all functions reachable through the provided entry point into a single sub model. If besides the mandatory main entry point additional entry points are specified, functions definitions reachable through these are again clustered to sub models.
4) Read and write analysis of entry points parameters. For each function which is identified as entry point, the analysis detects read and write operations to parameters (i.e. C++ structures) and marks them accordingly.
5) Identification of database queries within program structures. Input data retrieved via database queries significantly increase complexity and hamper understandability of data-flow within program sources. Therefore, *Big Picture Analysis* highlights functions which perform database access. This step of the analysis is very specific for the examined software system. Database queries are executed via function calls to `SQLExecute` like shown in the provided code listing. The function `FudgeFactor` retrieves an additional input value from database. The query statement itself is passed as string parameter. Since no further data-flow analysis is required at this point a simple pattern matching on respective function calls, i.e. calls to `SQLExecute`, is sufficient to identify database queries.

B. Static Analysis for Stepwise Computation

In general the analysis performs an interprocedural data flow analysis [12] that computes the flow of data across function boundaries, considering parameter usage and global variables [13]. To show definitions, i.e. program statements, reaching model outputs the analysis constructs use-definition chains [14] from an interprocedural control-flow graph. The applied analysis fulfills the following objectives:

- Show how model outputs are calculated
- Resolve all definitions reaching the model outputs
- List path conditions for each reaching definition
- Show influence of model inputs on model outputs
- Display defining source statements in mathematical notation

The analysis to compute reaching definitions for a set of model outputs starts with computing call-relations between function definitions starting from an initial entry point. The thereby created call-graph is used to build an *interprocedural* control-flow-graph (CFG). The graph is aware of interprocedural calls and constructs new control-flow sub-graphs for called function definitions. However, multiple calls to the same function definition yield the same control-flow graph. The analysis of use-definition chains traverses the control-flow graph and collects definitions and uses of variables. Basic blocks of the CFG which call functions are split into an *entry* block preserving the calling context and an *exit* block which merges definitions from interprocedural calls. Definitions are thereby propagated in forward direction throughout the interprocedural control-flow graph. Next to the actual source expression, variable name and symbolic information derived from specification of model inputs and model outputs, the analysis collects path conditions and attaches them the computed definitions. Next, the complete use-definition information collected by traversing the interprocedural control-flow graph is used to compute use-definition chains for all model outputs defined by domain experts.

An interesting aspect of the analysis is the role of entry point specifications. In the running example described above the function `SnowAnalysis` acts as main entry point for the analysis. All functions which are only calling this entry point, e.g. `Main`, are not considered in static dependency analysis. Moreover, if additional entry points are specified, the analysis creates the following results: (1) a result set containing all definitions starting from main entry point,(2) result sets containing reaching definitions starting from the additional entry points, and (3) a result set starting at the main entry point, but cutting off all definitions collected from functions reachable through additional entry points. Instead only a single function call to the entry point function is included. This indicates, that the model output depends on the submodel. Figure 3 shows result from static analysis which contains all reaching definitions starting from entry point function `SnowAnalysis`. If, as indicated in the *Big Picture*, `SnowDensity::Calculation` is specified as an additional entry point, two more result tables are created. One that contains results from dependency analysis for method `SnowDensity::Calculation` and a second which cuts off reaching definitions from `SnowDensity::Calculation`. Therefore, rows 2–7, and 10 are not include in the result table. Instead, an additional row with the field `out.snow_density` and formula `SnowDensity::Calculation()` is added. By this, analysis of the software system is subdivide and overall complexity of the generated documentation is reduced.

A fundamental part of the analysis not described yet, is the extraction of formulae and path conditions from program sources. Simple mathematical formulae are obtained by transformation of an assignment statement into a formula. The left hand side of an assignment becomes the subject of the formula and the expression on the right hand side of an assignment statement becomes the specification how the subject can be calculated based on a given set of variables. Path conditions are again collected by traversing the interprocedural control-flow-graph and are added to results.

V. Dynamic Analysis

Analysis to compute model executions performs a flow-sensitive program interpretation and merges the values calculated during program interpretation with use-definition chains produced in static analysis. Merging calculated values from model executions and use-definition chains yields dynamic results which are more compact and easier to understand. In detail, the analysis works as follows.

Model executions are computed from a set of source code files, an entry point specification, model inputs, model outputs, and a set of concrete input values used in program interpretation. Similar, to the presented static analysis for reaching definitions, the analysis processes an interprocedural control-flow graph (CFG) constructed from the call-relations between function definitions. Starting from the specified entry point the analysis processes each basic block in the CFG, substitutes variables with concrete values and interprets program statements using its AST representation. Besides constant values defined in program code, concrete values are defined by a set of input values. Typically, input values are defined for each model input element.

The running example provided in Section III shows how input values are used by the analysis. For each field in the C++ structure `input` concrete values are defined. E.g. `input.snow_height = 1.6`, `input.gamma_rho_S = 200`, and `input.use_fudge_factor = true`. Moreover, input values can be defined to be used as return values for entire functions. This is useful when depending from input values passed to an entry point, other input values are acquired, e.g. in the case of database queries. From the code listing of the running example one can see that depending on the actual `input.snow_height` a fudge factor is retrieved from a database query. This code snippet is substituted be defining a return value for the entire function `FudgeFactor` (0.5). Control structures, i.e. conditional statements and loop statements are recognized and interpreted accordingly. This can be seen in the result table generated from the running example (Figure 4). Since the provided `input.snow_height` (1.6) does not fall under a constant `minHeight` subsequent assignment statements are not executed and grayed out in result representation.

Again, basic blocks which contain interprocedural calls are split into an *entry* block and an *exit* block. The entry block creates a valid calling context and executes interpretation of the called function. Finally, the exit block collects the results from the interpretation of the called function and merges values back into the calling context. As stated in the beginning of this section, computation of model executions start at the main entry point. Opposed to static analysis described in the previous section additional entry points have no influence. Concrete values are collected for the entire program. After interpretation the results of model execution, i.e. a list of calculated values per program statement, are merged with result tables created by the previous analysis step.

VI. Results and Lessons Learned

This section reports on results taken from generating documentation for a process model as described in Section II. Domain experts provided definitions of model outputs, definitions of model inputs and mapped model inputs to three sets of standard input values. In this way, data was provided for two entry points within the examined software system. First, a main entry point *F1* which is used to cut out the process model *DeMo* from its embedding software infrastructure, and a second entry point *F2* which identifies the core computation of the desulfurization of iron. For entry point *F1*, 20 model outputs and 21 model inputs were defined; for entry point *F2*, 21 model outputs and 22 model inputs were defined. Basically the defined model inputs/outputs resemble members of C++ structures passed as parameters to entry point functions.

Tables I, II, and III give the number of dependent model inputs, and the number of rows in a result table, i.e. number of reaching definitions, for each model output. The tables list data collected from static analysis (columns 1 and 2) and from three example calculations using dynamic analysis (columns 3 - 8).

The Tables I and II show data from analysis which is restricted to sub models only. In Table I we list data from analysis which started from entry point *F1* and was bounded by *F2*, i.e. data-flow analysis for reaching definitions contained within the bounding sub model are substituted by a single

TABLE I
RESULTS FOR ENTRY POINT F1 ONLY.

	Static		Dynamic 1		Dynamic 2		Dynamic 3	
	input	result	input	result	input	result	input	result
o1	9	65	3	11	3	11	0	15
o2	9	63	3	11	3	11	1	11
o3	5	50	0	6	0	6	0	6
o4	5	50	0	6	0	6	0	6
o5	4	43	0	5	0	5	0	5
o6	4	41	0	6	0	6	0	6
o7	4	41	0	6	0	6	0	6
o8	5	38	0	5	0	5	0	5
o9	3	38	0	19	1	19	0	18
o10	3	25	0	5	0	5	0	5
o11	1	24	0	13	0	13	0	11
o12	1	24	0	13	0	13	0	12
o13	2	21	0	5	0	5	1	5
o14	1	15	0	5	0	5	0	5
o15	1	11	0	4	0	4	0	4
o16	0	6	0	3	1	3	3	3
o17	0	1	0	1	0	1	1	1
o18	0	1	0	1	0	1	0	1
o19	0	1	0	1	0	1	0	1
o20	0	0	0	0	0	0	0	0

TABLE II
RESULTS FOR ENTRY POINT F2.

	Static		Dynamic 1		Dynamic 2		Dynamic 3	
	input	result	input	result	input	result	input	result
o1	12	190	8	41	8	41	1	3
o2	11	174	7	35	7	35	1	23
o3	11	170	7	35	7	35	1	3
o4	11	166	7	33	7	33	6	21
o5	11	164	7	33	7	33	4	3
o6	11	164	7	31	7	31	6	20
o7	11	163	7	31	7	31	1	3
o8	11	153	5	19	5	19	1	3
o9	11	151	5	18	5	18	4	12
o10	2	14	2	4	2	4	1	3
o11	1	13	0	3	0	3	0	3
o12	1	12	0	0	0	0	0	3
o13	1	11	0	0	0	0	6	0
o14	1	11	0	0	0	0	1	0
o15	1	9	0	2	0	2	0	0
o16	1	2	1	2	1	2	0	2
o17	0	2	0	2	0	2	1	5
o18	0	2	0	2	0	2	0	2
o19	0	0	0	0	0	0	0	2
o20	0	0	0	0	0	0	1	0
o21	0	0	0	0	0	0	0	0

function call to *F2*. Table II shows data which is restricted to functions reachable by entry point *F2*.

Reasons for the division of the analysis of the software system at these points are twofold. First, function *F2* was chosen to match existing documentation with the documentation generated by our approach. The function contains the main algorithm to calculate desulfurization and was developed by a third party company, which provided as well the documentation of the mathematical model. Second, due to the complexity of *F2* maintainers of the software tended to build shells around the core, rather than changing *F2* itself. To highlight this usage in the generated documentation *F1* was chosen as entry point which is bounded by *F2*, i.e. does not contain any reaching definitions from *F2*. This can be well observed by the much lower number of reaching definitions in Table I compared to Table II.

TABLE III
RESULTS FOR ENTIRE PROCESS MODEL.

	Static		Dynamic 1		Dynamic 2		Dynamic 3	
	input	result	input	result	input	result	input	result
o1	4	268	9	62	9	62	0	7
o2	9	267	11	58	11	58	10	45
o3	9	267	10	58	10	58	3	17
o4	5	259	8	51	8	51	7	39
o5	5	259	8	51	8	51	0	8
o6	9	256	9	56	9	56	0	7
o7	4	247	7	36	7	36	6	29
o8	4	247	7	37	7	37	0	8
o9	3	236	8	65	8	65	7	52
o10	1	228	8	59	8	59	0	45
o11	1	228	8	59	8	59	0	14
o12	2	227	8	55	8	55	0	42
o13	3	218	0	6	0	6	0	6
o14	1	34	0	8	0	8	0	7
o15	1	12	0	3	0	3	0	5
o16	0	4	0	3	0	3	0	3
o17	0	1	0	1	0	1	0	1
o18	0	1	0	1	0	1	0	1
o19	0	1	0	1	0	1	0	1
o20	0	0	0	0	0	0	0	0

Table III on the contrary lists data collected from analysis which is not restricted to sub model definitions. As in Table I model inputs and outputs are defined for entry point *F1*, but are not restricted by any sub model definition. Therefore, reaching definitions and depending model inputs are collected for the entire process model. As one can see from all tables the numbers for result rows per output computed from static analysis range from 0 to 268. While understandability of the generated result documentation is satisfying for outputs with a low number of result rows (i.e. < 40), larger result sets are hard to comprehend and usability of generated documentation can be questioned.

However, as one can see from comparing the number of rows computed by static and dynamic analysis, a significant drop in the number can be observed. For all three example calculations the number of result rows is reduced. This corresponds to general observations in dynamic analysis [15]. To us, not the reduction in general but the amount to which result rows could be reduced was of interest. The three example calculations were executed using input values, which are derived from standard desulfurization calculations. For the top 5 model outputs in Table II ranging from 164 to 190 the reduction of result rows was between 78% to 98%. Dynamically filtering rows which were not executed from result tables significantly improved usability as well as acceptance of the generated documentation. Furthermore, evaluation of formulae with real values stimulates the understanding of a single formulae as well as the understanding of data-flow through a program. Expected model output values, taken from real executions, matched our interpreted results.

The *Big Picture* computed by static analysis is presented in Figure 6. Names of functions and model input and output were replaced with symbolic names. In total 38 functions were detected, from those 38 functions 12 functions are not considered to be a part of the main process model. The process model is comprised by 26 functions (blue rectangle, entry

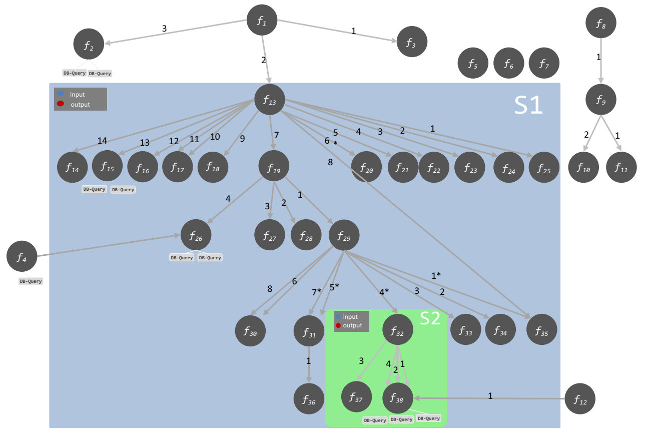


Fig. 6. Big Picture generated by static analysis for examined system: process model with entry point $F1$ (function f_{13}) and a sub model with entry point $F2$ (function f_{35}).

point $F1$, function f_{13}). From those 26 functions, 3 functions are again grouped to a separated sub model (green rectangle, entry point $F2$, function f_{35}). For both sub models boundaries, model input structures and output structures are given. Order of function calls are given by numbers over the edges between functions. 8 function calls appear within a loops. This is indicated by an asterisk (*) following the call order number. Moreover, the analysis detected 10 unique database queries. 7 queries are found within the process model, 3 queries were detected within embedding infrastructure code.

VII. Discussion

The presented tool support was specifically created for the examined industrial use case. Implementation and optimization of the analysis are limited in several ways. For instance, bindings are resolved statically only. For the industrial use case this does not impose any problem. However, in the general case it obviously would. In particular, dynamic analysis must consider dynamic binding issues, if filtering and evaluation of formulae in an interprocedural setting should produce correct values. Furthermore, we do not provide a 100% language coverage for C++ language. Features like templates, lambda expressions, variadic parameters, automatic type inference using `auto` or declaration of attributes are not covered by our analysis. Moreover, we do not support pointer arithmetic.

The limitations of the provided implementation suggest, that if used out-of-the box the presented approach generates valid results for the industrial use case or very similar implementations only. Other features of the analysis, like the detection of database queries are implemented in way, that work for the industrial use case only. Besides, limited language support for C++ and use case specific implementation, thorough checks on analysis execution are missing. Failing or missing detection or abortion of infinite loops in analysis runs are examples to that and would currently clearly hinder the adoption of our approach in the general case. In the case of missing input values, interpretation of program statements cannot be performed and undefined values are added, yielding incomplete results. A combination of concrete and symbolic interpretation, i.e. concolic execution [16] , could hereby provide improvement. The definition, of model inputs and outputs are fixed for the examined use case and are provided through source code. However, this could be easily changed to facilitate adaption of the existing use cases or adoption of new use cases by maintainers of a software system. Equality of results values produced by an interpreted program run and those produced by real program execution are an open issue which has to be treated thoroughly. Even though the interpreter produces sound result values for the tested input values, simply the fact that the interpreter component is implemented in Java and the real program in C++ can yield different results, due to differences in language implementations concerning floating point operations or varying usage of compiler switches. To overcome this limitation instrumentation of a real program run and recording execution values via a debug interface appears to be promising.

Partly on purpose, but still, the applied extraction of formulae from source code is limited as well. The idea of formula extraction can be taken much further. In a previous paper [5] we have shown how to use substitution to replace variables within formulae with their last assignment in order to build closed formulae, merge assignments with different path conditions to formulae with case and decision tables, or apply a pattern-based approach to extract aggregate functions, such as sum or product, from a sequence of program statements.

An obvious limitation of the analysis of the *Big Picture* is that it does not scale for large software systems. A key requirement for this analysis was to grant quick overview and present the results in a compact style. If the number of functions is much larger no comprehensive overview can be given. In this case other top-level abstraction, e.g sub models or components, would have to be identified.

Nevertheless, we think that the presented approach has high potential to improve understandability of software code in general and in particular scientific and engineering software. The integration of dynamic analysis techniques with static analysis techniques yields comprehensible program slices, which can be used to generate documentation facilitating different stakeholders to understand the specification underlying an implementation by example. Initial experiments with open source software reinforce this opinion. For instance, we applied VARAN to the open source software FLake (`http://www.flake.igb-berlin.de/`). FLake is a freshwater lake model capable of predicting the vertical temperature structure and mixing conditions in lakes. Similar to the software system of the presented industrial use case, FLake calculates a set of model outputs from a set of model input.

VIII. Conclusion

In this paper, we have provided insight in our approach, tool, and application of documentation generation for legacy process models. We have integrated static analysis techniques and dynamic analysis techniques to generate documents that show the stepwise computation of model outputs. The integration of dynamic analysis not only facilitates comprehension by providing specific values but also reduces the complexity of the results from static analysis.

The generated documentation is promising and will be used by two domain experts for subsequent maintenance tasks of the process model *DeMo*. We expect that using the generated documentation for specific maintenance tasks will reveal shortcomings of our approach and will trigger new ideas to improve our approach and tooling. We plan to incorporate such feedback in future and eventually also fix some limitations of the current solution in order to apply our approach to other software systems.

Acknowledgment

The research reported in this paper has been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

References

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

[2] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet, "Extracting business rules from cobol: A model-based tool," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, Oct 2013, pp. 483–484.

[3] I. Roanc, "Framework for web application domain knowledge extraction," in *Information Communication Technology Electronics Microelectronics (MIPRO), 2013 36th International Convention on*, May 2013, pp. 705–710.

[4] C. E. Silva and J. C. Campos, "Combining static and dynamic analysis for the reverse engineering of web applications," in *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS '13. New York, NY, USA: ACM, 2013, pp. 107–112. [Online]. Available: http://doi.acm.org/10.1145/2494603.2480324

[5] 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.

[6] J. Pichler, "Extraction of documentation from fortran 90 source code: An industrial experience," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, March 2013, pp. 399–402.

[7] M. Moser and J. Pichler, "Documentation generation from annotated source code of scientific software: position paper," in *Proceedings of the International Workshop on Software Engineering for Science*. ACM, 2016, pp. 12–15.

[8] B. Dorninger, "A process model runtime environment based on OSGi," in *2009 7th IEEE International Conference on Industrial Informatics*. Institute of Electrical & Electronics Engineers (IEEE), jun 2009. [Online]. Available: http://dx.doi.org/10.1109/INDIN.2009.5195922

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

[10] 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/

[11] B. G. Ryder, "Constructing the call graph of a program," *IEEE Trans. Softw. Eng.*, vol. 5, no. 3, pp. 216–226, May 1979. [Online]. Available: http://dx.doi.org/10.1109/TSE.1979.234183

[12] J. M. Barth, "A practical interprocedural data flow analysis algorithm," *Commun. ACM*, vol. 21, no. 9, pp. 724–736, Sep. 1978. [Online]. Available: http://doi.acm.org/10.1145/359588.359596

[13] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," *SIGPLAN Not.*, vol. 21, no. 7, pp. 152–161, Jul. 1986. [Online]. Available: http://doi.acm.org/10.1145/13310.13327

[14] M. J. Harrold and M. L. Soffa, "Efficient computation of interprocedural definition-use chains," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 2, pp. 175–204, Mar. 1994. [Online]. Available: http://doi.acm.org/10.1145/174662.174663

[15] B. Korel and J. Rilling, "Dynamic program slicing methods," *Information and Software Technology*, vol. 40, no. 11-12, pp. 647–659, dec 1998. [Online]. Available: http://dx.doi.org/10.1016/S0950-5849(98)00089-5

[16] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750