

Reverse Engineering Component Models for Quality Predictions

Steffen Becker, Michael Hauck, and Mircea Trifu
 FZI Research Center
 Software Engineering
 Karlsruhe, Germany

Klaus Krogmann
 Karlsruhe Institute of Technology
 Software Design and Quality
 Karlsruhe, Germany

Jan Kofroň
 Charles University in Prague
 Distributed Systems Research Group
 Prague, Czech Republic

Abstract—Legacy applications are still widely spread. If a need to change deployment or update its functionality arises, it becomes difficult to estimate the performance impact of such modifications due to absence of corresponding models. In this paper, we present an extendable integrated environment based on Eclipse developed in the scope of the Q-ImPrESS project for reverse engineering of legacy applications (in C/C++/Java). The Q-ImPrESS project aims at modeling quality attributes (performance, reliability, maintainability) at an architectural level and allows for choosing the most suitable variant for implementation of a desired modification. The main contributions of the project include i) a high integration of all steps of the entire process into a single tool, a beta version of which has been already successfully tested on a case study, ii) integration of multiple research approaches to performance modeling, and iii) an extendable underlying meta-model for different quality dimensions.

I. INTRODUCTION

The goal of the Q-ImPrESS project¹ is to provide an integrated method and appropriate tools to support a quality-aware evolution of service-oriented software architectures, which allows software engineers to compare alternative architectural designs for a given evolution scenario and choose the best alternative already in the design phase, thus avoiding costly trial-and-error prototyping.

Q-ImPrESS is concerned with three quality attributes: performance, reliability, and maintainability, as well as the typical trade-offs between them. Although such trade-offs constitute an unavoidable aspect of every architectural design decision, their handling is implicit and not supported in traditional software development and evolution processes. Q-ImPrESS aims to change these processes by allowing software engineers to predict the impact of their design decisions on the considered quality attributes, and to understand the trade-offs between them.

The main component of the Q-ImPrESS method [4] is the Service Architecture Model (SAM), a central repository containing the architecture of a service-oriented system. It includes elements required for performance and reliability impact predictions such as static structure (components and connectors), component behaviours, and deployment, as well as quality annotations and usage profiles and can serve as a base for maintainability analysis and formal protocol checking.

¹Q-ImPrESS is a research project funded by the European Union under the Information and Communication Technologies priority of the Seventh Research Framework Programme. See <http://www-q-impres.eu>.

The SAM is typically extracted from the source code of an existing software system, using the reverse engineering process described below, and used to simulate several alternative architectures for each considered evolution scenario, while predicting for each alternative the impact of the taken design decisions on and the trade-offs between the considered quality attributes.

Being a middle-sized focused research project (STREP), Q-ImPrESS bundles eight partners from five European countries. The Q-ImPrESS consortium consists of: FZI Research Centre for Information Technology in Karlsruhe (Germany) as project coordinator, ABB Corporate Research Centre in Ladenburg (Germany), Mälardalen University (Sweden), Politecnico di Milano (Italy), Charles University in Prague (Czech Republic), Itemis GmbH (Germany), Softeco Sismat S.p.A. (Italy), and Ericsson Nikola Tesla d.d. (Croatia). The project started in January 2008, is scheduled to run for 36 months, and has a total budget of 4.68 mil. EUR.

This paper focuses on the reverse engineering process used to extract the SAM from existing source code and presents selected analyses, while the scope of Q-ImPrESS is broader. The paper is structured as follows: Section II presents the reverse engineering approach and its sub-parts, Section III details on first experiments and experiences with the tool chain. Then, Section IV presents selected related work before Section V concludes the paper and shows future working directions.

II. REVERSE ENGINEERING PROCESS

The Q-ImPrESS reverse engineering tool chain comprises of multiple tools which are running consecutively to discover components, composite components, and interfaces together with behavioural models from source code and possibly additional information sources like deployment descriptors. Fig. 1 provides an overview on the tooling. The output models ultimately enable performance and reliability predictions (SAM incl. Behaviour), maintainability analysis (SAM), and, for demonstrating the formal capabilities of the SAM, checking protocol interoperability (Threaded Behaviour Protocols). Besides, the reverse engineered models help understanding a software architecture in terms of possible component abstractions.

The SISSy² tool is capable of extracting a Generalised Ab-

²The tool for Structural Investigation of Software Systems (SISSy) is an open-source tool for internal quality assessment of object-oriented software. See <http://sissy.fzi.de>

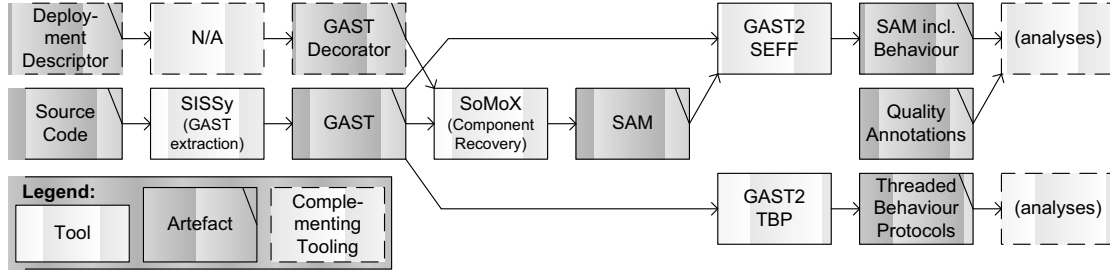


Fig. 1. Overview on the Q-ImPRESS reverse engineering tool chain

tract Syntax Tree (GAST) from source code. This GAST then serves as input for component recovery (SoMoX) and a transformation to Threaded Behaviour Protocols (GAST2TBP). SoMoX outputs components of the architecture model SAM. In an additional step, GAST2SEFF then adds a behavioural abstraction from GAST and SAM (providing component boundaries) to serve together with quality annotations (e.g. execution durations of internal actions) as a base for performance (e.g. Palladio [3]) and reliability analysis.

The Q-ImPRESS reverse engineering can be compared with ArchiRec [5] and Java2PCM [7], but goes beyond it: It is extendable by new input models (see Fig. 1 “Deployment Descriptor”), not bound to Java (C/C++ and Java are supported), capable of driving multiple prediction methods (e.g. performance and maintainability), fully integrated and automated in Eclipse, and does not rely on proprietary tools like Sotograph. The comparison is further discussed in Section IV.

GAST Extraction The purpose of the GAST Extraction step is to create a programming language-independent abstract representation of the source code, which allows a uniform handling of different software systems in later stages of the reverse engineering process. A GAST is a typical data structure constructed by a compiler frontend, which, despite its name, is not a tree but rather a graph. Beside the usual syntactic edges (tree edges), the GAST contains additional semantic edges, added during the semantic analysis and linking each use of a symbol to its definition.

The GAST meta-model, based on the QBench³ meta-model [16], contains first class entities for all major object-oriented and procedural language constructs, from high-level structuring constructs such as packages, through type definitions, function definitions, statements, variable definitions, down to the level of individual expressions. Furthermore, the meta-model also uses first class entities, called *accesses*, to represent resolved relations between other model elements, such as inheritance relations, function calls, or variable reads and writes. The meta-model stores the complete control and dataflow information, allowing an efficient implementation of different static analyses.

The extraction of the GAST model is fully automated and implemented as part of the SISSy tool. SISSy was designed to parse syntactically correct, but not necessarily complete

source code, written in Java, C, C++, or Delphi, and it does so successfully for large systems having one MLOC and above.

Component Recovery Component recovery from the extracted GAST model is taken over by SoMoX [1]. SoMoX evaluates multiple source code metrics and combines these metrics within a single weight, which is then fed into a clustering algorithm. During clustering, classes from the GAST model are aggregated and associated with components. Running several reverse engineering iterations results in increasingly abstract components. In SoMoX, component interfaces, provided and required interface ports are reverse engineered along with components and results in instances of the SAM. The SAM is the base for further analyses and defines component boundaries for the subsequent behaviour extraction.

Source code metrics and their combination to an overall result capture heuristics, which SoMoX uses for component recovery. Each metric is calculated for a component candidate comprising of a number of classes. Components are then, later in the clustering, composed from positively evaluated component candidates. Metrics are *abstractness* (how many abstract entities like interface and abstract classes are present), *instability* (how much does an component candidate depend on component internal and external classes and interfaces), *distance from the main sequence* (combines the previous metrics to reflect how well a balance between a abstract stable and concrete instable component is maintained), *name resemblance* (how well do the names of classes and interfaces fit together), *interface violation* (communication by-passing interfaces), *coupling* (dependencies and interconnections), *package mapping* (organisation in packages), *slice layer architecture quality (SLAQ)* (organisation in a slice and layers architecture style), *subsystem component* (summarises package mapping and SLAQ), and *directory naming* (how well do component candidates match to the directory naming). For details on the metrics, see [5], [11].

To take source code metrics to an advanced level, SoMoX combines multiple metrics and takes their interdependencies into account. For example, considering the similarity of package names of classes only makes sense if the classes are coupled at all. These interdependencies are being respected when calculating the overall result of a component candidate. Component candidates exceeding a certain threshold are then clustered into a single component. If there have been previous clustering iterations, components result in composite com-

³QBench was a research project funded by the German Ministry of Education and Research (BMBF), focused on internal quality assurance of evolving object-oriented systems. See <http://www.qbench.de>

ponents (encapsulating components of previous iterations). This allows higher-level component abstractions which can be traced back to lower levels.

Per project, SoMoX can be adjusted to project-specific requirements. Weights per metrics and the clustering threshold can be defined on a per-project base. Additionally, namespaces of libraries, class names indicating pure data objects or common pre and post fixes of class names (like “EJB_”) which do not indicate components belonging together, can be filtered out. This enables the creation of a focused component model which does not involve unnecessary details. Additional information sources like EJB deployment descriptors, can be easily integrated into SoMoX by providing an Eclipse extension for a pre-defined extension point.

Traceability is tackled by a decorator of the architectural model (SAM). Here, links to the original source classes are stored for each SAM entity. Through this mechanism, any later architecture-level analysis result can be mapped back to low level classes and methods.

GAST2SEFF Applying performance prediction approaches to reverse engineered models requires a representation of component behaviour. GAST2SEFF transforms behaviour present in the GAST to an abstract component-level behaviour. Here, for example any component-internal actions are reduced to single nodes in the behaviour model. Any control flow which is not affecting other components (no calls to other components) is abstracted away. The abstraction step is required since performance prediction approaches cannot handle full detail models.

GAST2TBP Source code representation in the form of GAST can be transformed to the formalism of Threaded Behavior Protocols (TBP, [10]). Basically, TBP models describe the behaviour of each component as visible from outside, i.e., as activity of the component at its provided and required interfaces. Having such a behaviour model for each component, one can verify (via application of model checking techniques) the compatibility of components utilising functionality of each other. For a reversed-engineered model, no incompatibilities should be discovered, however, the model forms a basis for model level changes and verification of their correctness with respect to communication with other components.

Quality Annotations Once a SAM including components, connectors, and behaviour has been created, additional quality annotations have to be provided in order to enable analyses of quality dimensions such as performance and reliability. Therefore, the Q-ImPRESS SAM provides options to specify such quality annotations. In case of performance, important quality annotations would be resource demands that occur in certain behaviour actions, or branch probabilities (if different branches incur different resource demands). In the case of reliability, it is necessary to specify failure probabilities for behaviour actions and resources. The decorated SAM is then input to analyses.

Currently, such quality annotations cannot be derived automatically from existing source code but have to be specified manually. Usually, analysts have to run analyses on the system

to retrieve these annotations.

Tooling: GAST2SEFF Currently, GAST2SEFF is implemented using Java and no transformation engine. A second implementation using QVT Relations [13] is under development.

Tooling: G-AST2TBP The transformation is implemented in the jAbstractor tool, which is now able to create a behaviour model for most inputs. The problematic constructs, which cannot be handled in an automatised way, include recursion, code of anonymous classes, and calling required interfaces from within an expression. If such constructs are present, a manual update of the model is required to avoid an over-abstraction of the model and to provide a better correspondence with the implementation.

SISSy and SoMoX tooling is captured by the experiments.

III. INITIAL EXPERIMENTS

In an initial experiment, the Common Component Modeling Example (CoCoME, [14]) has been used to gain insights into the quality of the Q-ImPRESS reverse engineering tool chain. CoCoME represents a distributed store trading system which covers a store from its cash desk line to the central enterprise reporting system on current sales. CoCoME has a total of 9,521 lines of code, 126 classes, and 21 interfaces. We compared the results of the reference decomposition of CoCoME documented in the architecture against the results of the reverse engineering; specifically the results of component recognition.

CoCoME comprises of a total of 23 components documented in the architecture. 2 of them are at the system-level (very coarse grained), another 4 components at the sub-system-level. Since the two components at the system-level are very coarse grained, we did not expect them being found. The Q-ImPRESS tooling reverse engineered a total of 20 composite components from the source code. Of these, 5 are components at a lower level than documented in the architecture. Besides the system-level components, the four sub-system-level components have not been recognised.

Of the cash desk line part of CoCoME, 8 out of 8 components were recognised correctly. Only the event bus which is represented as a kind of component in the architecture document was not recognised as a component. For the inventory part, most components were recognised correctly: One additional and undocumented testing component was discovered in the data access layer. From 3 inventory application components, 2 were found. The reporting component missing in the inventory application was assigned to the GUI, for which 5 instead of 2 components in the architectural document were found.

In total, of the 21 components at the sub-system-level and below, 13 components directly matched the expected output. Mainly for the sub-system-level and the GUI, the component abstraction did not match the expected results. Instead, components have been discovered, which are reflected in the source code (multiple GUI components) but are not documented in the architecture document. Although this is a

mismatch between expected and de-facto output, when having a look into the code, results are meaningful.

To adapt the reverse engineering tooling for the CoCoME project, we set up the reverse engineering run to consider only `org.cocome.*` packages to be part of the CoCoME core. Additionally, we advised SoMoX to blacklist `*TO` and `*Event` classes for component recognition. To not impact the results, the reverse engineering was performed fully automated. A complete reverse engineering run recovering components from source code using SISSy and SoMoX takes about 10 minutes for CoCoME.

IV. RELATED WORK

The Q-ImPRESS project is related to a number of reverse engineering approaches and partially makes use of them. A previous overview on Q-ImPRESS, which did not detail on reverse engineering, was published in [4].

The MoDisco project [12] aims at reverse engineering support in the Eclipse modeling context. By means of model transformations, a general reverse engineering infrastructure and reverse engineering process has been created. The ArQuE project (Architecture-Centric Quality Engineering, [9]) deals with “architecture-centric development and strategic quality engineering” and as such also involves reverse engineering activities like Q-ImPRESS. Both projects do neither emphasize component support nor prediction of quality attributes.

Internally, Q-ImPRESS uses the static source code analysis of SISSy [15], [16]. For Q-ImPRESS, SISSy has been extended to output the EMF-based GAST model. Component recovery of SoMoX bases on the work of Klatt [8] and the ArchiRec approach by Chouambe et al. [5]. Compared to ArchiRec, SoMoX does not rely any more on the proprietary Sotograph tool [6]. Besides, it is easily extendable by new metrics, easier to configure, bases on EMF models, and integrates with Eclipse.

As already discussed in Section II, Java2PCM by Kappler et al. [7] is comparable with GAST2SEFF. Q-ImPRESS goes beyond Java2PCM in not relying on components covering at most a single class. Since GAST2SEFF works on a language independent representation (GAST), it is able to reverse engineering any object oriented language supported by SISSy (currently Java, C/C++, Delphi).

V. CONCLUSION

In this paper, we have presented an overview of the Q-ImPRESS project with focus on the reverse engineering process. Currently, we have implemented the reverse engineering part up to transformations to specific models aiming at quality prediction and verification of communication correctness. We validated the process on the CoCoME application with positive results. The aforementioned tools are integrated into the Eclipse-based Q-ImPRESS IDE, which is freely available for download [2]. The next steps in the project include completing implementation of the transformations and prediction tools as well as their integration into the Q-ImPRESS IDE. Eventually, the entire tool chain will

be evaluated on demonstrators provided by industrial project members; the demonstrators are service-oriented applications of non-trivial size from various domains (telecommunications, control systems). The Q-ImPRESS IDE will be complemented by graphical editors for the SAM model.

ACKNOWLEDGMENT

The authors would like to thank all members of Q-ImPRESS for their contributions to this paper, during various project working sessions, and for their contribution to the very pleasant and constructive work environment in the project.

REFERENCES

- [1] SoMoX – Software MOdel eXtractor. <http://www.somox.org>.
- [2] EU project Q-ImPRESS, Quality Impact Prediction for Evolving Service-oriented Software, 2009. <http://www.q-impress.eu/>, last retrieved 2009-10-20.
- [3] Steffen Becker, Heiko Koziolok, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.
- [4] Steffen Becker, Mircea Trifu, and Ralf Reussner. Towards Supporting Evolution of Service Oriented Architectures through Quality Impact Prediction. In *1st International Workshop on Automated engineering of Autonomous and run-time evolving Systems (ARAMIS 2008)*, September 2008.
- [5] Landry Chouambe, Benjamin Klatt, and Klaus Krogmann. Reverse Engineering Software-Models of Component-Based Systems. In Kostas Kontogiannis, Christos Tjortjis, and Andreas Winter, editors, *12th European Conference on Software Maintenance and Reengineering*, pages 93–102, Athens, Greece, April 1–4 2008. IEEE Computer Society.
- [6] hello2morrow. Sotograph homepage. <http://www.hello2morrow.com/products/sotograph>. last retrieved 2009-10-12.
- [7] Thomas Kappler, Heiko Koziolok, Klaus Krogmann, and Ralf H. Reussner. Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering. In *Software Engineering 2008*, volume 121 of *Lecture Notes in Informatics*, pages 140–154, Munich, Germany, February 18–22 2008. Bonner Köllen Verlag.
- [8] Benjamin Klatt. Software Model eXtractor (SoMoX). study thesis, Universität Karlsruhe (TH), Software Design and Quality Group, Karlsruhe, Germany, January 2009.
- [9] J. Knodel, T. Mende, M. Leszak, F. Guder, G. Meier, C. Ruckert, and C. Schitter. ArQuE: Architecture-Centric Quality Engineering. In *13th European Conference on Software Maintenance and Reengineering, 2009. CSMR '09.*, pages 289–292, March 2009.
- [10] Jan Kofroň, Tomáš Poch, and Ondřej Šerý. TBP: Code-Oriented Component Behavior Specification. In Karin Breitman, Roy Sterritt, and Shawn Bohner, editors, *Proceedings of SEW-32, Greece*, Institute of Electrical and Electronics Engineers, 2009.
- [11] Robert Martin. OO Design Quality Metrics – An Analysis of Dependencies. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1994.
- [12] Hugo Bruneliere Jeff Gray Frederic Jouault Mikael Barbero, Jean Bezivin. Reverse Engineering in Eclipse with the MoDisco project. Talk at EclipseCon 2007, <http://www.eclipsecon.org/2007/index.php?page=sub/&id=3709>.
- [13] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (ptc/07-07-07), 2007.
- [14] Andreas Rausch, Ralf Reussner, Raffaella Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 2008.
- [15] Olaf Seng, Frank Simon, and Thomas Mohaupt. *Code Quality Management*. dpunkt Verlag, Heidelberg, 2006.
- [16] Mircea Trifu and Peter Szulman. Language independent abstract metamodel for quality analysis and improvement of oo systems. In *Proceedings of the 7th German Workshop on Software-Reengineering (WSR 2005), Bad Honnef, Germany*, volume 25 of *Softwaretechnik-Trends*, 2005.