Gemeinsamer Workshop der GI-Fachgruppen SRE und TAV

14. Workshop Software-Reengineering

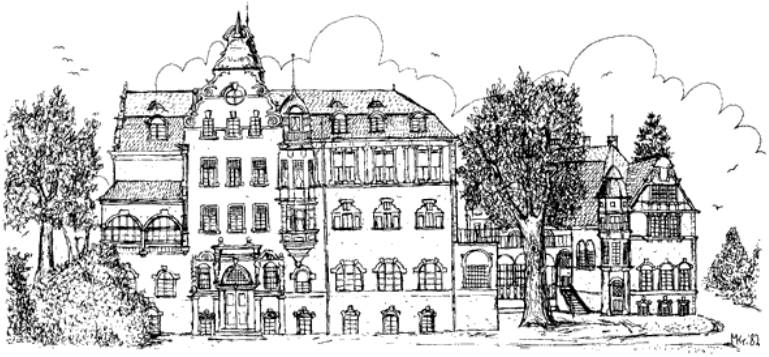
GI-Fachgruppe Software-Reengineering (SRE)

33. TAV-Treffen

GI-Fachgruppe Test, Analyse und Verifikation von Software (TAV)

Design for Future 2012

GI-Arbeitskreis Langlebige Softwaresysteme (L2S2)



Physikzentrum Bad Honnef

2.-4. Mai 2012

Programm

Mittwoch, 2. Mai 2012		
bis 11:00	**Anreise**	
11:00-12:30	**WSR: Comprehension, Deployment, Transformation**	
	Rainer Gimnich (IBM)	Information based Transformation
	Rebecca Tiarks (Uni Bremen) und Tobias Röhm (TU München)	Challenges in Program Comprehension
	Balthasar Weitzel (Fraunhofer IESE)	Towards Transparent Architectural Decisions for Software Deployment
	Jan Nonnen und Jan Paul Imhoff (Uni Bonn)	Analysing the Vocabulary to Identify Knowledge Divergence
12:30-13:45	**Mittagessen**	
13:45-15:30	**WSR: Clones**	
	Jan Harder (Uni Bremen)	Code Clone Authorship - A First Look
	Francesco Gerardi (HS Reutlingen) und Jochen Quante (Robert Bosch GmbH)	Type 2 Clone Detection On ASCET Models
	Saman Bazrafshan (Uni Bremen)	Late propagation of Type-3 Clones
	Mikhail Prokharau (Uni Stuttgart)	Aspects of Code Pattern Removal
	Torsten Görg (Uni Stuttgart)	A Model-Based Approach to Type-3 Clone Elimination
15:30-16:00	**Kaffeepause**	
16:00-18:00	**Design for Future**	
	Martin Küster und Benjamin Klatt (KIT)	Leveraging Design Decisions in Evolving Systems
	Marvin Grieger, Baris Güldali und Stefan Sauer (Uni Paderborn)	Sichern der Zukunftsfähigkeit bei der Migration von Legacy-Systemen durch modellgetriebene Softwareentwicklung
	Steffen Lehnert (TU Ilmenau) und Matthias Riebisch (Uni Hamburg)	Tackling the Challenges of Evolution in Multiperspective Software Design and Implementation
	Timo Kehrer, Udo Kelter (Uni Siegen) und Gabriele Taentzer (Uni Marburg)	Integrating the Specification and Recognition of Changes in Models
	Sven Euteneuer und Daniel Draws (SQS AG)	Digital Preservation: New Challenges for Software Maintenance
18:00-18:30	**Fachgruppensitzung der GI-Fachgruppe Software Reengineering**	
	• Genehmigung der Tagesordnung und des Protokolls der letzten Fachgruppensitzung • Bericht der Fachgruppenleitung • Planung weiterer Veranstaltungen • Sonstiges	
ab 18:30	**Abendessen, anschließend traditioneller Spaziergang**	

Donnerstag, 3. Mai 2012

08:45-10:15	**WSR: Identifiers and Static Analysis**	
	Holger Knoche, Wolfgang Goerigk (b+m Informatik AG), André van Hoorn und Wilhelm Hasselbring (Uni Kiel)	Automated Source-Level Instrumentation for Dynamic Dependency Analysis of COBOL Systems
	Fabian Beck, Alexander Pavel und Stephan Diehl (Uni Trier)	Interaktive Extraktion von Software-Komponenten
	Sören Frey, André van Hoorn, Reiner Jung, Benjamin Kiel und Wilhelm Hasselbring (Uni Kiel)	MAMBA: Model-Based Software Analysis Utilizing OMG's SMM
	Uwe Erdmenger (pro et con GmbH)	Oberflächenmodernisierung mit MaTriX
10:15-10:30	**Kaffeepause**	
10:30-12:30	**WSR/TAV/DFF: Eingeladene Vorträge**	
	Sprecher der FG SRE und TAV sowie des AK L2S2	Kurze Vorstellung und Begrüßung
	Andreas Zeller (Uni des Saarlands)	Experimentelle Programmanalyse
	Ina Schieferdecker (FU Berlin/Fraunhofer FOKUS)	Testing of ICT in Urban Management Systems – A Research Perspective
12:30-14:00	**Mittagspause**	
14:00-15:30	**WSR/TAV/DFF: gemeinsame Sitzung**	
	Jochen Quante (Robert Bosch GmbH)	When Program Comprehension Met Bug Fixing
	Christian Becker und Uwe Kaiser (pro et con GmbH)	Test der semantischen Äquivalenz von Translatoren am Beispiel von CoJaC
	Harry Sneed (ANECON GmbH)	Validierung der funktionalen Äquivalenz konvertierter JAVA Programme durch einen dynamischen Source-Abgleich
15:30-16:00	**Kaffeepause**	
16:00-18:00	**WSR/TAV/DFF: Gemeinsame Sitzung**	
	Uwe Hehn und Sebastian Kern (Method Park Software AG)	Systemtest im agilen Entwicklungsprozess
	Daniel Simon und Frank Simon (SQS AG)	Fuzzing: Testing Security in Maintenance Projects
	Daniel Speicher, Jan Nonnen und Andri Bremm (Uni Bonn)	Smell Museums as Functional Tests for Static Analyses
	Harry Sneed (ANECON GmbH)	Validierung der funktionalen Äquivalenz konvertierter JAVA Programme durch einen dynamischen Source-Abgleich
ab 18:30	**Conference Dinner**	

Freitag, 4. Mai 2012		
09:00-10:30	**WSR/TAV: Eingeladene Überblicksvorträge**	
	Rainer Koschke (Uni Bremen)	Statische Code-Analysen für die Qualitätssicherung
	Mario Winter (FH Köln)	Überblicksvortrag Test
10:30-11:00	**Kaffeepause**	
11:00-12:30	**WSR: Software Quality**	
	Andreas Fuhr, Daniel Bildhauer, Jürgen Ebert, Judith Haas, Volker Riediger (Uni Koblenz), Marcus Rausch, Johannes Bach, Max Doppler, Daniel Höh und Martin Schulze (Debeka)	Einführung von COBOL-Wartbarkeits-Metriken bei der Debeka
	Jan Jelschen und Andreas Winter (Uni Oldenburg)	A Toolchain for Metrics-based Comparison of COBOL and Migrated Java Systems
	Nils Göde und Florian Deissenboeck (CQSE GmbH)	Delta Analysis
	Moritz Beller und Elmar Juergens (TU München)	How Strict is Your Architecture?
12:30-13:30	**Mittagspause und Abreise**	

Information based Transformation

Rainer Gimnich
IBM Software Group
Information Agenda Team Europe
Wilhelm-Fay-Str. 30-34, D-65936 Frankfurt
gimnich@de.ibm.com

Abstract
Information is increasingly seen as the major asset of an enterprise. Many new business goals are information-based, e.g. to achieve a 360° view of the customer, to optimize risk and compliance management, to achieve a higher level of operational efficiency.
In order to support information based transformation, an enterprise architecture method is beneficial, as well as its customization and supporting tools, with a focus on 'data reengineering'.
This paper describes an architecture approach to information based transformation and some practical experience.

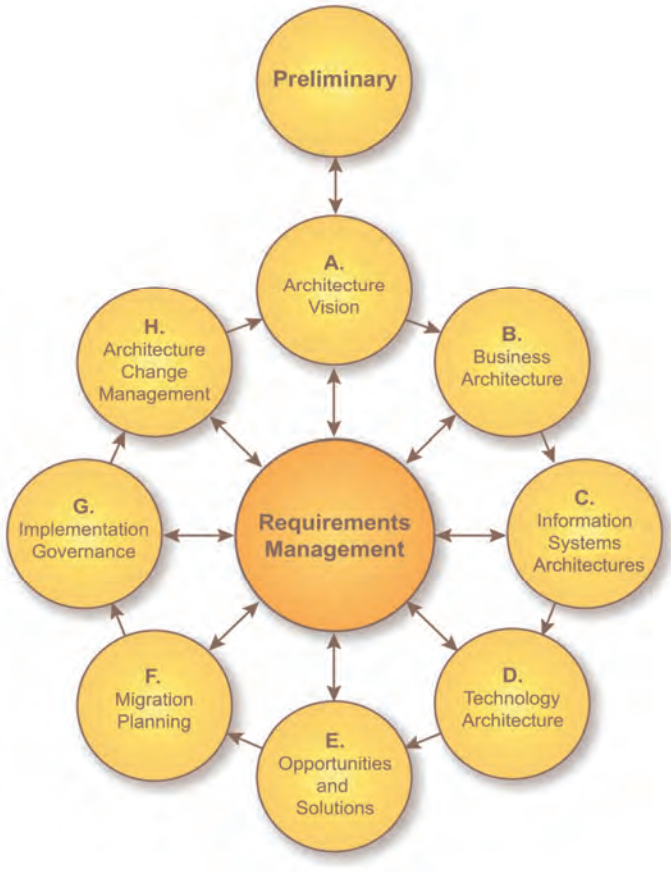
1. Transformation: why and what?

'The business of IT is business.' This statement came up with the first Service Oriented Architectures (SOA) some 10 years ago. Essentially, IT has no value in itself but as a response to business needs and problems. As markets and businesses change, business strategies and their implementation in the form of IT need to change, with an ever shorter 'cycle time'.
To control and support this **Enterprise Transformation**, an Enterprise Architecture approach has proved useful. This is strongly driven from business needs, includes business architecture, provides the links to other architecture domains (application, data, technology), provides architecture governance and maximizes reuse, including reference architectures.
There are good reasons to strengthen the **Information based** transformation approaches in the Enterprise roadmap projects:

- They help realize very high business value in short timeframes.
- They rely on recently advanced technologies.
- They provide a good complement to process based approaches, which have been predominantly used in SOA transformations [1].

2. Using TOGAF™ (The Open Group Architecture Framework)

TOGAF™ [2] provides several important assets in this context: an Architecture Capability Framework (including architecture maturity and architecture governance) and an Architecture Development Method (ADM). The ADM is a generic approach that is meant to be customized to the specific enterprise:



The Information Systems Architecture phase includes several steps regarding data architecture:

- The baseline data architecture is developed, and this may include analyzing the existing landscape of data, data bases, data models, data warehouses, master data systems, content systems, etc. and representing the findings in suitable artifacts.
- The target data architecture is developed in a similar way. Here, data reference architectures and industry specific data models may be of additional benefit.

- Roadmap components are defined, resulting from a gap analysis between target and baseline data architecture. The roadmap components from Phases B to D are consolidated in Phase E (Opportunities and Solutions) and provide the basis of the Architecture Roadmap and the Implementation and Migration Plan. Where the distance between Baseline and Target Architecture is deemed too big, so-called Transition Architectures are defined. They enable an incremental development and deployment of the intended target solution.

3. Using tools for data analysis, integration and transformation

There are a number of tools on the market which help in analyzing, designing, integrating and migrating data architectures:

- **Glossary tools**: to create and manage business vocabulary and relationships, related to physical sources. This tool provides mappings of physical data to an enterprise-wide system of business terms and classifications.
- **Data Discovery tools**: to discover data transformation rules and heterogeneous data relationships. These provide business insights and reduce project risk.
- **Data Architecting tools**: to design and manage enterprise data models and to enforce model conformance to enterprise standards. This speeds design activities and populates the Glossary from model terms.
- **Information Analyzers**: to analyze source data quality and monitor adherence to integration and quality rules. These tools monitor quality metrics over time for compliance and create business confidence in the data.
- **Capturers** are tools to capture design specifications and accelerate translation into data integration projects. They accelerate development and provide a centralized management of specifications.
- **Metadata Management** tools visualize and trace information flows across the enterprise landscape ('data lineage'). They help in understanding the impact of making changes to the information environment, in avoiding system disruptions and in providing audit information for data governance.

4. Project examples

Looking into on-going Information based Transformation projects is important in order to

- validate the approach described above,
- generate deeper insight into the transformation complexity in practice,
- identify new areas of research and/or knowledge transfer (potential research topics arising from industry projects are mentioned in [3]).

The first practice example is the **redesign of the customer relationship management (CRM)** area of a large European company.

Here, a dedicated TOGAF™ ADM cycle is used to build the Architecture Landscape. Then several development cycles address the major transformation needs:

- Redesign of the advisory processes (both branches and call center) providing all customer and product data where they are required and in near realtime.
- Transforming the 'scattered' and insufficient master data stores into an enterprise-wide, SOA based master information management system. This includes introducing a governance structure for both information and architecture.

The second practice example addresses major extensions to the **self-service capabilities** of a large credit card company. Currently, the company's customers can only see their transactions and produce simple reports. A new business strategy requires more detailed reports, categorization of transactions, added value services such as related products and partner discounts based on transaction content, etc.
In order to achieve the goals, the existing IT landscape needs to be transformed, including a flexible data warehouse, with more data and relationships, access to unstructured data and additional customer data within and outside the enterprise.
A number of transformations are required: architecture transformations, model transformations, data transformations, etc. All of these can be governed using TOGAF™ and specific tools – most importantly for handling enterprise models and model transformations and for performing data transformation and replication.

5. Outlook

There is a growing demand for Information based Transformation projects, responding to business needs and introducing new analytics capabilities, including the use of 'big data' and Watson [4] technologies in an enterprise context.
The intention is to apply and customize proven methods and tools for Enterprise Architecture, SOA and Reengineering to efficiently build the required enterprise transformations.

References

[1] R. Gimnich et al.: PIDT - Process and Information Driven Transformation. IBM internal report, 2012. Parts to be published.
[2] The Open Group: TOGAF™ Version 9. http://www.opengroup.org/togaf/
[3] R. Gimnich, A. Winter: Software Evolution in Forschung und Praxis. OBJEKTspektrum - Business Application Modernization, 2010.
[4] IBM: Watson. http://www-03.ibm.com/innovation/us/watson/index.html

Challenges in Program Comprehension

Rebecca Tiarks
University of Bremen
beccs@tzi.de

Tobias Röhm
Technical University Munich
roehm@in.tum.de

March 8, 2012

Abstract

Program comprehension as a subtask of software maintenance and evolution consumes about half of the time spent by the developers who have to explore a systems' source code to find and understand the subset of the code which is relevant to their current task. The problems encountered during the comprehension process influence the time spent on program comprehension to a great extent. Although many empirical studies have been conducted in the field of program comprehension, only little is known about the challenges developers face when trying to understand a software system. This paper reports on an observational study of 28 professional developers, investigating their behaviour with respect to the occurring problems.

1 Introduction

During software maintenance and evolution, program comprehension is a major subtask, which is driven by the need to change software. Research efforts aiming at addressing challenges of program comprehension can be characterized by both the tools that are used to assist a comprehension task as well as the cognitive theories that provide explanations on how developers understand software. Although research in the field of program comprehension has considerably evolved over the past 20 years and many theories have been proposed to explain how programmers may comprehend software, there are still open issues that require further investigation. A survey of Storey [5] provides an overview of the field including future trends. Still, only little is known about the strategies employed by the developers and the problems they encounter during the comprehension process. Whenever humans are involved, empirical studies such as surveys, case studies or controlled experiments are a rigorous means of empirical research. In the field of program comprehension, a number of studies have been published that involved human subjects. Those studies suffer from limitations calling for a deeper, up-to-date examination. According to [2] much of this work only used small programs or novice users. In order to analyze developers' behavior and problems, we undertook an observational study of 28 professional developers from seven different companies [4]. The goals of the original study were to investigate the strategies and tools a developer uses and the information the developer interacts with. The study showed that the problems faced by the developer during program comprehension are worth being further investigated. Therefore the research question we are exploring in this paper is the following:

- What problems do developers encounter during program comprehension?

With our report, we aim at pointing out some of the problems we experienced in our study to guide future research and tool development.

2 Study Method

The goal of our study was to get a deeper insight into how program comprehension is done in practice. Therefore we chose a combination of observation and interview to approach our goal. We observed developers in their real work environment, and the tasks the participants worked on were chosen by themselves. All participants were professional developers from seven software development companies. We chose participants with different tasks, different project roles, different experiences, and from different company sizes. The observation and the consecutive interview took 1.5 hours in total. Details about the study can be found in the conference paper [4].

3 Problems

During the study, we observed occurring problems and in the interviews, we asked the participants which problems they encounter when trying to understand a software system. That means that the reported problems are not only the problems we noticed during the observation but also the problems explicitly stated by the developers themselves.

Understanding somebody else's code When developers implement a solution for a specific problem, they construct a mapping from a problem domain to a programming domain. Between these domains may exist several intermediate domains and all of these domains have to be reconstructed when trying

to understand the corresponding piece of code. Often this reconstruction is hypothesis-driven. We observed some participants asking and answering questions leading to informal hypotheses which were then verified by the developers. This problem of discovering individual human-oriented concepts is often called *concept assignment problem* [1]. During the interview, many of the participants identified "understanding somebody else's code" as one of the major challenges in program comprehension. One participant argued that "others have another style and other way of thinking", which makes it difficult to reconstruct the mappings mentioned above.

Search and History Several studies have shown that developers quickly forget details when they move to a different location within the source code or even a different task. During the observation, we noticed some developers that wrote down notes either on a piece of paper or used an editor as temporal memory. Some used these notes as a search history and wrote down which variable or function names they had already searched for. Nevertheless some participants searched for the same function or variables names more than once because they did not remember every detail of the result or to validate their hypothesis about that specific piece of code. During the interview, some participants mentioned that they often have to execute the same search many times and that a tool that keeps track of the searches and problem-solving sessions would be extremely helpful. This finding matches with suggestions by Storey [5].

Loosing context because of interrupts, switching between tasks and window changes Discussions, meetings and phone calls are a necessary part of software development. Interrupts disturb developers during their activities. Developers being interrupted lose their current focus and context and it takes them extra time to recover from the interrupts and previously gathered knowledge may be lost. Most of the interrupts we observed were initiated by the developers themselves and were related to knowledge or experience exchange. Another problem we observed were the interrupts caused by window changes or switching between tasks. One participant, for example, had to switch between the development environment and the editor in which he wrote the documentation quite often. After every window change, he had to recover the information needed for the current task.

Finding suitable breakpoints when working with the debugger When trying to locate a bug, many developers used the debugger to inspect the state of the application [3]. Therefore they have to find suitable breakpoints in order to reproduce the data and control flow of the program. We observed several participants that encountered problems to find the *right* breakpoints and during the interview they approved that deciding where to set a breakpoint is "extremely difficult". The strategy they applied was to set a lot of breakpoints in the beginning wherever they felt it could be useful, starting the debugger and diminish the amount by removing those breakpoints that turned out to be irrelevant. That means that they had to inspect a lot of breakpoints to decide whether they were useful or not, which caused significant overhead.

4 Conclusion

The task of understanding code becomes more difficult as software is constantly getting more complex. In this paper, we report on problems developers face when trying to understand a software system. The results reveal interesting aspects of program comprehension that can guide future tool development. We found that one of the main problems is to understand somebody else's code, which is due to the fact that developers implement a solution by applying individual human-oriented concepts. These concepts vary with the type of task, developer personality, the amount of previous knowledge and the type of application, which makes it difficult to reconstruct. A solution would be to represent the implemented concepts in a uniform way. Another problem describes the loss of parts of the mental model. This applies to searches as well as to interrupts. We suggest that tools should implement features to support rediscovery of knowledge by keeping track of the searches and problem-solving sessions. Further as far as interrupts are concerned, a better integration of different tools could help to prevent frequent window changes that distract the developer. With respects to debugging breakpoints we found that developers start the debugging process by inserting a vast number of breakpoints and refine this set during the comprehension process by removing irrelevant breakpoints. To reduce this overhead tools could suggest more appropriate breakpoints according to the context of the current task. Future work is necessary to get further insight and to understand the generality of these findings.

References

[1] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proc. of the 15th ICSE*, pages 482–498, 1993.

[2] J. I. Maletic and H. Kagdi. Expressiveness and effectiveness of program comprehension: Thoughts on future research directions. In *FoSM 2008.*, pages 31–37, Oct. 2008.

[3] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23(4):76–83, 2006.

[4] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? 2012. accepted for publication.

[5] M.-A. Storey. Theories, tools and research methods in program comprehension: *past, present and future. SQJ*, 14:187–208, 2006.

Towards Transparent Architectural Decisions for Software Deployment

Balthasar Weitzel
Fraunhofer IESE
Fraunhofer-Platz 1, 67663 Kaiserslautern
Email: balthasar.weitzel@iese.fhg.de

Abstract:
The operation of large scale information systems requires investment into hardware infrastructure and causes running cost for keeping it in a productive state. This especially applies in an enterprise environment where also expenses for software licenses costs or penalties for downtime occur. The deployment of software influences these costs both in their amount and their composition. In order to optimize them a transparent view on these costs and the deployment is mandatory. In this paper, we present a conceptual model of deployment. The model is populated by reverse engineering of deployment descriptors but as well uses runtime traces and usage profiles. We envision – having both made explicit on an architectural level – a comprehensive decision making and optimization of software deployment.

1. Introduction

The development of software system (i.e., writing high-quality source code) is a challenge – but only one side of the coin. The other side of the coin – and nonetheless challenging – is the operation of the software system.
Operating a software system causes cost of different types. There are fixed costs like procurement or replacement of infrastructure. Additionally running costs for the operation as such in terms of energy, staff and software license cost, but also penalties for downtime may apply if certain quality levels of service are not achieved.
Different deployment alternatives influence both the total cost but also individual matters of expense. For example, the decision about the allocation of computation effort in a distributed information system can significantly raise or lower the investment costs for a central server. Considering cloud services, the running costs can be varied significant.
Such decisions are made on an architectural level in conjunction with various others architectural concerns. Architecture defines the boundary for possible deployment alternatives and enables to reason about their impact. For achieving such a holistic reasoning it is necessary to connect the deployment view of the systems architecture to other views, such as structural or behavioral ones.
When optimizing existing systems, deployment decisions already made have to be handled and balanced with potential migration effort. Historic data regarding operation characteristics and usage profile are available.
We envision that having an integrated view on the relevant aspects of deployment facilitates conscious architectural decision making. The planning of migration or change effort is supposed to be more reliable.
Our experiences in industry show a limited awareness of the influence of deployment on operation cost. Nevertheless, nearly all industry partners are able to tell a story where such a decision caused high cost (e.g., replacing a third party component to reduce license cost due to usage on multiple servers or enhancing network connections to locations to handle the traffic a new feature introduced). Decisions are typically made implicitly or have been deliberated once (at the time of the first release) and remained unchallenged since then (although the system evolved).

2. Related Work

Deployment on architectural level is represented sparse in literature. An example is presented in [1], where the distribution of the software is in focus. Analysis of cost factors is not considered, the overall goal is to present a specific technology.
The need to capture the complex aspects of building a software system is mentioned in [4], with a focus on the linkage of development and deployment view.
An overview of deployment technologies is presented in [2], where again the infrastructures used to realize specific deployment alternatives are considered.

3. Conceptual model of deployment

We defined a conceptual model of deployment on an architectural level that is very close to the UML notation of deployment aspects. The core elements of the model and their relationship are depicted in Figure 1. In general deployment is about mapping

the *software elements* onto *hardware elements*. This distinction is represented by the two abstract software and hardware model elements which are further refined. On the hardware side there are *nodes* which are *physically linked* to each other or *grouped* together. The part of the model that represents the software is divided into *software execution environments* (like operating systems, application servers, database management servers, etc.) and the *executables* that are handled by them. An executable contains one or more modules and can thus also have *logical links* to others.

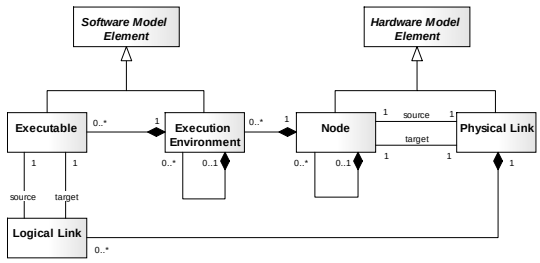


Figure 1: Core Conceptual Deployment Model

The connection between hardware and software is on the one hand represented by having the execution environment deployed on a physical node. On the other hand there are logical links between executables that are mapped to physical links between nodes.

Having this conceptual model cost factors can be introduced with the concept of profiles for expressing characteristics of a specific element. We identified four types of *profiles* that can hold expected or measured data:

- *Module Resource Utilization*: Load that a component causes
- *Executable Resource Utilization*: Aggregation of load caused by modules of an executable
- *Resource Capability*: Capacity of a hardware resource
- *Software Usage*: Typical usage behavior of a part of the software system by a group of users

4. Envisioned support for architects

Our goal is to support decision making on basis of an explicit deployment of a software system. In case of optimizing existing systems we can analyze historical usage data of the system. This enables us to provide the software usage profiles with heuristics about the typical usage behavior.

Having also runtime traces at hand will give the possibility to populate the three other types of profiles with data (module and executable resource utilization, resource capability). This highly depends on the granularity of these traces.

Finally analyzing deployment descriptors and thus reengineering the current deployment of the system allows giving an architectural view on the deployment which is enhanced by sound information gained from the past experience.

It is not mandatory to reveal these facts by automated reverse engineering tools, a manual modeling based on expert knowledge should be possible, too.

For optimizing a deployment an architect has the possibility to develop new deployment alternatives by analyzing the existing one. If the reduction of cost is in focus main cost drivers can be identified easily and their cause could be traced back to an architectural decision.

Alternative deployment options can be defined based on the model that represents the current deployment of the system. Such new options can consist of changes in the software itself, in the hardware landscape or in the mapping between them. The characteristics of such an alternative can be evaluated based on the different profiles that are available in the model and that are based on historic experience. Such characteristics are on the one hand related to resource demands but the main focus is on cost behavior.

If the model and the profiles are tool supported an easy explorative way of finding and automatically analyzing new deployment alternatives can be chosen. A virtual “monitoring” of the intended deployment option gives direct feedback on any changes that are performed.

5. Research Agenda

The conceptual model we presented needs to be evaluated by applying it in industrial case studies to ensure that it is expressive enough to represent multiple technologies that are used but at the same time is simple enough to be accepted in practice. Especially the concept of profiles is at an early stage and needs further refinements. To facilitate the application of the approach tool support is needed, currently only a prototypical realization of reverse engineering of deployment descriptors has been realized.

We received promising feedback when presenting our vision to industrial stakeholders, which motivates us continuing this line of research.

References

[1] Malek, S. et al.: An Extensible Framework for Improving a Distributed Software System's Deployment Architecture; Software Engineering, IEEE Transactions on; 2012

[2] Carzaniga, A. et al: A Characterization Framework for Software Deployment Technologies; Technical Report; Dept. of Computer Science, University of Colorado; 1998

[3] Object Management Group: UML 2.0 Specification; 2005; http://www.omg.org/spec/UML/2.0/

[4] Tu, Q.; Godfrey, M.W.; The build-time software architecture view; Proceedings of ICSM; 2001

Analysing the Vocabulary to Identify Knowledge Divergence

Jan Nonnen, Paul Imhoff

University of Bonn
Computer Science III
Bonn, Germany

{nonnen, imhoffj}@cs.uni-bonn.de

Keywords: vocabulary, active vocabulary, vocabulary evolution, software evolution, history mining, program comprehension

1 Introduction

During the development of a project, words used in source code add up to a big vocabulary, which may lead to a divergent word-understanding and word-knowledge between developers. Even the drop out of a single developer may lead to a big loss of knowledge about words and their meaning. By keeping track of the active developers vocabulary one is able to identify and react upon such situations. In this work we propose a way to identify these by analysing the words contained in identifiers obtained through the commit history in a version control system.

A person's vocabulary contains all words that they have used or have knowledge about, and can be subdivided in an active and passive part. The active side are all the words that the person uses in speech or in written form. In contrast, the passive vocabulary contains all words that one knows. The content of the vocabulary constantly changes. On the one side, it increases due to knowledge gaining, e.g. by reading books, and on the other side it decreases due to forgetting words.

Using the commit history of a project one is able to compute the active vocabulary of each developer in the team and to verify if a shared team language can be identified.

This work represents an excerpt of the study presented by the authors in [6].

2 Related work

In programming not only the code structure can be used to express concepts, but also the identifiers carry semantics for a developer [2]. In [8] Deißenböck and Ratiu considered the knowledge in source code to be represented by the identifiers and the code structure.

Abebe et al. [1] studied changes of the team vocabulary by analysing two software systems. Their results indicate that the code size increases faster than the size of vocabulary. Thus developers tend to use a low word variance for identifiers.

Hindle et al. [5] analysed commit comments and used a time windowed topic analysis to locate trending topics in a project.

Grant et al. [3] compared two techniques for analysing topic evolution. Topics are collections of words that co-occur frequently in the project history. They could identify global changes in their evolution visualisation.

3 Methodology

Our current approach utilizes the Git source code management system to obtain the history of a project. For each commit we calculate the changes based on its parent. This so called *change set* consists of the time and the changed lines of code.

From the change sets we preserve only identifiers. Afterwards, every identifier is splitted into constituent words using camel case rules (e.g. 'addQueryResults' gives {add, query, results}). These words form the used vocabulary of the commit. In our current approach we do not apply normalization on the words, e.g. normalizing 'aborted' to 'abort' and we do not remove non dictionary words, e.g. 'i'. We use this to also analyse how word forms evolve over time.

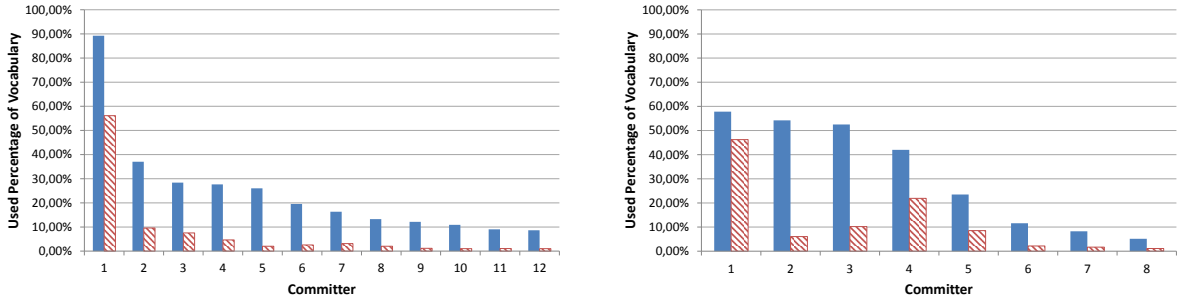
Our history model H is a set of tuples (c, a, w), where c is the commit time, a the author, and w the word. The *individual vocabulary* $V(a)$ of committer a can then be defined as $V(a) = \{w \mid \exists(c, a, w) \in H\}$.

4 Study and Results

For a study of vocabulary evolution we analysed two projects: JGit[1] and Cultivate[2]. JGit is an implementation of Git in Java and is in development since 2006. Cultivate is a static code analysis project for Eclipse that is developed at the University of Bonn and is in development since 2003.

For both projects we obtained a repository containing the commits of 25 months, see Table 1.

[1] http://www.eclipse.org/jgit/
[2] http://roots.iai.uni-bonn.de/cultivate



(a) JGit (b) Cultivate

Figure 1: Committers and their percentage of the used vocabulary (solid) in relation to the percentage of commit sizes in words (slashed).

Interestingly, the number of commits was higher in Cultivate (970) than JGit (695). Still the size of commits in JGit was bigger. Thus, the commits in Cultivate were more frequent, but also smaller. Using an observation from Hattori and Lanza [4], this means that in JGit more new features were added and the development in Cultivate focussed more on maintenance.

	JGit	Cultivate
Committers	52	15
Commits	695	970
Vocabulary Size	3391	1872

Table 1: Comparison of JGit and Cultivate between October 2009 till November 2011.

RQ: Does the knowledge of committers in both projects differ?

Figure 1 visualises how much of the total vocabulary each committer used and how many words she used in her commits. In JGit a single committer used nearly 90% of the total vocabulary, the second highest committer used only 36%. In contrast to this, in Cultivate three committers have similar vocabulary percentage of over 50%. This could indicate a single point of knowledge in the JGit project. More research is required in identifying the reasons for the disparity between both projects and whether this really represents a threat to the project health. An extension of the performed evaluation to more projects should provide a higher sample set. This may then be used to explicitly define the conditions under which a divergence occurs.

5 Conclusion and Future Work

In this work we presented the novel idea to detect knowledge divergence through analysis of the individual vocabulary usage of committers in source code. Also, we presented a study on two projects on which we evaluated our research question. We were able to identify a single point of knowledge in form of a disparity of vocabulary usage in the JGit project.

In the future we want to analyse how far program comprehension can be improved by visualising trending or stagnating words and showing each developer words that he has not used, but that all others have. These visualisations could improve the overall project comprehension and provide a knowledge improvement.

Further these words can be linked with an introduction location proposed by us in [7]. Introduction locations are locations in source code that show the meaning of a certain term. Information about "interesting" words for a developer in combination with an introduction location for each word is a seamless integration of both approaches.

References

[1] S. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol. Analyzing the Evolution of the Source Code Vocabulary. In *Proc. of the European Conference on Software Maintenance and Reengineering*, 2009.

[2] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proc. of the Centre for Advanced Studies on Collaborative Research*, 1998. Proc. of the 1998 conference of the Centre for Advanced Studies on Collaborative Research.

[3] S. Grant, J. Cordy, and D. Skillicorn. Reverse Engineering Co-maintenance Relationships Using Conceptual Analysis of Source Code. In *Proc. of the Working Conference on Reverse Engineering*, 2011.

[4] L. Hattori and M. Lanza. On the nature of commits. In *Proc. of the International Conference on Automated Software Engineering-Workshops*, 2008.

[5] A. Hindle, M. Godfrey, and R. Holt. What's hot and what's not: Windowed developer topic analysis. In *Proc. of the International Conference on Software Maintenance*, 2009.

[6] J. Nonnen and P. Imhoff. Identifying Knowledge Divergence by Vocabulary Monitoring in Software Projects. In *Proc. of the European Conference on Software Maintenance and Reengineering*, 2012.

[7] J. Nonnen, D. Speicher, and P. Imhoff. Locating the Meaning of Terms in Source Code - Research on 'Term Introduction'. In *Proc. of the Working Conference on Reverse Engineering*, 2011.

[8] D. Ratiu and F. Deissenboeck. Programs are Knowledge Bases. In *Proc. of the International Conference on Program Comprehension*, 2006.

Code Clone Authorship — A First Look

Jan Harder
University of Bremen, Germany
harder@informatik.uni-bremen.de

Abstract

Code clones are said to threaten the maintainability of software systems. Changes to one cloned code sequence likely require propagation to its copies. Proper change propagation may be more difficult when the clones are created and maintained by different authors. We present an approach to track the authors of code clones and report on a first case study. The results indicate that the number of contributing authors has an effect on the probability that clones are changed and the probability that these changes are carried out consistently.

1 Introduction

The effect of clones on maintainability has been analyzed by various studies in the recent past. On the one hand, their results indicate that clones may cause maintenance problems. On the other hand, they show that clones can not be regarded as a threat to maintenance in general. This raises the question under which conditions clones threaten maintainability. One factor that makes clones harder to maintain may be the number of developers who contribute to a clone. For instance, when a developer copies code that is maintained by another developer, it may be more likely that these clones do not change consistently in the future. We previously discussed that the management of clones may be more difficult if multiple authors are involved. [3].

2 Authorship

This section describes our approach to track the authors of source code clones. First, we will define our tracking for source code in general and then apply it to code clones.

Code Authorship. To determine the authors of a source code sequence, we use a token-based approach that is based on the metadata of the *subversion* source code management system. The general idea is to annotate all tokens in all file revisions with the user who added them or modified them last.

Let $r_0, \ldots, r_n \in R$ be the set of revisions we analyze and $F(r_i)$ the set of source code files in revision r_i. Our approach consists of two steps. First, we build the token stream $T(f, r_0)$ for each source code file $f \in F(r_0)$. For each token $t \in T(f, r_0)$ we define the author of the token as $author(t)$, which is the user who committed r_0 to the repository. Second, we iterate over all upcoming revisions r_i with $0 < i \leq n$. If a file f was changed, we use the *Longest Common Subsequence* algorithm on its old token stream $T(f, r_{i-1})$ and its new token stream $T(f, r_i)$. The result provides the tokens that have been deleted and added in f in r_i. If a token is replaced with another one, this will be reported as a deletion and an addition. Let $TA(f, r_i)$ be the set of tokens that have been added to f in r_i ($TA(f, r_i) \subseteq T(f, r_i)$). For all $t \in TA(f, r_i)$ we define $author(t)$ to be the user who committed r_i to the repository. For all other tokens that were not changed $u \in (T(f, r_i) - TA(f, r_i))$ we define $author(u)$ as the author the token was assigned to in r_{i-1}.

The first step that handles the first revision r_0 must be changed if r_0 is not the initial revision of the project and contains source code that already has a history. This is the case when r_0 is not chosen as the first revision in the repository or if the repository was initialized with existing code (i.e., in a migration from another SCM system). In this case all tokens in all $f \in F(r_0)$ are assigned the artificial user *unknown* and the authorship information must be built from the following revisions until a significant amount of tokens has been mapped to their authors. This, however, does not apply for this study where we deliberately chose r_0 as the very first version of the system's history.

A sequence of source code S is a subsequence of the tokens from a file in one particular revision: $S \subseteq T(f, r_i)$. The tokens $t \in S$ may be assigned to different authors. Thus, a code sequence has a list of authors $a_1, ..., a_n$ of which each a_i with $1 \leq i \leq n$ contributed a subset of the tokens $S_{a_i} \subseteq S$. For this study we define the *main author* of a sequence as the author a_i who owns most of the tokens in S, that is the one with the highest $|S_{a_i}|$. If two or more authors contributed the same number of tokens, one author is chosen randomly as the main author.

One of the advantages of our token-based approach lies in its robustness against changes to the code layout. Line-based differencing techniques would report changes when a code sequence was only reformatted. In this case code formatting, which is often applied automatically, would change the authorship.

Clone Authorship. Source code clones are code sequences that are copied in different locations. We call

these sequences *clone fragments*. All clone fragments that are identical to each other from a *clone class*. Using our definitions for code authorship we can also define the authors of clone fragments. This is straightforward, because fragments are code sequences and, thus, have a list of authors as well as a main author. The main author of a clone class is defined as the author who is the main author of most fragments in the class. If two or more authors are main author of the same number of fragments, one of them is chosen randomly as the main author of the class. In the remainder of this paper we will refer to the main authors of fragments and classes simply as fragment author or class author.

3 Case Study

We conducted a first case study using our authorship tracking approach. The main goal was to evaluate the feasibility of our tracking approach and to gain first insights into clone authorship. Consequently, we chose a pragmatic setup using one of our own programs as subject system and detecting only exact clones.

Subject System. To start with the authorship analysis we analyzed the source code of our own clone detection tool named *clones* and the libraries it depends on. *Clones* is an active project that has been developed for more then 10 years. We analyzed all 1,516 svn revisions from october 2001 to march 2012. In this period *clones* steadily grew from 0 to 75 KSLOC of Ada code. A total number of 23 authors contributed to the code. That is, the system may contain clones that where created and maintained by different developers.

Clone Detection. We used our incremental clone detector [2] to detect clones in all *svn* revisions of the subject system. Only exact clones having fragments with a minimum length of 50 tokens were detected. During the incremental detection our tool tracks the token streams of all files and the changes applied to them. This allowed us to collect the authorship information in-process during clone detection.

4 Results

Across all revisions we detected 137,391 clone classes. That is, 91 clone classes per revision on average. We first analyzed how many clone classes have fragments with different authors. Among all clone classes we detected in all revisions, 33.7% had fragments with at least two different authors, whereas 66.3% of all clone classes have fragments from only one author. That is, most clones seem to be created by the author of the original code. Nevertheless, a significant amount of clone classes are created and maintained by different authors. The number of clone fragments in a clone class does not seem to be correlated with the number of fragment authors. We found many larger clone classes with up to 11 fragments that had only a single fragment author, but also many small classes with only two fragments which have different authors.

We further analyzed how likely it is that clone classes with single or multiple authors are affected by changes. Although changes to clones are rare, as was previously found in another study on this system [1], the probability that a class with different fragment authors changes is twice as high as the probability that a clone class with only a single fragment author changes.

Besides the probability of change we were also interested in the change propagation across the clone fragments. That is the question whether all fragments of a clone class change consistently (exactly the same change is applied) or inconsistently. Our results show that when clone classes with a single fragment author change, this happens consistently in 50.9% of all cases and inconsistently in 49.1%—this matches previous findings on the evolution of exact clones in this system [1]. This is different for clone classes with multiple fragment authors. These change consistently in only 20.4% of the cases, whereas 79.6% of the changes to these clone classes are inconsistent.

5 Conclusion and Future Work

We presented a technique to track the authors of cloned code fragments and conducted a case study that sheds first light on the effect multiple authors have on clones. Most clone classes were authored by a single user. Nevertheless, one third of the clone classes we found had multiple fragment authors. Clone classes with multiple authors are more likely to change than classes with only a single author. When a clone class with multiple authors changes, the probability of inconsistent changes is higher than for clone classes with single authors.

These results indicate the the authorship of clones does have an effect on how they evolve. Our future research will extend and refine the clone authorship analysis and investigate the effect of multiple authors in more detail. It will be directed to the causes, which will require qualitative analysis of the changes to clones with multiple authors. Furthermore, we will analyze further and larger systems. Problems related to multiple clone authors may be more likely in larger systems with more contributors. Different kinds of clones, such as near-miss clones, may also lead to different results.

References

[1] N. Göde. Evolution of type-1 clones. In *Working Conference on Source Code Analysis and Manipulation*, 2009.

[2] N. Göde and R. Koschke. Incremental clone detection. In *European Conference on Software Maintenance and Reengineering*, 2009.

[3] J. Harder and N. Göde. Quo vadis, clone management? In *International Workshop on Software Clones*, 2010.

Type 2 Clone Detection On ASCET Models

Francesco Gerardi
Reutlingen University
Fakultät Informatik
Reutlingen, Germany
fgerardi@web.de

Jochen Quante
Robert Bosch GmbH
Corporate Research
Stuttgart, Germany
Jochen.Quante@de.bosch.com

Abstract

Clones are a well-known bad smell pattern in software. So far, research has concentrated on detection of clones in textual languages, while model-based development becomes increasingly important. This is in particular true for the automotive domain, where modelling languages like ASCET are used. This paper presents the adaptation and extension of an existing approach for detection of clones in models. The main novelty is a graph analysis that can detect clones of type 2 (i. e., identical structure, but renamed elements) and distinguish between consistently and inconsistently renamed model elements.

1 Introduction

Cloning means the duplication of artifacts within a system. This behavior is considered harmful for software development: Clones are the number one in the stink parade of bad smell patterns [2]. Such redundancies increase the effort for maintenance and consequently the costs for development of software systems.

Clone detection on the level of source code is a well-known and long-term topic in research. Over the years, a lot of approaches have been established for clone detection in textual programming languages [4]. The same problem is also present in higher-level programming, the modelling of software. In embedded software development, an executable application for control units is generated from data-flow oriented models, e. g., ASCET models[1]. Real world models often contain thousands of connected elements. This fact complicates the manual search for clones in such models or even makes the search impossible. However, there have only been very few activities in model clone detection research.

The seminal work in this area is from Deissenboeck et al. [1]. Their approach is publicly available as part of ConQAT[2]. It provides the base functionality for detecting clones in Matlab/Simulink models. We call it CMCD (ConQAT's Model Clone Detection) in this paper. The contribution of this paper is an adaption of this algorithm to ASCET models and its extension for detection of additional clone types.

[1] `http://www.etas.com/en/products/ascet_software_products.php`

[2] ConQAT is an open-source toolkit for various software analysis and quality measurements. `http://www.conqat.org/`

2 Adaption and Extensions

The core of CMCD works on a general graph structure with additional labels for each node and edge. Therefore, the existing ASCET model has to be transformed into a CMCD graph, and the labels have to be determined from the model. The result of CMCD are pairs of subgraphs that are clones of each other. This means that the subgraphs' structures are identical, as well as the labels. We extended the basic CMCD framework by further parameterizable preparation steps (Preprocessing) and detailed graph analyses (Postprocessing) for retrieving better interpretable results.

Preprocessing

Preprocessing includes the correct transformation from the ASCET-specific model elements (nodes) and connectors (directed edges) to the model structure as needed by CMCD as input. Furthermore, a canonical label is built for each model element. It depends on the user-selected clone type that should be detected. Table 1 shows how the use of different attributes as the canonical label allows detection of different clone types. This makes it possible to detect clones with nodes that have different visible labels in the original model, but have other commonalities. For example, a variable with a different name is still a variable.

CMCD's Clone Detection

CMCD starts its clone detection by first identifying clone candidates, which are pairs of nodes with matching labels. Next, each of these candidates is successively enlarged according to the graph structure and

	Attribute	Example
Type 1	Label	`"&&"`
Type 2	Class	`"Operator"`
Type 2x	Class+Category	`"OperatorLogical"`

Table 1: Variants of constructing the node labels.

labels. Only clones of a certain minimum size are reported. Optionally, clones are summarized into clone cluster, a representative form for many identical clone pairs. For details, see Deissenboeck's paper [1].

Postprocessing

The postprocessing step performs further graph analyses on CMCD's clone results. The following analyses and visualizations are provided:

- An overview graph shows which parts of the graph contain clones.
- Optionally, the number of occurrences of each node in all clones can be visualized to identify regions of intensive cloning.
- Each clone pair is checked for inconsistently or consistently renamed labels (for type 2(x))
- One detailed graph per clone pair shows this additional information.
- Uncloned nodes that are directly connected to a cloned node can optionally be included to get more context information.
- Statistics about the size of the found clone pairs, along with information about the number of consistently and inconsistently renamed nodes.

To distinguish between consistently and inconsistently renamed labels, the algorithm in Figure 1 is used. The algorithm assumes that a graph is defined by $G = (V, E, L)$ where V describes the set of nodes, E the set of directed edges and L the labelling function $L : V \cup E \to N$. While the subgraph $G_1 = (V_1, E_1, L_1)$ points to the original artifact, $G_2 = (V_2, E_2, L_2)$ means its corresponding duplicate. The function f maps a node from V_1 to its cloned counterpart.

The algorithm basically checks for each group of identically labelled nodes in G_1 (M) whether the corresponding set of nodes in G_2 also has a unique label (N). If this is the case, then the labels are renamed consistently, otherwise they are not. This check has to be performed in both directions.

Require: $G_1 = (V_1, E_1, L_1), G_2 = (V_2, E_2, L_2)$

1: $Q := \{v \mid v \in V_1 \wedge L_1(v) \neq L_2(f(v))\}$
2: **while** $|Q| > 0$ **do**
3: take one $q \in Q$ arbitrarily
4: $M := \{v \in Q \mid L_1(v) = L_1(q)\}$
5: $N := \{L_2(f(v)) \mid v \in M\}$
6: **if** $|N| > 1$ **then**
7: mark all $v \in M$ as **inconsistent** labels
8: **else**
9: mark all $v \in M$ as consistent labels
10: **end if**
11: $Q := Q \setminus M$
12: **end while**

Figure 1: Algorithm for detecting inconsistencies

3 Evaluation

The approach was implemented and applied to a number of real-world models with up to 1,500 nodes. The calculation was performed in reasonable time (< 35s). It delivered relevant clone pairs of remarkable size (up to 250 nodes). However, there were also a number of false positives among them, in particular for type 2(x) clones.

An indicator for detecting these false positives is the relationship between clone size and the number of inconsistent labels. Small clones with a big share of inconsistencies can mostly be classified as occasional matches. In contrast to that, large clones with a small proportion of inconsistent labels are more interesting and should be considered more intensively. The statistics produced by the postprocessing provides this information. By appropriate sorting of the data sets it is possible to identify relevant clone pairs.

4 Conclusion

So far, there was no possibility to perform tool supported clone detection on ASCET models. The adaptation of an existing approach for clone detection in models makes it now possible for ASCET. The additional postprocessing provides convenient visualizations and interpretation support. The algorithm for detecting inconsistencies is the first designed and implemented for model clones at all. It can as well be applied to clone detection on other kinds of models. The evaluation has shown potential for increasing the accuracy of results by adjusting parameters. Overall, this work is another building block for further improving the quality and efficiency of model-based embedded software development.

References

[1] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proc. of 13th ICSE*, pages 603–612. ACM Press, 2008.

[2] M. Fowler and K. Beck. *Refactoring: Improving the design of existing code.* The Addison-Wesley object technology series. Addison-Wesley, Reading MA, 1999.

[3] F. Gerardi. Erkennung und Visualisierung von Klonen in modellbasierter eingebetteter Software. Bachelor's thesis, Reutlingen University, Germany, 2012.

[4] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

Late propagation of Type-3 Clones

Saman Bazrafshan

Universität Bremen
saman.bazrafshan@informatik.uni-bremen.de

Abstract

Type-3 clones are duplicated source code fragments that span two or more identical sequences of tokens (whitespace and comments are ignored) that form a contiguous source code fragment interrupted by non-identical token sequences. Several studies on the evolution of code clones have been conducted to detect patterns that can help to manage clones [3,6]. One of those patterns that is assumed to be of special interest is late propagation [1,2,4]. In this paper, ways of detecting late propagation in the evolution of type-3 clones are proposed and discussed.

1 Introduction

During the last years, different studies focused on detecting clone patterns that are considered to have a negative impact on code quality and therefore on maintainability of software. Missing or inconsistent propagation of changes to clones is identified as one pattern that may introduce new defects or prevent the removal of existing ones. To find these clone patterns and enable clone management, a series of tools have been introduced—including clone detectors and clone genealogy extractors. Clones reported by a clone detector are generally distinguished according to their level of similarity. Clones that are identical except for comments and whitespaces are called type-1 clones. Type-2 clones extend type-1 clones by tolerating differences in parameters (e.g., variables, identifiers and literals). Type-3, moreover, allow the insertion and deletion of statements. For a general overview of clone research, please refer to [5]. In this paper, we will name type-1 clones as identical clones and type-2 and type-3 clones as near-miss clones.

An important aspect of clone management is that not all detected clones are equally relevant. To filter relevant clones out of the large number of clones reported by a clone detector, it is of advantage to extract and analyze the evolution of clones—also called clone genealogy [3,6]. One evolution pattern that has been studied in recent studies is the late propagation. The late-propagation pattern denotes a change to one or more fragments of a clone class that is not propagated to all fragments of the clone class at the same time. Considering defect-correcting changes, late propagation pattern is an indicator that code clones were not intentionally changed inconsistently [1,2,4].

2 Late Propagation of Near-Miss Clones

The definition of a late propagation regarding identical clones is straightforward: an inconsistent modification of an identical clone causing the fragments to be non-identical until another inconsistent change to the fragments makes them identical again. However, the definition is not suitable for near-miss clones because they are not completely identical–changes between the identical and the non-identical parts have to be differentiated. The challenging question that arises from this fact is:

> *What are the essential characteristics of a change that makes an inconsistent change to a near-miss clone consistent at a later point of time?*

One way to define the late propagation pattern for near-miss clones is to focus exclusively on the identical parts of a clone disregarding the gaps as the gaps are already not common between the cloned fragments. In this case, we would regard a near-miss clone to be changed consistently if the identical parts undergo the same modifications and continue to be identical–analogously to the definition of a late propagation of identical clones. Hence, to recognize an inconsistent change to a near-miss clone that makes a preceding inconsistent change to the same clone consistent at a later time, all deltas of the clone fragments have to be remembered and compared to every new inconsistent change. Considering a clone class with more than two clone fragments, a gap between two fragments might exist at a different position, in a different form (e.g., different in size), or even not present at all compared to the other fragments of the same clone class. For this reason, it has to be taken into account that a change might hit a gap regarding one or more fragments but an identical block with respect to the other fragments of the clone class. In addition, it is possible that an inconsistent change makes more than one preceding inconsistent change to various fragments consistent at once. Thus, different combinations of deltas have to be compared against an inconsistent change for a sufficient analysis of the consistency of all fragments

belonging to the same clone class. These considerations suggest that detecting late propagations in large near-miss clone genealogies with a high number of inconsistent changes is not feasible using an approach based on the comparison of fragment changes because of the costly comparison strategy.

Defining the late propagation pattern for near-miss clones based on a clone class rather than on its fragments helps to overcome the problem of the cost-ineffective comparison of fragment changes. Instead of focusing on consistent or inconsistent changes regarding clone fragments, a clone class is extended by a state attribute that provides information about the clone class being consistent or inconsistent. A consistent state is defined as follows: At the time of its creation, a clone class is in a consistent state. In the course of the evolution of its clone fragments, a clone class is in a consistent state if it contains at least all clone fragments that have been part of the same clone class the last time it was known to be in a consistent state. The only exception to this rule are fragments that were deleted by the deletion of a whole file. This exception is made because the deletion of source code by a file deletion is not considered to be related to an intended removal of a clone fragment. To make sure we do not handle a moved or renamed file as a deleted one we use the log information of software repository tools that enable file mappings between moved and renamed files between two versions. This way, the deleted clone fragment does not prevent its former clone class to get consistent again. In contrast to a removal of a clone fragment by a file deletion, a deletion of a fragment itself is considered to be a conscious decision to remove the clone fragment. Hence, a clone class that once contained such a clone fragment can not get in a consistent state again. Note, that the information about deleted clone fragments is based on the applied clone detector. Figure 1 shows the evolution of two clone classes throughout three versions of a software system. The left clone class containing the clone fragments a, b, c is in a consistent state in version V_i. In version V_{i+1}, the clone fragment c is not part of the clone class anymore—c was not deleted by a file deletion. Because of the missing fragment c that was part of the clone class the last time it was in a consistent state, namely, in version V_i, the clone class is in an inconsistent state in version V_{i+1}. After the fragment c returns to be part of the clone class in version V_{i+2} it is back in a consistent state. The switch from an inconsistent state back to a consistent one indicates a late propagation considering our definition of a late propagation for near-miss clone genealogies. The right clone class illustrates the above mentioned exception regarding a clone fragment being deleted by a file deletion. Although the clone fragment f, that was part of the clone class in version V_i, is not part of it in the following two versions, the clone class remains in a consistet state. In addition, it can been

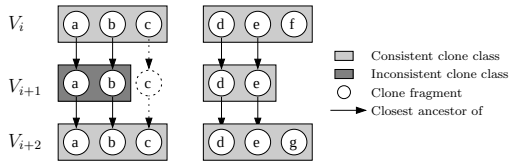


Figure 1: Evolution of two clone classes throughout three versions of a software.

seen that a newly introduced fragment g in version V_{i+2} does not change the state of the clone class from consistent to inconsistent because it is only required that at least all fragments are contained by the clone class that have been contained in the same clone class the last time it was in a consistent state.

One might argue that regarding the changes to clone fragments as black boxes instead of doing a more precise analysis of the changes might not be sufficient enough to be considered an indication for a possible late propagation.

3 Conclusion

Currently there is no study in clone research–to the best of our knowledge–that concentrates on adapting the late propagation pattern of identical clones to near-miss clones. In this paper, we propose two different approaches that might be adequate to detect late propagations of near-miss clones and discuss their possible shortcomings. This work is in progress and right now we are working on the implementation of the approaches. Afterwards an evaluation of both approaches using different software systems is planned. Besides a quantitative study of each approach, we intend to compare the results to existing studies of the late propagation pattern in genealogies of identical clones. By comparing our results to existing ones that are generally assumed to be valid, we expect to be able to draw some conclusions regarding differences of late propagations in idetical and near-miss clone genealogies.

References

[1] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *Proc. of 11th European CSMR*, pages 81–90. IEEEPress, 2007.

[2] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *Proc. of the 27th ICSM*, pages 273–282. IEEEPress, 2011.

[3] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proc. of the 10th ESEC/FSE*, pages 187–196. ACM, 2005.

[4] H. H. Mui. Studying late propagations using software repository mining. Masters thesis, Delft University of Technology, 2010.

[5] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queens University at Kingston, Ontario, Canada, 2007.

[6] R. K. Saha, C. K. Roy, and K. A. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *Proc. of the 27th ICSM*, pages 293–302. ACM, 2011.

Aspects of Code Pattern Removal

Mikhail Prokharau

University of Stuttgart
Universitaetsstr. 38, 70569 Stuttgart, Germany
mikhail.prokharau@informatik.uni-stuttgart.de

Abstract

While many approaches for detecting undesirable or recurring code patterns have been presented over the years, there often remains the question how to properly interpret their results. This article presents a number of aspects of semi-automatic pattern elimination and discusses consequences of their application for safety and efficiency.

1 Introduction

The idea behind undesirable pattern detection is obviously to remove the detected patterns and, thus, to improve the overall quality of the analysed software. Often the removal itself has to be performed manually, in which case the user has to decide whether it is at all reasonable to eliminate a particular instance, how to do it best and, last but not least, how to perform this operation safely guaranteeing that the semantics of the original code remains intact. Another important criterion which is often overlooked, however, is how the change is going to affect code generation and program analyses performed on the software later on.

Many detection approaches divide the recurring pattern space into four types [2] of patterns named clones. The types are classified according to code similarity. The higher the pattern type, the higher the degree of modification that is required to perform the removal. Yet, as more changes are introduced, the higher the likelihood of incorrect or in some way detrimental modification becomes. To alleviate the problem, parts of the elimination process can be automated. To do so, the results of program analyses can be used that provide a number of important details fundamental to the decision whether the actual elimination should be performed and how to do it best. An advanced program analysis tool featuring suitable intermediate representation that preserves close links to the original code is most suitable for this purpose. The Bauhaus project [1] is a good example.

2 Semi-automatic Pattern Removal

While pattern detection is effectively and efficiently possible using higher-level code abstractions such as abstract syntax trees, the removal can profit significantly from the results of program analysis techniques applied to lower-level code abstractions. Specifically, a number of candidates definitely known to be false positive can be automatically excluded from having to be considered by the user. The exclusion is often performed by pattern detection tools, yet the results might be rather limited depending on the availability and precision of control and data-flow analysis data.

Some of the common code removal techniques include identifying subprograms which perform the same action and unifying them into a single subprogram. In some other cases it might be reasonable to unify certain cloned regions of code into a single subprogram and call it separately from within each context, possibly with varying parameters and case-statements to differentiate among the contexts. A number of non-local variables used in the fragment might become parameters of the new subprogram. Depending on their visibility some of the non-local variables might be accessed directly. Program analysis techniques can provide the user with detailed information regarding the specific choices.

If more than two instances of the same code passage have been detected and some yet not all of those have additional common properties as inferred by comprehensive program analyses, further subgrouping is a reasonable way of their presentation to the user. This is the case, for example, if subprogram pointers were identified to point to one specific subprogram in a subset of detected recurring patterns as a result of the must-point-to analysis. In fact, if this is the only difference, automatic substitution of the subprogram pointer with a direct call of the identified subprogram results in promoting the relevant code patterns from Type-2 to Type-1 clones.

3 To Remove or not to Remove?

Advanced programming constructs such as references, function pointers or dispatching calls are found in many contexts. These contexts probably present significant difficulty for the user who has to decide whether patterns using such constructs lead to the same execution in all contexts. In those situations the results of the previously conducted in-depth static program analyses are of particular value. For instance, if the program analyses had difficulty reducing the points-to-sets to a specific object, providing the

detailed analysis information to the user may facilitate her or his decision-making.

Unification of certain contexts where previously mentioned advanced programming constructs are present may also significantly affect subsequent code generation and analyses. For instance, in cases where the reduction of the number of objects being possibly pointed to by a pointer was previously attainable, it might no longer be so. This may have a number of potentially very negative consequences, such as lower efficiency of the resulting code due to the limited number of possible optimisations and the larger number of false positives delivered by the analyses depending on the pointer target sets. Where such effects are present, the user should be informed of the consequences of code pattern removal. The feedback may range from a generic warning to specific details regarding the expected complications.

4 Patterns in Specialised Contexts

Particular care should be taken when removing code patterns in specialised contexts.

Where parallel execution might be present, certain variables can be used for special tasks such as locks or synchronisation variables guaranteeing separate execution. An improper modification can result in serious and difficult-to-detect errors such as race conditions.

Introducing additional subprogram calls instead of cloned code may negatively affect the code efficiency. Unless inlining is used, a subprogram call normally requires a context switch taking longer execution time. While not likely to cause problems in linear parts of the code where only a small number of calls occur, this is of particular importance if loops, especially with a high degree of nesting, are involved.

In the context of real-time systems, where deadlines must be adhered to, pattern removal techniques that affect the resulting code efficiency quickly become unusable unless the exact quantifiable timing effect can be provided.

5 Practical Consequences

In the example shown in Figure 1 a code fragment is given where functions `f1` and `f2` might be recognised to be Type-3 clones by a clone detection analysis. In Figure 2 a possible result of pattern removal is shown. As can be observed, a function pointer was introduced to represent the code parts which are different. A similar strategy can be used in object-oriented languages where dispatching calls would substitute function pointers. In all of those cases the resulting context fusion poses a significant problem for points-to program analyses. Since pattern removal is performed in the synchronised context, the parallelisation analyses are also affected. The result might be a significant increase in the number of detected points-to targets and, consequently, false positive race condition notifications.

```
1  void f1() {
2      pthread_mutex_lock (&mutex1);
3      global++;
4      global *= 2;
5      pthread_mutex_unlock (&mutex1);
6  }
7
8  void f2() {
9      pthread_mutex_lock (&mutex2);
10     global++;
11     global *= 3;
12     pthread_mutex_unlock (&mutex2);
13 }
14
15 void c() {
16     pthread_mutex_lock (&mutexc);
17     f1();
18     f2();
19     pthread_mutex_unlock (&mutexc);
20 }
```

Figure 1: The original code fragment

```
1  void work1() {
2      global *= 2;
3  }
4
5  void work2() {
6      global *= 3;
7  }
8
9  void f(pthread_mutex_t* lock, void (*work)(void)) {
10     pthread_mutex_lock (lock);
11     global++;
12     work();
13     pthread_mutex_unlock (lock);
14 }
15
16 void c() {
17     pthread_mutex_lock (&mutexc);
18     p = work1;
19     f(&mutex1, p);
20     p = work2;
21     f(&mutex2, p);
22     pthread_mutex_unlock (&mutexc);
23 }
```

Figure 2: The code fragment after pattern removal

6 Conclusion

While removal of undesirable or recurring code patterns may prove very useful in improving a number of important software metrics, care should be taken not to introduce significant performance overheads while maintaining a high degree of the software analysibility. Semi-automation based on in-depth static program analyses can be quite valuable for reducing the otherwise unforseen negative effects.

References

[1] A. Raza, G. Vogel, and E. Plödereder. Bauhaus - a tool suite for program analysis and reverse engineering. In *Ada-Europe 2006*, volume 4006 of *LNCS*, pages 71–82. Springer, 2006.

[2] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

A Model-Based Approach to Type-3 Clone Elimination

Torsten Görg

University of Stuttgart
Universitaetsstr. 38, 70569 Stuttgart, Germany

torsten.goerg@informatik.uni-stuttgart.de

Abstract

This paper presents an approach to elimination of Type-3 code clones based on generative techniques. At a first step it is shown how to replace Type-3 clones by higher-order functions. Then this approach is further generalized by utilizing generative designs.

1 Introduction

Code clones are a well known phenomenon that occurs in program source code. As defined in [3] usually four types of code clones are differentiated. This article focuses on Type-3 clones that do not crosscut syntactical boundaries. In contrast to Type-2 clones, Type-3 clones are characterized as copied code fragments with any modifications. Many clone detection mechanisms have been developed. The question that comes up after clones in a system are detected is what to do with them. Many authors believe that code clones impact maintainability negatively and it is a good idea to get rid of that redundancy. This can be achieved by code transformations. The following sections shows several techniques for clone elimination. It is presumed that one set of code fragments that are all similar to each other is provided as input.

2 Elimination of Type-2 Clones

Generally code clones can be eliminated by introducing a new abstraction or extending an existing abstraction. Several publications show how to implemented that for Type-2 clones, e.g., [2].

A further classification dimension for clones is their granularity related to the syntactical level they appear on. Syntactical levels can also be used as a criteron to classify clone elimination techniques. For clones at different snytactical levels different elimination techniques are applied. These are some syntactical levels: expression level, statement level/statement sequence level, subprogram (procedure/function) level, and type level/class level.

Many elimination techniques have been developed for the statement sequence level and the subprogram level. Common statement sequences can be extracted to a new procedure. Cloned subprograms can be merged. As clones live in different contexts, access to variables and subprograms that are referenced by the clone code has to be established through parameters that are added to the signature of the extracted or merged subprogram.

3 Elimination of Type-3 Clones

To eliminate Type-3 clones the techniques mentioned in the previous section can be extended. The subprogram extraction for the statement or subprogram level has to consider that the abstracted statement sequences are different. The extracted subprogram must behave differently depending on the calling context. This can be handled with additional subprogram parameters. The easiest way is to introduce an ID to discriminate different contexts. The execution of context-specific code is controlled by case switches. This approach has the disadvantage that context specifics are introduced into the extracted subprogram. It violates the separation of concerns principle.

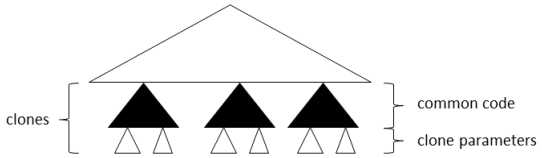


Figure 1: AST of code containing Type-3 clones

This problem can be solved with higher-order functions. The idea is to pass the code parts that are different in the clones as parameters. This can be viewed as a horizontal cut in the AST that represents the clone code (see Figure 1). The upper region above the cut is common for all clones. Code differences are expressed by subtrees that are below the cut. These subtrees can be written as lambda expressions or as separate functions and passed to the extracted subprogram via a function parameter. Of cource functional programming languages are predestinated for that technique, but in many imperative languages this is possible as well. Here is an C# example:

```
void extracted( Action<int, int> specific ) {
   int a = 5; int b = 3;
   Console.Write( "Result: " );
   specific( a, b );
}
...
extracted( (x, y) => { Console.Write(x); } );
extracted( (x, y) => { Console.Write(y); } );
```

In the example the parameter function `specific` is called with `a` and `b` as parameter values. This provides context information but, in contrast to the extracted subprogram that needs the outside context, here the inside context is required for the parametrization that is done outside.

4 Model-Based Techniques

Even more powerful are generative techniques. A generator produces some output based on templates. Obviously all instantialitions of the same template are clones. The generation is parameterized with model data. The resulting clones are Type-3 clones. E.g., UML tools generate source code skeletons from class models, the code for all classes looks similar as it is based on the same template. Here is another example with GUI code that creates some buttons:

```
w.add( new Button( "Yes", handlerYes ) );
w.add( new Button( "No", handlerNo ) );
w.add( new Button( "Cancel", handlerCancel ) );
```

If we reverse the generation process, it can be used for clone elimination. Given cloned code fragments are reduced to a domain-specific model. The model is free of redundancy and simplifies modifications during maintenance. But a model is not directly executable. For compilation, source code is required. In the build process source code corresponding to the model data is generated as described above. An appropriate generator has to be available. In fact, reversing the generation process does not really eliminate clones. The clones are just hidden. They are implicitly created by the generator. But the effect is similar as with clone elimination, as no manual modifications have to be done on the generated code. Just the model and the generator are modified manually.

To eliminate Type-3 clones this way, three artifacts have to be derived from the clones: a domain-specific model, a metamodel, and a corresponding generator.

(1) The model contains all differences between the clones. Similar to the parameters of an extracted subprogram, the model provides the contexts. For the GUI example above these are the extracted differences:

```
"Yes" handlerYes
"No" handlerNo
"Cancel" handlerCancel
```

(2) The data contained in the model has a structure that is specified by a metamodel. In Figure 2 it is shown for the example. The metamodel also has to be extracted from the clone code. The metamodel is important for the generator to know which template to apply to the model data.

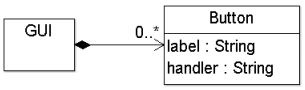


Figure 2: Metamodel for the GUI example

(3) The generator mainly consists of template definitions with placeholders for referenced parameters. Here is the Button template for the GUI example:

```
w.add( new Button( <<label>>, <<handler>> ) );
```

A template is created based on the clone code by replacing varying parts by placeholders for clone parameters.

So far the example does not go far beyond handling Type-2 clones. For a full support for Type-3 clones, the derived metamodel is combined with the metamodel of the underlying programming language. This is similar to the extension of extracted subprograms to higher-order functions, as shown in the previous section. Where delegates are used to express complex variations in higher-order functions here arbitraty ASTs can be embedded in the domain-specific model. Figure 3 provides a combined metamodel for

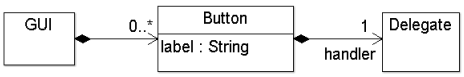


Figure 3: Combined metamodel for the GUI example

the GUI example. Instead of referencing handler functions by name, the extended metamodel enables to model handlers as arbitrary delegates.

5 Related Work

A realization of the higher-order function approach for Haskell code is described in [1].

6 Conclusion

Higher-order functions can be used to eliminate Type-3 clones at the statement sequence level. Generative techniques provide full flexibility to eliminate Type-3 clones that are arbitrarily parameterized at any syntactical level.

References

[1] C. Brown and S. Thompson. Clone Detection and Elimination for Haskell. In *PEPM '10 Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*. ACM, 2010.

[2] R. Fanta and V. Rajlich. Removing clones from the code. *Journal of Software Maintenance*, pages 223–243, 1999.

[3] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 2009.

Leveraging Design Decisions in Evolving Systems

Martin Küster, Benjamin Klatt
FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
{kuester,klatt}@fzi.de

1 Introduction

When a system needs to evolve, it is necessary to not only find code that needs to change, but also to understand why it has been developed the way it was (e.g. which requirements have influenced the software design). Due to cost constraints, missing knowledge, or the initial intention to build a prototype only, it is common practice not to trace requirements to design decisions from the beginning. Existing requirements engineering and documentation approaches provide too generic linking capabilities. Even mature trace frameworks (e.g. EMFTrace[1]) do not provide sufficient support for tracing design decisions and their origins (e.g. requirements). Our approach aims to provide i) models and traces capturing software design decisions, ii) a light-weight process for incrementally building trace information having a minimal impact on design and development efforts, and iii) a view-concept to support developers during future system evolution. To illustrate the application of this concept, we provide an example in the context of migrating several product copies to software product lines and their future evolution.

2 Documentation of Design Decisions

Design decisions are major entities in the software development process. Even if software developers often make them implicitly, it is critical to find out i) why a specific design alternative has been chosen, and ii) how a requirement manifests itself in the software. To answer those questions, it is important to prevent software design erosion when the software evolves, and to efficiently perform the necessary changes while knowing the entire context.

Derived from these goals, a design decision links requirements, software entities (e.g. components, interfaces, and classes), and related design decisions (e.g. those made earlier) as shown in Figure 1. Nowadays, mature models and ontologies exist for requirements [6] as well as software entities [5], however the link between these models has not been sufficiently addressed. The trace model that we developed captures design decisions, the references between them, and their related requirements and software entities.

[1] http://sourceforge.net/projects/emftrace/

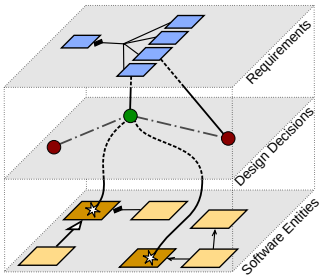


Figure 1: Design decisions linking requirements and software entities

3 Design Decision Views for Evolution Support

When a software engineer has to implement a new requirement in an existing system, she might already have an idea which part of the system might be affected. Several existing approaches enable software engineers to locate features that need to be modified in the system. However, the engineer still needs to decide how to modify the system without unintentionally breaking the architecture of the software.

To support her in making a more informed decision, we propose views presenting previously made design decisions related to the software entities she is working on. This includes the latest design decisions (history) as well as links to their (causal) predecessors. In addition to the design decisions, the requirements that caused the design decision can also be made visible.

The challenge of presenting information from different models in one view (requirements, decisions, architecture, design, code) is tackled using a graph-based visualisation. All elements that are related to a focused element can be displayed around it. The potentially large dependency graph can be reduced according to the engineer's current task and the desired kind of relationship (timeline, causal chain, etc.). When navigating through the net of relationships, the graph is auto-layouted using standard layouting algorithms.

4 Process to Capture Decisions

The capturing of design decisions faces two major challenges. First, software engineers make implicit

design decisions and are not used to capturing them. Second, there might be resentements against the additional effort. We assume the latter can be offset by the benefits of making the decisions explicit, which in turn saves efforts.

To better support the capturing of design decisions we propose a guided process which makes this intuitve and unconscious. When an engineer makes a design decision to implement a requirement a supporting tool should be aware of her current context (code, architecture) and offer her capabilities to i) describe the decision itself, and ii) link it to the requirements as well as the affected software entities. We propose this as a generic concept which could be further enhanced with domain-specific adaptations. For example, taking domain-specific models into account can be used to present more specific information and to better assess the engineer's current context. Furthermore, integrating the tool into the engineer's individual development environment might enable a better interpretation of her current context and an even more domain-specific presentation and access of the elements. To illustrate a domain-specific application, we give an example from our work on migrating software product copies into common product lines.

5 Product Copy Migration Example

Due to organisational and technical constraints, such as costs, time, and evolving domains, software products are often copied and customised without taking concepts such as product lines into account. In [3] we explain how to identify and recommend related variation points in the software. The product line engineer has to make design decisions on i) which variation points to merge, and ii) how to realise their variability. The possible realisation alternatives are influenced by the individual requirements of the product line (e.g. compile-time vs. run-time configurable systems). Those decisions are later used to refactor the product copies to the software product line.

During this process, the necessary information to capture the design decisions is nearly complete: The reason for introducing the variation points is the consolidation of the product copies. The variation point and the corresponding variants are linked to the software entities and the design decision. The rationale for the design decision is represented by the individual requirements of the resulting product line. With a domain-specific tool the engineer can simply accept to capture them in the decision model.

Suppose the product line evolves and a variation point should now be bound during run-time instead of compile-time to enable a multi-tenant system. To realise this requirement, the product line engineer needs to answer the following questions: What architectural elements are touched by the decision? How is the variation point manifested in the design? What was the initial requirement to have compile-time binding? These questions, leaving the scope of a typical view on the system, can be displayed in a view as described in Sec. 3. Exploring the context of the evolution task before actually implementing it will increase effectiveness and efficiency.

6 Related Work, Outlook

Bode et. al [1] proposed a quality-focused method for design traceability based on software categories. Various approaches based on semantic wikis have been proposed. They are mostly focused on forward-engineering coming from requirements. Little support for evolution is provided. Preliminary work on the semantics of traces between requirements and architecture has been proposed by Goknil et al. [2]. Our work is similar to Könemann's [4] with respect to explicitly considering design decisions as first-class entities, but Könemann gives no hint on how to exploit this information.

We plan to show the usefulness of the provided views in an empirical study: Two groups of developers will conduct the same evolution task. One group is provided with the respective views gained from the recorded design decisions, and the other is not. To facilitate this, an integration into Eclipse is currently being developed.

In this paper, we outlined how views can support software evolution using documented design decisions. We described a lightweight process to record such design decisions. Giving an example from the area of product-line engineering we showed how the approach can be adapted for a specific domain and the gained knowledge can ease later evolution steps.

References

[1] S. Bode and M. Riebisch. Tracing Quality-Related Design Decisions in a Category-Driven Software Architecture. In *Software Engineering*, pages 87–97, 2009.

[2] A. Goknil, I. Kurtev, and K. van den Berg. Tool support for generation and validation of traces between requirements and architecture. *Proceedings of the 6th ECMFA Traceability Workshop on - ECMFA-TW '10*, pages 39–46, 2010.

[3] B. Klatt and K. Krogmann. Model-Driven Product Consolidation into Software Product Lines. In *Workshop on Model-Driven and Model-Based Software Modernzation (MMSM'2012)*, Bamberg, 2012.

[4] P. Könemann and O. Zimmermann. Linking design decisions to design models in model-based software development. *Software Architecture*, pages 246–262, 2010.

[5] OMG. Architecture-Driven Modernization : Knowledge Discovery Meta-Model (KDM). Technical Report August, OMG, 2011.

[6] A. Tang, P. Liang, V. Clerc, and H. V. Vliet. Traceability in the Co-evolution of Architectural Requirements and Design. In *Relating Software Requirements and Architectures*. Springer Berlin Heidelberg, 2011.

Sichern der Zukunftsfähigkeit bei der Migration von Legacy-Systemen durch modellgetriebene Softwareentwicklung

Marvin Grieger, Baris Güldali, Stefan Sauer
Universität Paderborn, s-lab – Software Quality Lab, Zukunftsmeile 1, 33102 Paderborn
{mgrieger, bguldali, sauer}@s-lab.upb.de

1 Modernisierung von Legacy-Systemen

Softwareunternehmen stehen zunehmend vor der Herausforderung, große betriebliche Informationssysteme, die mit Softwaretechnologien wie den 4. Generationssprachen (4GL) und -werkzeugen aus den 90er Jahren entwickelt wurden, durch moderne Systeme mit mehrschichtigen Architekturen abzulösen, die auf fortgeschrittenen Softwaretechnologien aufsetzen. Für sie gibt es leistungsfähige Entwicklungsplattformen, und die neuen Technologien und Architekturen erlauben es, die Systeme einfacher zu warten und sie leichter an sich ändernde Anforderungen anzupassen. Das erhöht die Zukunftsfähigkeit der neuen Systeme. Gleichzeitig sind die Softwareunternehmen bestrebt, die mit viel Aufwand entwickelte Funktionalität der Legacy-Systeme wieder zu verwenden.

Benötigt werden deshalb Migrationsverfahren für Legacy-Systeme, die das Wissen der Entwicklerteams wiederverwendbar festhalten und die Zukunftsfähigkeit der neuen Systeme über die Migration hinaus sicherstellen. Dabei setzen wir auf Techniken der modellgetriebenen Softwareentwicklung (engl. Model-driven Software Development, MDSD). Modelle werden als formale Beschreibungsmittel verwendet, und aus den Modellen werden automatisiert Software-Artefakte (weitere Modelle oder Programmcode) generiert. Dadurch dokumentieren wir erstens das Wissen über die Systeme mittels der Modelle explizit und stellen es in dieser Form für die weiteren Migrationsaktivitäten zur Verfügung. Zweitens liefern die Modelle die notwendige Abstraktion um die Auswahl der Zieltechnologie flexibel zu halten.

Die Idee der modellgetriebenen Migration wurde bereits von anderen Autoren adressiert [1-3]. Viele dieser Arbeiten beschäftigen sich mit der kurzfristigen strukturellen und architektonischen Migration der Legacy-Systeme und vernachlässigen dabei das während der Migration entstehende Wissen zu erhalten. Mit unserer Forschung möchten wir Verfahren entwickeln, dass dieses Migrationswissen bewahrt und langfristig nutzbar macht.

In einer früheren Arbeit [4] haben wir die Abgrenzung von anderen Ansätzen beschrieben. Hier erläutern wir unseren Migrationsprozess und die behandelten Abstraktionsebenen, um das Migrationswissen zu konservieren.

2 Verwendung von MDSD-Methoden

Ein zentrales Prinzip der MDSD ist die Verwendung formaler Modelle zur automatischen Generierung von lauffähigen Softwaresystemen. Formale Modelle beschreiben dabei einen Aspekt des Softwaresystems vollständig. Wir beabsichtigen, diese formalen Modelle soweit wie möglich automatisch aus dem vorhandenen Legacy-System abzuleiten und sie für die Migration in das neue System zu nutzen. Die Modelle sollen technologieunabhängig sein, wodurch die Wiederverwendbarkeit erhöht wird. Wir erläutern im Folgenden, wie die verschiedenen Modellarten in den Migrationsaktivitäten erstellt bzw. verwendet werden.

2.1 Abstraktionsebenen

Modelle stellen immer eine Abstraktion dar und können somit einer Abstraktionsebene zugeordnet werden. Dies macht die Komplexität, die einem Softwaresystem zu Grunde liegt, beherrschbar. Gleichzeitig werden verschiedene Sichten auf das System ermöglicht.

Die Abstraktionsebenen sind in der Abb. 1 dargestellt, entsprechend dem MDA-Ansatz der OMG. Auf der untersten Ebene befindet sich die Systemebene, welche die Implementierung einer Applikation bzgl. einer Plattform sowie die zugehörige Laufzeitumgebung beinhaltet. Ausgehend davon leiten wir unter Verwendung von Referenzmodellen plattformspezifische Modelle (Platform Specific Model, PSM) ab. Sie ermöglichen eine Dekomposition des Systems in Komponenten, aus denen wir wiederum plattformunabhängige Modelle (Platform Independent Model, PIM) ableiten. Diese Modelle beschreiben die einzelnen Komponenten formal und technologieunabhängig.

Neben den formalen Modellen werden zusätzlich fachliche Modelle benötigt, um die Refaktorierung bestehender Nutzungsprozesse zu ermöglichen. Diese können wir jedoch nicht aus den Softwaresystemen ableiten, da sie zwar die Fachlichkeit umsetzen, aber in den betrachteten Systemen die Nutzungsprozesse nicht explizit zu erkennen sind. Aus diesem Grund erstellen wir sie manuell, in Form von fachlichen Modellen bzw. Geschäftsmodellen (Computation Independent Model, CIM).

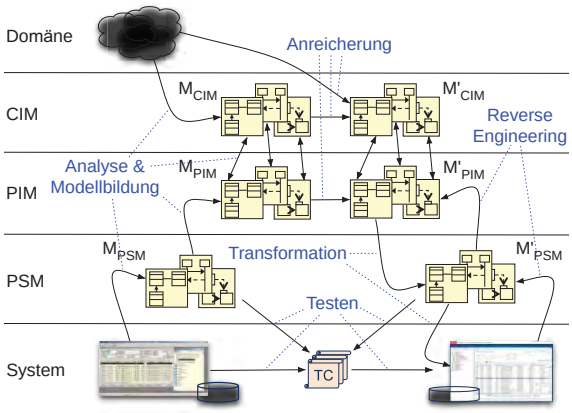


Abb. 1. Migrationsebenen und -aktivitäten

2.2 Migrationsaktivitäten

Unser Ansatz enthält die folgenden Aktivitäten, um die Abstraktionsebenen des Legacy-Systems zu erfassen und sie schrittweise in das Zielsystem zu überführen: Analyse & Modellbildung, Anreicherung, Transformation, Roundtrip-Engineering und Test.

2.2.1 Analyse und Modellbildung

In der Analysephase werden das Legacy-System sowie die zu Grunde liegende Fachlichkeit in Form von Modellen erfasst. Die Modellierung der Fachlichkeit erfolgt manuell, während wir die formalen Modelle des Legacy-Systems automatisch ableiten. Dazu greifen wir auf statische und dynamische Analysemethoden zurück [1]. Statische Methoden ermöglichen uns, Strukturen und Abhängigkeiten im Programmcode und den Architekturebenen zu erkennen. Dabei setzen wir Code-Parser zur Erstellung von Syntaxbäumen und Data-Mining-Werkzeuge zur Klassifikation von Programmcode-Teilen ein. Dynamische Methoden dienen der Erfassung von Abläufen. Ein Ansatz ist die Verwendung der Aspektorientierten Programmierung zur Erstellung von auswertbaren Logdateien.

Der Schwerpunkt in der Analysephase ist der Aufbau einer Werkzeugkette für eine automatische Extraktion der Modelle. Sofern vorhanden sollen dazu existierende Werkzeuge verwendet werden. Beispielsweise existieren Lösungen um strukturelle Informationen automatisch plattformunabhängig zu extrahieren [5]. Die allgemeine Extraktion von Informationen über Zusammenhänge oder Abläufe wird insbesondere dadurch erschwert, dass die Plattform durch die zugehörige Laufzeitumgebung Funktionalität bereitstellt, deren Semantik nicht formal beschrieben ist. Dieses Problem möchten wir lösen, indem wir Analyseverfahren definieren, die Informationen über die Plattform sowie das Entwicklungsprofil, das der Implementierung zu Grunde liegt, einbeziehen. Dieses Profil umfasst u.a. Programmierrichtlinien oder anderweitige Konventionen.

Alle extrahierten Informationen werden in einem zentralen Repository gespeichert und miteinander verknüpft. Dieser Ansatz resultiert in angereicherten formalen Modellen und sichert somit eine hohe Wiederverwendbarkeit.

Eine weitere Herausforderung besteht in der Verknüpfung der extrahierten formalen und manuell erstellten fachlichen Modelle. Erst dadurch wird eine fachliche Anreicherung möglich. Wir beabsichtigen den Einsatz eines Werkzeuges, das eine Abbildung der fachlichen auf die formalen Modelle erzeugt, sobald der fachliche Prozess durchlaufen wird. Solch ein Werkzeug wurde z.B. im Projekt SOAMIG entwickelt [3].

2.2.2 Weitere Aktivitäten

Um die Ergebnismodelle der Analyse in das Zielsystem zu überführen, verwenden wir Modelltransformation und Code-Generierung. Dabei wird das M'_{PIM}, das durch die (interaktive) Anreicherung des M_{PIM} entsteht, zu M'_{PSM} transformiert, das die technologischen Eigenschaften der Zieltechnologie enthält. Aus dem M'_{PSM} werden mittels Code-Generatoren syntaktisch vollständige Codeteile erstellt. Der generierte Code wird manuell vervollständigt. Um die Konsistenz zwischen Code und Modellen zu erhalten, verwenden wir Roundtrip-Engineering-Ansätze, die die Änderungen im Code durch statische Analyse und Mustererkennung finden und die Modelle synchronisieren (vgl. www.fujaba.de).

Um die funktionale Korrektheit nach der Migration zu prüfen, testen wir das neue System mittels Testfällen (*TC*), die auf Basis des Legacy-Systems erstellt wurden. Aus den plattformspezifischen Modellen erstellen wir Testfälle durch modellbasierte Testverfahren. Die Laufzeit-Informationen in den Logdateien dienen als Quelle für konkrete Testdaten. Damit agiert das Legacy-System als Testorakel im Testprozess.

Literatur

[1] A. van Hoorn et al.: DynaMod project: Dynamic analysis for model-driven software modernization. In Proc. MDSM 2011, vol. 708 of CEUR Workshop Proc., pp. 12-13, 2011

[2] S. Efftinge et al.: Einsatz domänenspezifischer Sprachen zur Migration von Datenbankanwendungen. In Proc. BTW 2011, vol. 180 of LNI, GI 2011

[3] Andreas Fuhr, Tassilo Horn, Volker Riediger: Dynamic Analysis for Model Integration, In Softwaretechnik Trends, Band 30, Heft 2, Mai 2010

[4] B. Güldali, S. Sauer, P. Löhr: Entwicklung eines Softwarewerkzeugs für die modellgetriebene Migration betrieblicher Informationssysteme. In Proc. MMSM 2012 (to be published in Softwaretechnik-Trends)

[5] O. S. Ramón et al.: Model-Driven Reverse Engineering of Legacy Graphical User Interfaces. In Proc. ASE 2010, pp. 147-150

Tackling the Challenges of Evolution in Multiperspective Software Design and Implementation

Steffen Lehnert

Ilmenau University of Technology
Ilmenau, Germany
steffen.lehnert@tu-ilmenau.de

Matthias Riebisch

University of Hamburg
Hamburg, Germany
riebisch@informatik.uni-hamburg.de

Abstract

The design and implementation of software require the usage of different perspectives and views to cope with its static structure, dynamic behavior, and requirements. Artifacts of different views are dependent on each other and subject of frequent changes. Anticipating those changes becomes difficult, as most impact analysis approaches are not designed to work in multiperspective environments. They treat artifacts of different perspectives in isolation, which tends to introduce further inconsistencies and faults. In the related research area of consistency checking, the problem of multiple views has been addressed for a long time and solutions have been developed. We aim on combining the predictive capabilities of impact analysis with the approach of multiperspective consistency checking, to bridge the gap between artifacts of different views. We propose an approach which facilitates impact analysis of different UML models and Java source code. Our approach is based on impact propagation rules, which analyze traceability relations between software artifacts and the type of change applied by a developer.

1 Introduction

Studying the "ripple effects" of changes [6] is in the focus of research in software evolution for many years. Various techniques have been proposed to predict and assess the impacts of changes, e.g. slicing [3], history mining [7] or probabilistic analysis [2]. The majority of those approaches is limited to a certain type of software artifacts only, e.g. source code or static UML models. However, software artifacts do not evolve in isolation and ripple effects spread over the boundaries of the different views of software. Insufficient impact analysis increases the drift between high-level design and implementation, as well as gaps between different views in general. On the other hand, a considerable effort has been invested in studying consistency checking in the light of multiperspective software design, e.g. [5]. Although able to analyze multiple different artifacts, current consistency checking approaches are not able to predict the impacts of future changes, nor do they facilitate cost or time estimations. Thus, merging impact analysis with the concept of multiperspective consistency checking yields several improvements, aiding change prediction and consistency maintenance likewise. Expanding impact analysis beyond isolated views increases its ability to forecast cost estimations, and it helps developers to better understand which artifacts require rework according to a change. Improved impact analysis on the other hand results in fewer inconsistencies, if artifacts of different views are addressed adequately, thus reducing the effort required for consistency checking.

2 Multiperspective Impact Analysis

Impact analysis faces two major challenges when applied in multiperspective environments. First, dependent artifacts of different views must be joined and treated in a unified manner to enable multiperspective analysis. Secondly, it requires an impact propagation technique which is capable of analyzing multiple different artifacts, emerging from the different perspectives. Addressing the first challenge can be accomplished by mapping all artifacts on a unifying framework and using an integrating repository for managing all artifacts, which is further explained in Section 3. Using this common basis allows us to tackle the second challenge. The ripple effects of changes can be computed by impact propagation rules, which are able to address arbitrary artifacts. Our impact propagation rules analyze dependency relations which exist between artifacts to determine if and how a change propagates to related artifacts, according to the type of relation and type of applied change. Likewise, consistency management can be accomplished by applying consistency checking rules [5] on the models. Figure 1 illustrates the steps required for impact analysis, consistency management, and steps required for both activities (dashed lines). To enable this kind of analysis, we first need to elicitate the dependencies which exist between the various models and record them in a structured way. We achieve this by applying traceability mining rules which transform dependency relations into traceability links. Our traceability mining rules analyze the structure, the names, and relations

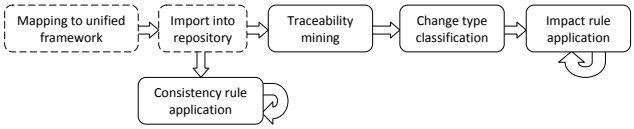


Figure 1: Overview of our approach.

between model elements, and are able to determine the type of relation which exists between them [1]. The second part of our propagation approach requires a structured treatment of change types. In previous work [4] we defined a set of change types on which real reengineering activities can be mapped upon. Using this set of changes, we defined a catalog of more than 140 change operations of different granularity, e.g. changing the return type of a method or merging UML components. Having both, traceability links and change types, we define a set of impact propagation rules which are used to compute the propagation of changes. Our rules are then applied in a recursive manner, until all impacted elements have been identified. An exemplary impact propagation rule is shown in Listing 1. Impacted elements can then be handed over to developers to perform a deeper inspection, or they are used to prepare cost and time estimations based on the amount and types of affected elements.

```
impact rule(Model a, Model b, Change c)
   report b as affected when
      a.type == "UML:Component"
      b.type == "UML:Class"
      c.type == "Rename_component"
      c.target == a
      b.relationTo(a) == "Refinement"
end
```

Listing 1: A simple impact propagation rule

3 Tool Support and Evaluation

As we are concerned with impact analysis between different UML models and Java source code, we chose the EMF-framework[1] as the common basis for our approach. We extended our prototype case tool EMF-Trace[2] for the purpose of multiperspective analysis. The tool is based on the EMFStore[3] repository and provides means for executing SQL-like rules to mine traceability relations among models [1]. Our rule-concept has been extended to carry out the task of rule-based impact analysis as explained in Section 2. Importing the required UML models and Java source code is achieved as depicted by Figure 2. We import UML models from two different CASE tools and convert their XMI-based output to an EMF-based Ecore representation using XSLT. Java source code is imported using the Ecore-mapping features provided by the MoDisco[4] tool environment. We are currently conducting a case study to evaluate our approach with a system comprised of 16500 UML model elements and Java source code artifacts. We compare the results of our approach against manual inspections, to obtain the figures for recall and precision.

[1] http://eclipse.org/modeling/emf/
[2] http://sourceforge.net/projects/emftrace/
[3] http://www.eclipse.org/emfstore/
[4] http://www.eclipse.org/MoDisco/

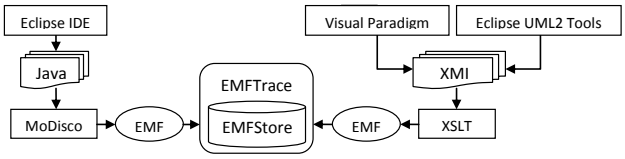


Figure 2: Our tool environment.

4 Conclusion

We outlined typical problems of impact analysis approaches when confronted with multiperspective software designs, and proposed to merge impact analysis with multiperspective consistency checking to tackle the challenges of software evolution. Using the EMF framework, we are able to map different static and dynamic UML models and Java source code on a common base to allow for multiperspective analysis. We extract dependencies from the models, which in combination with the type of applied change, serve as input for our rule-based impact propagation approach. Depending on the type of relation between two models and the change type at hand, our rules then determine further change propagation. We implemented our approach in a prototype tool and are currently performing a first case study to evaluate its performance.

References

[1] S. Bode, S. Lehnert, and M. Riebisch. Comprehensive model integration for dependency identification with EMFTrace. In *Workshop on Model-Driven Software Migration (MDSM 2011)*, pages 17–20, 2011.

[2] M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta. An eclectic approach for change impact analysis. In *32nd ACM/IEEE Int. Conf. on Software Engineering*, pages 163–166, 2010.

[3] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[4] S. Lehnert, Q.-U.-A. Farooq, and M. Riebisch. A taxonomy of change types and its application in software evolution. In *19th Annual IEEE Int. Conference on the Engineering of Computer Based Systems*, 2012.

[5] T. Sunetnanta and A. Finkelstein. Automated consistency checking for multiperspective software specifications. In *Workshop on Advanced Separation of Concerns (ICSE2001)*, 2001.

[6] S. S. Yau, J. S. Collofello, and T. M. McGregor. Ripple effect analysis of software maintenance. In *Computer Software and Applications Conf.*, pages 60–65, 1978.

[7] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

Integrating the Specification and Recognition of Changes in Models

Timo Kehrer, Udo Kelter
Praktische Informatik
Universität Siegen
{kehrer,kelter}@informatik.uni-siegen.de

Gabriele Taentzer
Fachbereich Mathematik und Informatik
Philipps-Universität Marburg
taentzer@mathematik.uni-marburg.de

Abstract

Model-based software development has become a widely accepted approach in application domains where software is long-living and must be maintainable. Models are subject to continuous change and have many versions during their lifetime. The specification and recognition of changes in models is the key to understand and manage the evolution of a model-based system. However, model transformation and model versioning tools currently available are based on low-level operations on models, which are sometimes incomprehensible for users. Thus, both classes of tools should be better integrated and extended to supporting high-level edit operations, e.g. refactoring operations.

1 Introduction

Development technologies for long-living software systems are faced with the huge challenge of supporting the evolution of requirements and platforms. In model-based software development (MBSE), the use of platform-independent models as primary development artifacts reduces maintenance cost caused by the evolution of platforms. However, MBSE does not suffice to successfully manage all aspects of evolution, and actually creates new problems: When requirements change, models must change, too. Models of long-living software systems therefore have many versions during system lifetime. Traditional line-oriented differencing and merging approaches, which are successfully used for the version management of program source code, can usually not be applied to software models. This has triggered a lot of research into model versioning recently [2].

Being able to specify changes between revisions in a precise and meaningful way is of primary importance to understand and plan the evolution of a model-based system. Specifications of changes in models between versions play (a) a *prescriptive* role, i.e. as specification of modifications to be performed on an existing model version, and (b) a *descriptive* role as a means to describe the observed difference between two versions, notably past changes in the history of a model.

Meaningful specifications of changes must reflect the way in which models are edited by users, which particularly includes complex editing operations, a concrete example is provided in Section 2. These complex modifications are already supported by model transformation and refactoring tools [1]. However, their recognition is an open problem which is briefly introduced in Section 3. Thus, typical model versioning tasks such as patching and merging still operate on the basis of low-level changes; they should be lifted to the level of user edit operations. In other words, model editing and model versioning tools must be integrated in the sense that they are based on the same set of edit operations which are applicable from a user's point of view. A summary of our research agenda is given in Section 4.

2 Editing of Models

Changes in models can only be specified precisely if a runtime representation of models is available. To that end, the concept of metamodeling was established in the modeling community. Models are considered as abstract syntax graphs (ASG), metamodels define the types of nodes and edges of the ASG. Metamodels do not directly specify behavior, i.e. *edit operations* which can modify models. Many basic model editing operations, e.g. the creation of a model element of type T or the setting of a property of an element, can be directly deduced from a given metamodel.

However, basic model edit operations can violate well-formedness rules. The UML metamodel, for instance, defines many mutually dependent data items in an ASG and related consistency constraints. Project-specific modeling guidelines can define further constraints. Then a consistency-preserving change of data items requires several basic edit operations, i.e. a complex edit operation. Refactoring operations and tool commands lead to further complex edit operations. Figure 1 shows a sample Simulink block diagram that has been modified as follows: The linear function $f(x) = mx + c$, which processes the observed sine signal, is extracted into a separate subsystem. This effect can be achieved by *one* user-level command, namely the edit operation "Create Subsystem".

3 Differencing of Models

Some approaches to model differencing are based on the logging of edit commands in editors. However, these approaches requires closed environments and do

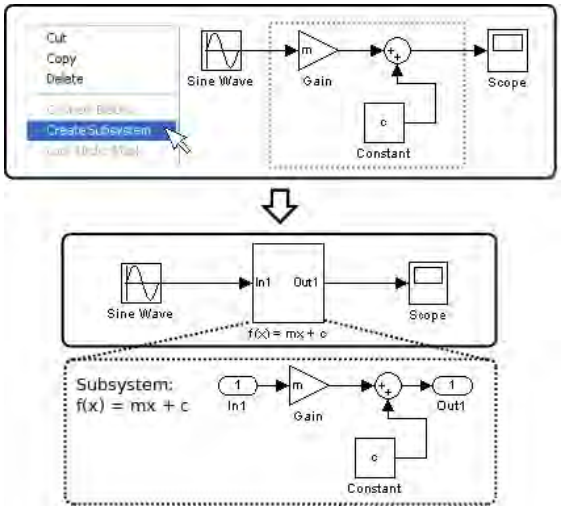


Figure 1: Creation of a subsystem in Matlab/Simulink

not provide a general solution to the problem [6].

State-based approaches compare models on the basis of their state and usually work on an ASG representation of the models [5]. Initially, a matching procedure identifies corresponding model elements and relationships in both models, i.e. corresponding nodes and edges in their ASGs. Model elements and relationships not involved in a correspondence are considered to be deleted or created; these insertions and deletions form a *low-level difference*. Low-level differences often contain pseudo changes which do not make sense from a users' point of view [4]. Developers perceive models in their external, typically graphical representation, and prefer changes to be explained in terms of conceptually meaningful edit operations.

A first approach to overcome this problem is presented in [3]: A *semantic lifter* identifies groups of low-level changes which implement an edit operation. However, this approach does not cover more complex edit operations yet. For example, the subsystem extraction of Figure 1 still leads to several low-level changes which can be briefly summarized as follows: Firstly, the created subsystem is embedded into the parent system by a subsystem block serving as black-box and providing the connector ports "In1" and "Out1". Secondly, within the implementation of the subsystem, the blocks "In1" and "Out1" provide the necessary connection points for incoming and outgoing data flows. Finally, the blocks (gain, constant and add-operator) and respective data flows realizing the linear function are finally relocated into the new subsystem.

4 Integrating Specification and Recognition of Model Changes

Technologies for editing and differencing models must be based on the same set of editing operations if one wants to be able to repeat observed changes (as a patch) and if model comparisons shall report the same complex changes which were actually performed. This is trivial in case of basic graph operations. It is a big challenge if complex edit operations are to be supported.

[3] presents a first approach for lifting model differences to representations of simple edit operations. This approach can be extended towards more complex edit operations. Formalisms for defining complex operations and methods to recognize the resulting changes mutually depend on each other. Our research agenda thus comprises the following sub-goals:

1. Identifying *meaningful complex edit operations* in models. These operations depend a lot on the application domain and the types of models which are used, they must be defined individually for each application domain. Model types used for embedded systems in domains like automation engineering etc., e.g. block diagrams, present a particular challenge due to their size and recursive structure.

2. Providing a comprehensive *specification language for complex edit operations* and developing the underlying theoretical background.

3. Developing *model comparison* algorithms which recognize complex edit operations performed between two revisions of a model.

4. Extending algorithms for *model merging and patching*, including conflict and dependency detection, and model history analysis to support complex edit operations.

References

[1] Arendt, T.; Biermann, E.; Jurack, S.; Krause, C.; Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations; in: Proc. MoDELS 2010; Springer, LNCS 6394; 2010

[2] Bibliography on Comparison and Versioning of Software Models; http://pi.informatik.uni-siegen.de/CVSM

[3] Kehrer, T.; Kelter, U.; Taentzer, G.: A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning; p.163-172 in: Proc. ASE 2011; ACM; 2011

[4] Kelter, U.: Pseudo-Modelldifferenzen und die Phasenabhängigkeit von Metamodellen; p.117-128 in: Proc. SE 2010; LNI 159; 2010

[5] Kolovos, D.S.; Ruscio, D.D.; Pierantonio, A.; Paige, R.F.: Different Models for Model Matching; p.1-6 in: Proc. CVSM 2009; IEEE; 2009

[6] Küster, J.M.; Gerth, C.; Förster, A.; Engels, G.: Detecting and Resolving Process Model Differences in the Absence of a Change Log; p.244-260 in: Proc. BPM 2008; Springer, LNCS 5240; 2008

Digital Preservation: New Challenges for Software Maintenance

Sven Euteneuer, Daniel Draws
SQS Software Quality Systems AG
Stollwerckstraße 11, 51149 Cologne, Germany
Email: sven.euteneuer|daniel.draws@sqs.com

Abstract: Digital preservation ensures continued access to digital information over time (e.g., 5, 15, 25 or 50 years). The long-term planning horizon of preserving relevant parts of a business processes raises a number of new research challenges for software design, development and maintenance (cf. [1]).
In this paper we describe the challenges caused by the preservation of software and give a short overview of the TIMBUS research project that aims to address these challenges.

1 Introduction

Business processes are an orchestration of different services. Many of these services are operated by different service providers. Processes are volatile in terms of services disappearing and fundamental changes in technology. Processes of vital interest in a remote future need to be preserved. Their functionality, usability, integrity and authenticity need to be guaranteed. Changes of technology (e.g., new formats or standards), changes of the environment (e.g., new legal obligations) and the disappearance of services need to be addressed for ensuring the usability of the process and its software in the long run.
Current maintenance approaches for software are dealing with online (and active) systems. Changes in the environment, technology, services and data potentially have immediate effects on the system functionality. For preserved software stored off-line in an archive these changes can remain concealed for years. New approaches for active management and maintenance of preserved information systems are required. The risk of irreconcilable inconsistency of the software system caused by technological obsolescence or unavailability of essential services and software components need to be addressed.
Preservation of software systems raise new challenges for all stages of the software life cycle. Starting from the design phase, well-documented and clearly designed structures of software reduce the dependencies on actual implementation technology. Other examples are requirements on the software quality raised by the fact that source code should be understood in the future.
Maintenance over the long term can include the replacement of obsolete technologies, such as runtime environments, operating systems, networking infrastructures, etc.
The TIMBUS research project addresses these and other research issues, amongst which are authenticity of the preserved information, legal, contractual and regulatory issues and risk management with respect to business process preservation.

2 Digital Preservation

The preservation of digital artefacts was identified as a serious issue and is being addressed by different research efforts over the course of the last years. Digital Preservation -ensuring continued access to digital assets- is traditionally focused on data-centric information. A number of research initiatives have addressed preservation of digital objects, such as office data, video or more complex data structures such as databases. An overview about relevant research projects within the ICT program of the European Commission is given in [2].
The data centric view considers the generation of data from a bitstream. This is addressed by the definition of representation information (c.f. [3]). A powerful representation information is the software that generated the data. Concerning the preservation of information for decades it is obvious that this software would need to be maintained for a long time. The difference to the established maintenance approaches is the fact that archived software stays unchanged for a long period. The development process often has to be re-instantiated in a completely different – and ex-ante unknown – environment. Exclusively preserving and provisioning source artefacts would comply with such a data centric preservation approach.
Unfortunately, the data centric view leaves out the relevance of the data's context. The context transports the semantics required to transform the captured data back into information. It also ignores important aspects of execution such as processing, analysing, transforming, rendering and interpreting.
Workflows and business processes (as well as software development processes) often rely on a variety of sub-processes and distributed, loosely orchestrated and service oriented architectures. Unavailability of data, services or context can lead to loss of information resulting in detrimental effects on the businesses they enable. Digital preservation of processes and services, as it is addressed by TIMBUS, is a novel approach, because it addresses the preservation of information in a holistic manner.

.

3 Holistic Preservation

Successful digital preservation of whole processes requires capturing sufficient detail of a process and its context to be able to re-enable its original behaviour at a future date, involving potentially different participating parties, different enabling technologies, different system components (hardware and software), changed services by different service providers or differences in other aspects of the context of the business process. This implies that there exists a set of activities, processes and tools that ensure continued access to services and software necessary to produce the context within which information can be accessed, properly rendered, validated and transformed into knowledge. Digital preservation of business processes and services therefore requires that this set can be preserved.

Traditional digital preservation approaches have a focus on preserving digital objects and their technical context [5]. However, from a TIMBUS perspective, traditional methods of digital preservation do have a too restricted perspective on context that they take into account. Assuming that a traditional method of digital preservation provides the ability to preserve system components and entire systems in addition to digital artefacts, often, it would not be clear at a given time, which information would be required later to restore the system. Things that constitute a clear boundary of technical feasibility today can be a parameter not accounted for the future.

The approach developed in TIMBUS is expected to identify all relevant artefacts generated and used in a business process. TIMBUS investigates to identify and resolve dependencies under given constraints to ensure re-executability of processes in the distant future. For the long term preservation of software systems with a holistic view all artifacts used in a development and deployment process have to be considered [4]. Besides the technical preservation of the software system, the context of business processes in question may be influenced by a variety of changes in the environment (such as change of legal aspects over time or the change of tacit knowledge). To give an example: Notations like UML are accepted and well understood today. General concepts and most notations can be used by developers without looking into the standard itself. But there is an uncertainty that this will be the same in about 50 years or more. The preservation process proposed in the TIMBUS project consists of three stages of digital preservation:

Expediency of digital preservation effort – establishing the risk of not preserving and the feasibility of digitally preserving business processes.

Execution of digital preservation process – performing the digital preservation of business processes.

Exhumation of digitally preserved assets – re-running a digitally preserved business process.

4 Summary and Outlook

TIMBUS is an extended and more curation-oriented approach for preserving business process. Adopted to software maintenance this implies a more holistic view on a software project also considering other collaterals besides source code. Expected outcomes of the project are amongst others:

- A risk model for business processes with respect to digital preservation.
- A context model for business processes.
- Preservation processes required to preserve business processes in the long term.
- Guidelines for ``digital preservation-aware" process design and implementation (including software development).
- Validation methods for integrity of preserved business processes.
- Overview of legal aspects for preservation of software services and their data.

5 Acknowledgements

Part of this work is funded by the European Commission's Seventh Framework Programme (FP7/2007-2013) under grant agreement No 269940.

6 References

[1] T. Kuny, “A Digital Dark Ages? Challenges in the Preservation of Electronic Information,” Sep. 1997.

[2] S. Strodl, P. Petrov, and A. Rauber, “Research on digital preservation within projects co-funded by the european union in the ict programme,” Vienna University of Technology, Tech. Rep., May 2011.

[3] CCSD, “Reference Model for an Open Archival Information System,” Consultative Committee for Space Data Systems, August 2009.

[3] D. Draws, S. Euteneuer, D. Simon, and F. Simon, “Short Term Preservation for Software Industry,” in *8th Int. Conf. on Preservation of Digital Objects* (iPRES 2011), Singapore, November 2011, pp. 1–9.

[4] D. Giaretta, “The CASPAR Approach to Digital Preservation,” The Internation Journal of Digital Curation, vol. 2, no. 1, 2007.

Automated Source-Level Instrumentation for Dynamic Dependency Analysis of COBOL Systems

Holger Knoche[1], André van Hoorn[2], Wolfgang Goerigk[1], and Wilhelm Hasselbring[2]

[1] b+m Informatik AG, 24109 Melsdorf

[2] Software Engineering Group, University of Kiel, 24098 Kiel

Abstract

Dynamic analysis, e.g. dynamic dependency analysis, requires the injection of monitoring code into an existing application. This paper presents an approach to automatically locate and process the required injection sites in the source code of a COBOL system and discusses issues arising from source-level instrumentation when applied to an industrial case study.

1 Introduction

The analysis of dynamic dependencies of software systems is a valuable source of information, especially in modernization scenarios. The concept of aspect-oriented programming (AOP) [2] has proven to be a helpful tool for inserting the required instrumentation. However, although approaches have been proposed [3], most legacy runtime environments still lack AOP capabilities.

In this paper, we present a pragmatic approach to add instrumentation automatically to an application's source code [1] in an AOP-like way. Furthermore, we present selected results and challenges which arose when we used this approach to perform a dynamic calling dependency analysis of an industrial COBOL case study system.

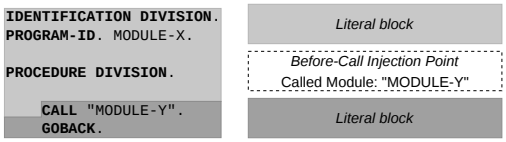
The remainder of this paper is structured as follows. Section 2 provides an overview of our source code instrumentation approach. Section 3 presents selected challenges that we encountered when we applied the approach to an industrial case study. Section 4 discusses the issue of incomplete traces which is likely to arise from source-level instrumentation. Finally, concluding remarks are presented in Section 5.

2 Source-Level Instrumentation

The process of automatically augmenting source code with instrumentation instructions is carried out in two subsequent steps. First, a static analysis component processes the source code using common static analysis techniques. This component produces an object model that represents the input file as a non-empty sequence of *blocks*, which may be either *literal blocks* or *injection points*. The latter are similar to AOP's *join points*, and like these, they may carry information about their static source context.

This work is partly funded by the German Federal Ministry of Education and Research (BMBF) under grant number 01IS10051.

This approach can be illustrated using the small COBOL code example presented in Figure 1(a).



(a) (b)

Figure 1: A simple COBOL example

Suppose we want to add instrumentation code before every `CALL` statement. In this case, the static analysis component could produce the following block sequence: one literal block from the beginning of the module to the beginning of the `CALL` keyword, one *before-call* injection point carrying the called module name as additional context information, and one literal block from the beginning of the `CALL` statement to the end of the module, as shown in Figure 1(b).

In a second step, the block stream is fed into a code-generation component which produces the instrumented code. When encountering an injection point, the code generator selects and expands an appropriate, user-provided code template depending on the type of the point. Thus, it is possible to exploit the provided specific context information during template expansion. The templates can therefore be seen as *advices* with simple *pointcut designators* that only depend on the point type.

Although this approach provides a flexible way of instrumenting source code and works well in practice, it has one major drawback: it can only be applied to source code written in a supported programming language. In heterogenous environments with third-party components, it is therefore likely that some components remain uninstrumented. Section 4 presents a way to handle this issue for dynamic calling dependency analysis.

3 Application to a Case Study

In order to evaluate the described approach, we applied it to perform a dynamic calling dependency analysis of a substantial industrial application consisting of more than 1,000 COBOL modules. Selected challenges that were encountered during this process are described below.

Static analysis of COBOL programs, which is required to locate the injection points, is challenging due to various reasons. First of all, many different dialects of and vendor-specific extensions to the language exist. In addition, many COBOL modules contain embedded transaction control or SQL statements that are replaced by specific preprocessors during the build process. However, particular transaction control statements are important for dependency analysis because they may affect the control flow.

In order to avoid implementing a complete COBOL grammar, we employed the concept of *island grammars* [4], which works particularly well for COBOL because of the language's many keywords. Since the code is only partially parsed in detail, such grammars tend to be more robust with respect to dialects and unknown constructs.

Albeit being helpful, island grammars have some subtle pitfalls. While most injections had to be inserted at the beginning of the respective statement, which was easy to find, some had to be added at the end. The location of these points proved to be challenging. The most notable example was the `PERFORM` statement, which had to be analyzed in detail not only to locate its end, but also to distinguish its procedural variant, which should be instrumented, from the inline variant.

Another challenge for both static analysis and code generation was COBOL's fixed source code format. Since common parser infrastructures are designed for free-format languages, it took some non-trivial adjustments to introduce format sensitivity. The main challenge for the code generator was to maintain the correct formatting after injecting code. Since the COBOL file format allows arbitrary data in certain columns, the code generator must preserve the exact start column of literal blocks when injections are made to prevent such data from slipping into the code area.

During our evaluation, we successfully managed to track module calls via both native COBOL mechanisms (`CALL`) and the employed transaction manager (`EXEC CICS LINK` and `EXEC CICS XCTL`) as well as section invocations (procedural `PERFORM`). A total of 140,351 injection points of 14 different types were inserted for the analysis.

4 Incomplete Event Traces

In order to collect trace information, we employ an event-oriented logging mechanism which logs call, entry, and exit events for modules and sections, respectively, into a database. These events are then transformed into a format compatible with our Kieker analysis and visualization framework [6].

The reason for monitoring both call and entry events is that this seemingly redundant information allows to partially reconstruct message traces involving uninstrumented components, such as third-party libraries for which no source code is available. We distinguish the following cases: (i) definite call: call $A \rightarrow B$, entry into B; (ii) immediate return from uninstrumented component: call $A \rightarrow C$, next logged event happens in A. In this case, we assume that C is successfully called and returns control to A. Note that there will always be a next event in A due to exit events; (iii) call through an uninstrumented component: call $A \rightarrow D$, entry into E. In this case, we assume that A has successfully called D and D (transitively) called E in the process; (iv) uncalled section: entry into a section S without a preceding call to S. In this case, we assume that the control flow has run through the section and do not interpret the situation as a call.

In order to visually distinguish assumed from definite call dependencies and components, we extended the Kieker framework. An example illustrating the aforementioned cases is shown in Figure 2. Assumed dependencies are shown using dashed lines, and assumed components are shaded.

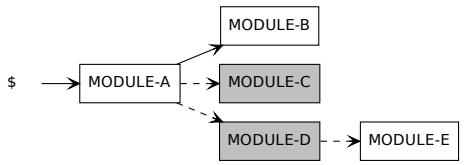


Figure 2: Module-level call dependency graph with assumed dependencies

5 Conclusions

Although we only implemented a small subset of the common AOP features, we were able to successfully perform a dynamic dependency analysis of an industrial case study application. The instrumentation approach integrates seamlessly with our earlier work [5], and its modular design allows to apply the approach to other analysis scenarios as well as to other languages and environments, what we plan for our future work.

References

[1] T. Chen, H. Kao, M. Luk, and W. Ying. COD — A dynamic data flow analysis system for Cobol. *Information & Management*, 12(2):65 – 72, 1987.

[2] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97*, pages 220–242. 1997.

[3] R. Lämmel and K. De Schutter. What does aspect-oriented programming mean to Cobol? In *AOSD '05*, pages 99–110, 2005.

[4] L. Moonen. Generating robust parsers using island grammars. In *WCRE '01*, pages 13–22, 2001.

[5] A. van Hoorn, H. Knoche, W. Goerigk, and W. Hasselbring. Model-driven instrumentation for dynamic analysis of legacy software systems. In *WSR '11*, pages 26–27, 2011.

[6] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *ICPE '12*, 2012. To appear.

Interaktive Extraktion von Software-Komponenten

Fabian Beck, Alexander Pavel und Stephan Diehl

Universität Trier
54286 Trier, Germany

{beckf,diehl}@uni-trier.de

Zusammenfassung

Im Rahmen der Wiederverwendung von Software, des Software-Redesigns oder des Outsourcings kann es sinnvoll sein, eine Komponente eines Software-Systems herauszutrennen, damit dieses Teilsystem unabhängig und effizient weiterentwickelt werden kann. Schwerpunkt dieser Arbeit ist das Design und die Realisierung einer Benutzerschnittstelle, die ein semiautomatisches Verfahren zur Komponentenextraktion nahtlos in die Entwicklungsumgebung Eclipse integriert und dabei eine interaktive und iterative Arbeitsweise unterstützt. Eine besondere Herausforderung stellt die effiziente Darstellung von Abhängigkeiten zwischen extrahierter Komponente und Restsystem dar.

1 Einleitung

Eine Software-Komponente wird gewöhnlich als ein Teil eines Software-Systems verstanden, der mit einer klare Schnittstelle versehen ist, unabhängig ausgeliefert wird und durch Dritte weiterverwendet werden kann [2]. Um ein größeres Anwendungsgebiet zu adressieren, wollen wir im Kontext dieser Arbeit jedoch den Begriff der Komponente etwas weiter fassen: Zusätzlich zu einer klar definierten Schnittstelle fordern wir nur, dass die Komponente unabhängig vom Restsystem weiterentwickelt werden kann. Eine Komponente muss also nicht eigenständig lauffähig sein, sondern kann weiterhin stärkere Abhängigkeiten zu anderen Teilen des Systems aufweisen. Möchte man eine solche Komponente zwecks eigenständiger Weiterentwicklung extrahieren, ist es dennoch vorrangiges Ziel, die Anzahl solcher Abhängigkeiten zu reduzieren – jede Abhängigkeit im Programmtext schafft Abhängigkeiten in der Software-Entwicklung, die einen raschen Fortschritt potentiell gefährden können.

Das hier vorgestellte Werkzeug zur interaktiven Komponentenextraktion basiert auf dem semiautomatischen Verfahren von Marx et al. [1], welches das Software-System als ein durch Abhängigkeiten verbundenes Netzwerk aus Klassen versteht und wie folgt arbeitet: Nachdem der Benutzer eine zu extrahierende Komponente durch Angabe zentraler Klassen grob skizziert hat, berechnet das Werkzeug den minimalen Schnitt im Netzwerk zwischen den zu extrahierenden Klassen und dem Kern des Restsystems (st-Cut-Problem) und schlägt so eine genaue Aufteilung des Systems vor. Zusätzlich wird automatisiert eine Schnittstelle für die so geschaffene Komponente erzeugt.

2 Interaktive Komponentenextraktion

In der Arbeit von Marx et al. [1] wird bereits ein interaktives Werkzeug zur Komponentenextraktion als eigenständige Anwendung beschrieben und in einer qualitativen Benutzerstudie evaluiert. Die im Folgenden eingeführte Benutzerschnittstelle ist als Weiterentwicklung des ursprünglichen Werkzeugs zu verstehen. Sie ist als Plug-In für die Entwicklungsumgebung Eclipse exemplarisch implementiert und erlaubt die Extraktion von Java-Komponenten.

Aus der vorherigen Arbeit übernommen wurde ein dreiteiliger Aufbau der Hauptansicht des Werkzeugs (Abbildung 1): Links befindet sich das Ursprungssystem (*original component*), rechts die zu extrahierende Komponente (*extracted component*); getrennt sind beide durch eine Repräsentation der Komponenten-Schnittstelle (*contract*). Diese Ansicht ist als *view* in Eclipse realisiert und kann gleichberechtigt zu anderen Ansichten angezeigt und genutzt werden.

Die drei Teile der Hauptansicht wurden mit Hinblick auf eine verbesserte Skalierbarkeit neu gestaltet. Sie stellen jeweils eine Menge von Klassen und Interfaces in einer durch die Package-Struktur des Software-Projekts definierten hierarchischen Gliederung dar. Wie im *Package Explorer* von Eclipse können Packages bis auf Methodenebene interaktiv auf- und zugeklappt werden. Der Benutzer weist nun interaktiv Elemente zu den Komponenten fest zu (schwarze Schrift). Der automatische Algorithmus zur Komponentenextraktion teilt die übrigen, grau dargestellten Elemente optimal auf die Komponenten auf. Diese Aufteilung kann iterativ verfeinert und verbessert werden.

Besonderes Merkmal der hierarchischen Darstellung der Komponenten ist eine aggregierte Repräsentation von Abhängigkeiten, welche die verschiedenen Software-Komponenten verbinden. Wir betrachten dazu drei aus UML entlehnte Typen von Abhän-

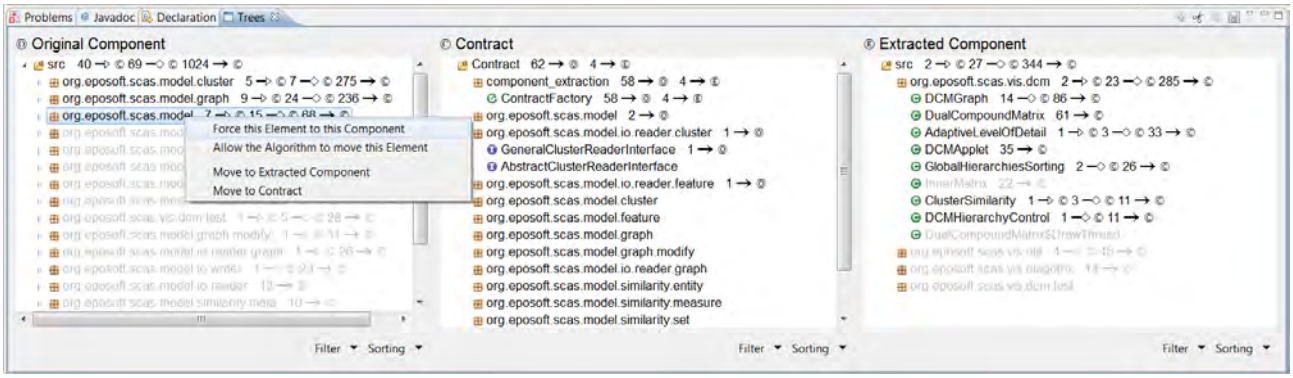


Abbildung 1: Dreigeteilte Hauptansicht des Eclipse Plug-Ins zur Komponentenextraktion.

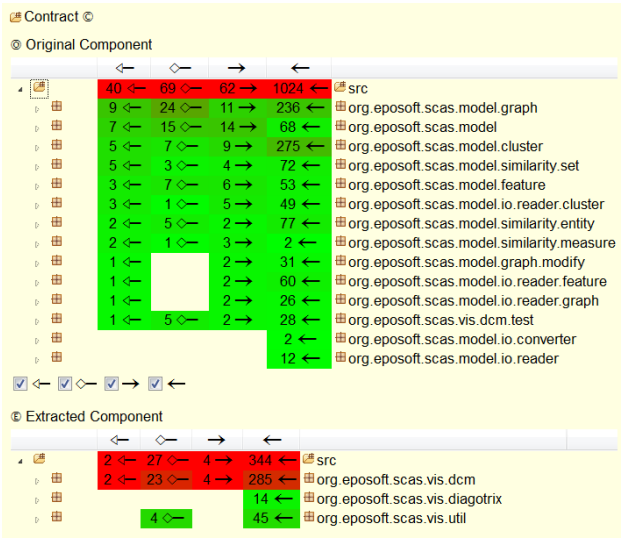


Abbildung 2: Tooltip zur detaillierten Auflistung von Abhängigkeiten.

gigkeiten: Vererbung (—▷), Aggregation (—◇) und Assoziation (→). Unter Verwendung der angegebenen Symbole wird für jedes Hierarchieelement angegeben, wie viele solcher Abhängigkeiten zu den jeweils anderen Software-Komponenten existieren (original component: ⓞ; contract: ©; extracted component: Ⓔ). Abhängigkeiten innerhalb einer Komponente sind hier weniger von Bedeutung und werden nicht referenziert. Durch eine Sortierung nach Anzahl der Abhängigkeiten können die problematischsten Stellen der momentanen Komponentenaufteilung, d.h. die am meisten Abhängigkeiten verursachen, schnell erkannt werden.

Zusätzlich zu dieser übersichtlichen, aggregierten Darstellungen ist es auch möglich, die Abhängigkeiten im Detail abzufragen. Dies geschieht über einen Tooltip, der zum jeweiligen Element eingeblendet wird und in ein permanent angezeigtes Fenster umgewandelt werden kann (Abbildung 2). Ein solcher Tooltip zeigt alle relevanten Abhängigkeiten aus der Perspektive des gewählten Elements. Unterschieden wird hierbei erneut der Typ der Abhängigkeiten und zu welcher Komponente diese gerichtet sind. Die Anzahl der Abhängigkeiten ist zur besseren Übersicht tabellarisch angegeben; die Zeilen der Tabellen sind weiterhin hierarchisch gegliedert. Durch eine Farbskala, die die Anzahl der Abhängigkeiten kodiert und spaltenweise normalisiert ist, wird direkt auf potentiell kritische Komponentenzuordnungen hingewiesen. Mittels Doppelklick kann jederzeit der Quelltext der angeklickten Klasse geöffnet und bearbeitet werden. Die Darstellung aus der Perspektive einzelner Elemente erlaubt es, auch große Netzwerke von Abhängigkeiten einfach zu untersuchen und mögliche Probleme geziehlt aufzulösen.

3 Fazit

Das vorgestellte Eclipse Plug-In setzt ein semiautomatisches Verfahren zur Komponentenextraktion als interaktives Werkzeug um. Der entschiedene Vorteil gegenüber der ursprünglichen Benutzerschnittstelle [1] ist die übersichtliche und effektive Repräsentation der Komponenten und deren Abhängigkeiten, die nun auch den Umgang mit größeren Software-Projekten erlaubt. Durch die Darstellung der Abhängigkeiten als explizite Verbindungslinien konnten bisher nur kleine Projekte sinnvoll bearbeitet werden. Zudem erlaubt die nahtlose Integration in die Entwicklungsumgebung Eclipse den einfachen Wechsel zwischen Quelltext-Bearbeitung und Komponentenextraktion. Diese Möglichkeit wurde in der Evaluierung des ursprünglichen Werkzeugs als fehlendes, wichtiges Feature identifiziert. Insgesamt verspricht die interaktive Extraktion von Komponenten die Weiterentwicklung eines Software-Systems zu erleichtern.

Literatur

[1] A. Marx, F. Beck, and S. Diehl. Computer-Aided extraction of software components. In *17th Working Conference on Reverse Engineering (WCRE 2010)*, 2010.

[2] C. Szyperski and C. Pfister. Workshop on component-oriented programming, summary. In M. Mühlhäuser, editor, *Special Issues in Object-Oriented Programming - ECOOP '96 Workshop Reader*. dpunkt Verlag, 1997.

MAMBA: Model-Based Software Analysis Utilizing OMG's SMM

Sören Frey, André van Hoorn, Reiner Jung, Benjamin Kiel, and Wilhelm Hasselbring
Software Engineering Group, University of Kiel, 24098 Kiel

Abstract

Most software system properties can be quantified through applying measurement processes. OMG's Structured Metrics Meta-Model (SMM) supports the meta-model agnostic definition of those measurement processes with an emphasis on architecture-driven modernization scenarios. We present the MAMBA framework that addresses major obstacles software engineers currently face when using SMM in practice. Among those are (1) the lack of appropriate tool support, (2) the cumbersome integration of precomputed measurement data, and (3) the complexity of specifying SMM models and queries.

1 Introduction

To statically or dynamically analyze a software system in a re-engineering context, selected quality attributes are often quantified following a measurement approach [3]. For example, the monitoring of response times during runtime or the determination of the module coupling metric as a decision support for restructuring a software architecture constitute measurement processes. To enable a meta-model agnostic definition of measurement processes, the Architecture-Driven Modernization (ADM) Task Force,[1] a sub-commitee of the Object Management Group (OMG), developed the Structured Metrics Meta-Model (SMM) [2]. In our previous work [1], we introduced MAMBA, a measurement architecture for model-based analysis that builds on SMM. Instances of SMM can be computed with our MAMBA framework or used for querying system models with the MAMBA Query Language (MQL).

In this paper, we (1) detail the basic metrics computation process of our tool,[2] (2) briefly describe how externally computed measurement data can be integrated, and (3) introduce the Metrics Definition Language (MDL) for simplifying the specification of the typically complex SMM models.

The remainder of the paper is structured as follows. Section 2 outlines the SMM concepts relevant to this paper. The MAMBA framework is presented in Section 3, before Section 4 draws the conclusions and describes future work.

This work is partly supported by the German Federal Ministry of Education and Research (BMBF) under grant number 01IS10051, the Program for the Future Economy of Schleswig-Holstein, and the European Regional Development Fund (ERDF).

[1] `http://adm.omg.org/`

[2] `http://mamba-framework.sf.net/`

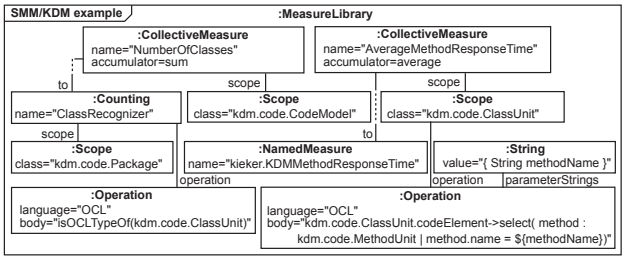


Figure 1: SMM example for KDM domain models

2 Structured Metrics Meta-Model

Central notions in the SMM standard are *measures* and *measurements*. A measure describes an algorithm for calculating specific properties of software system elements. A typical static measure is the number of code lines (LOC) that are contained in each class; an example dynamic measure is the average response time of software methods. In order to use an SMM measure, it must be contained in a *measure library*. SMM includes basic measures, such as *binary* (e.g., $\leq$) and *collective* (e.g., *sum*) measures. User-defined measures can be defined utilizing *direct* or *named* measures. Direct measures can be specified incorporating a query language (e.g., OCL); for *named measures* no executable semantics exist. A measurement is the counterpart of a measure, being produced through executing the latter. A model element that was measured is referenced via a *measurand* relationship of the respective measurement. Measurements observed in a concrete measurement process are grouped in *observations*, associated with additional information, such as the time of the measurement. An example SMM measure library is shown in Fig. 1.

3 MAMBA Framework

Fig. 2 provides an architectural view of the MAMBA framework in terms of core components and usage. The general idea is that users provide a set of domain models, e.g., instances of the Knowledge Discovery Meta-Model (KDM), along with a definition of requested measures in form of SMM models or textual representations which MAMBA translates to SMM. The framework executes the input SMM models and outputs these SMM models enriched by measurements (observations) for the requested measures. Note that the SMM models may contain additional MAMBA-provided extensions, such as user-defined (periodic) aggregate functions [1]. Measurement Providers are used to integrate external static or dynamic analy-

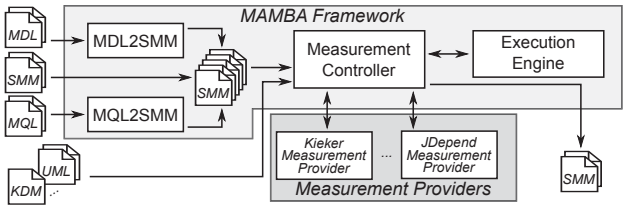


Figure 2: Framework with measurement providers

sis tools, e.g., by executing these and importing the resulting raw measurement data. Fig. 2 indicates the integration of the dynamic and, respectively, static analysis tools Kieker[3] and JDepend.[4]

3.1 Execution of SMM Models

The Measurement Controller creates an instance of the Execution Engine and passes the SMM models, including the list of requested measures, to the latter. As a first step, the Execution Engine inspects the SMM models in order to determine whether at least one named measure is required for the computation of the requested measures. We distinguish between two different modes of execution: if no dependency to a named measure exists (*closed mode*), the Execution Engine can directly compute the SMM measurements simply based on the domain model(s); otherwise (*open mode*), the Execution Engine provides the list of required named measures to the Measurement Controller, which initializes appropriate Measurement Providers for these named measures. The Measurement Providers create observations (i.e., measurements for named measures, including additional meta-information) that are passed to the Execution Engine via the Measurement Controller. After the termination of each Measurement Provider, the Execution Engine executes the SMM models just like the way it executes in closed mode. For reasons of brevity, we do not detail the execution of periodic measures in this paper.

3.2 Metrics Definition Language (MDL)

MDL provides an easy-to-understand and compact textual representation of measure definitions. Fig. 3a shows the MDL representation for the SMM measure library from Fig. 1 in our MAMBA IDE.

Each MDL model comprises a library name, a list of domain meta-models (e.g., KDM), and a collection of measure definitions. These measure definitions follow the same pattern: they start with a keyword, identifying the measure type, followed by a unique name, an optional list of parameter declarations, specific measure properties, and a scope definition. The scope can be any class from an imported domain meta-model or a more complex selector expression, as illustrated with the AverageMethodResponseTime measure in the example.

[3] http://kieker-monitoring.net/
[4] http://clarkware.com/software/JDepend.html

library.mdl | queries.mql | kdm.ecore

```
library example

model kdm "http://schema.omg.org/spec/KDM/1.2"

collect NumberOfClasses sum ClassRecognizer
    scope kdm.code.CodeModel

count ClassRecognizer
    select isOCLTypeof(kdm.code.ClassUnit)
    scope kdm.code.Package

collect AverageMethodResponseTime ( String methodName )
    average KDMMethodResponseTime
    scope ( kdm.code.ClassUnit.codeElement method |
            method.name == methodName && method instanceof kdm.code.MethodUnit)

measure KDMMethodResponseTime
```

(a) Measure library defined with MDL

library.mdl | queries.mql | kdm.ecore

```
library example

model myApp "myApplicationKdmModel.kdm"

select AverageMethodResponseTime("exampleMethod") as avgrt
    from myApp where avgrt > 500 group by kdm.code.MethodUnit
```

(b) MQL query example

Figure 3: MDL and MQL examples

3.3 MAMBA Query Language (MQL)

MQL applies SMM measures to concrete domain models. Therefore, it references one or more measure libraries and one or more domain models. The application of measures is specified with a set of queries which can provide further analysis. MQL2SMM (see Fig. 2) extends the referenced measure library by transparently adding auxiliary measures. The example in Fig. 3b utilizes the AverageMethodResponseTime measure and returns those average response times which exceed 500 milliseconds.

4 Conclusions and Future Work

OMG's SMM [2] enables the meta-model agnostic definition of measurement processes. However, specifying SMM models is cumbersome and error-prone. Our presented MAMBA framework provides tool support for a simplified definition of those SMM models and enables their automatic execution. Currently, the core components of MAMBA are implemented and have been partly evaluated in industrial contexts [1]. The MDL and MQL components are in a prototype stage and will be refined in our future work.

References

[1] S. Frey, A. van Hoorn, R. Jung, W. Hasselbring, and B. Kiel. MAMBA: A measurement architecture for model-based analysis. Technical Report TR-1112, Department of Computer Science, University of Kiel, Germany, Dec. 2011.

[2] Object Management Group. Architecture-Driven Modernization (ADM): Structured Metrics Meta-Model (SMM) Version 1.0. http://www.omg.org/spec/SMM/1.0/, 2012.

[3] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Quality-driven software re-engineering. *Journal of Systems and Software*, 66(3):225 – 239, 2003.

Oberflächenmodernisierung mit MaTriX

Uwe Erdmenger
pro et con Innovative Informatikanwendungen GmbH, Dittesstraße 15, 09126 Chemnitz
uwe.erdmenger@proetcon.de

Abstract

Die Modernisierung von Benutzeroberflächen ist ein wichtiges Thema in Softwaremigrationsprojekten. Dabei beschleunigt der Einsatz von Tools das Projekt und gestaltet es kostengünstiger. Gegenstand der folgenden Ausführungen ist *MaTriX* [1], ein Werkzeug für die Entwicklung und Modernisierung von Bildschirmmasken unter Beibehaltung der verarbeitenden Serverprogramme sowie der Middleware und unter Verwendung moderner Webtechnologien wie Ajax und HTML 5.

1 Motivation

Ausgangspunkt der Betrachtungen ist ein Message-basiertes Programmsystem, in welchem Bildschirmmasken mit Serverprogrammen durch Austausch von Messages (Telegrammen), die als COBOL-Strukturen definiert sind, kommunizieren. Für die Modernisierung solcher Benutzeroberflächen gibt es verschiedene Ansätze. Eine Möglichkeit ist die Maskenmigration unter Nutzung von Werkzeugen [1]. Bei einer Neuentwicklung kommen häufig Java-EE-Techniken (JSP, JSF, ...) zum Einsatz [2]. Dabei werden die Masken z.B. als JSPs oder Facelets neu erstellt, wobei auf umfangreiche Bibliotheken an Oberflächenelementen zurückgegriffen werden kann. Dieser Ansatz bringt jedoch auch eine Reihe von Nachteilen mit sich: Zusätzlich zur Middleware wird nun noch ein Application Server (z.B. JBoss) benötigt, welcher Administrationsaufwand und Serverlast verursacht. Diese wird noch erhöht durch die Notwendigkeit einer Schnittstelle, welche den Zugriff auf einzelne Felder der COBOL-Message von Java aus ermöglicht und dazu Kenntnisse des strukturellen Aufbaus jeder Message benötigt. Darüber hinaus müssen die Entwickler, die das Anwendungssystem warten und weiterentwickeln, die notwendigen Java-EE-Kenntnisse erwerben, wozu nicht immer Bereitschaft besteht.

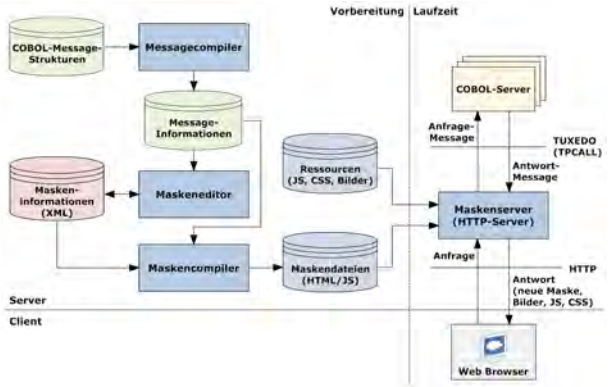
Ein Ziel der Entwicklung von MaTriX war es, möglichst *wenig Rechenlast auf dem Server* zu erzeugen. Die Auswertung der COBOL-Messages sollte erst auf dem Client erfolgen, die Realisierung aber dennoch *browserbasiert* sein und *keine zusätzliche clientseitige Installation* einer Software erfordern. Eine weitere Forderung war der *Einsatz moderner Web-Technologien* und die einfache Anpassbarkeit an neue technische Möglichkeiten, die sich z.B. durch HTML 5 bieten.

Die Entwicklung der neuen Oberfläche sollte einfach sein und keine Einarbeitung in eine neue Programmier- oder Auszeichnungssprache erfordern. Voraussetzung dafür war die Bereitstellung eines *grafischen Editors* (Maskeneditor), mit dem die Masken neu erstellt und bearbeitet werden können.

[1] MaTriX wird aus Mitteln der Europäischen Union und des Freistaates Sachsen gefördert.

2 Architektur

Die folgende Grafik verdeutlicht die prinzipielle Architektur von MaTriX:



MaTriX setzt sich aus vier zentralen Bestandteilen zusammen: Messagecompiler, Maskeneditor, Maskencompiler sowie einem Maskenserver. Diese Komponenten sowie ihr Zusammenwirken sollen im Weiteren erläutert werden.

Messagecompiler: Die Messages stellen die Schnittstelle zwischen den neu zu erstellenden Masken und den vorhandenen Serverprogrammen dar. Sie liegen als COBOL-Strukturen vor. Damit die anderen Komponenten auf die einzelnen Felder einer Message zugreifen können, ist eine Auswertung der COBOL-Struktur erforderlich. Es werden Informationen zu Namen, Verschachtelung und Typ der Felder ermittelt sowie deren Länge und der Offset berechnet und in einem definierten Format in Files abgelegt. Für die erforderliche syntaktische Analyse kommt ein von *pro et con* mit firmeneigenen Metawerkzeugen [3] entwickeltes COBOL-Frontend zum Einsatz.

Maskeneditor: Zum Erstellung und Bearbeiten von Masken wurde ein grafischer Editor entwickelt. Dadurch erübrigt sich das Erlernen einer speziellen Auszeichnungssprache (XML). Er greift auf die abgelegten Message-Informationen zu und gestattet damit eine dialogbasierte Bezugnahme auf Message-Felder. Darüber hinaus bietet er eine Vielzahl von Oberflächenelementen, eine flexible Möglichkeit für deren Anordnung in einem Grid-Layout, die Möglichkeit der Definition und Verwendung von Teilmasken und Maskentemplates usw. Er wurde als Web-Anwendung unter Verwendung von *jQuery* realisiert, so dass für ihn keine lokale Installation erforderlich ist.

Speicherung der Maskeninformationen: Der Editor legt die Maskeninformationen in XML-Files ab. Zu diesem Zweck wurde eine XML-Sprache definiert, welche leicht erweitert und an neue Anforderungen angepasst werden kann. Es existieren Elemente für Textanzeige, Eingabe (Input-Felder, Checkboxen, Selectboxen, Comboboxen, Kalender für Datumswerte uvm.), Aktionen (Links, But-

tons, Tastatureingaben), Layout, ... Ein besonderes Element für die häufig benötigten Tabellen wurde auf Basis der Funktionalität des jQuery-Plugins *DataTables* entwickelt. Es ist über einen Editordialog vielseitig konfigurierbar und erleichtert die Anzeige großer Datenmengen.

Maskencompiler: Da in dieser gespeicherten Form die Maskendaten nicht direkt vom Browser verarbeitet werden können, ist eine Übersetzung erforderlich. Sie erfolgt automatisch immer direkt beim Speichern des XML-Files auf dem Serversystem. Der Vorteil dieses Vorgehens ist eine Verlagerung der Kompilierung in die vorbereitende Phase, wodurch Server und Client zur Laufzeit entlastet werden. Für die Erstellung des Maskencompilers wurde die Programmiersprache *Perl* gewählt, da sie für die Verarbeitung von Text sehr geeignet ist und auch über performante Bibliotheken für den Umgang mit XML verfügt. Das Ergebnis der Übersetzung ist nicht direkt HTML, sondern ein JavaScript-File, welches bei Abarbeitung zur Laufzeit den notwendigen DOM-Tree aufbaut und anzeigt. Auf diese Weise können die Informationen aus der Message, die ja erst zur Laufzeit aktuell vorliegt, leichter eingebunden werden. Die in der XML-Definition `<input value="#{FD-MESSAGE.INOUT.BENUTZ}"/>` eines Eingabefeldes enthaltene Referenz auf das Messagefeld `BENUTZ` wird beispielsweise übersetzt in `inp_13.value = getMessageAlnum(182, 8);`, wobei 182 der berechnete Offset des Feldes in der Message und 8 seine Länge ist.

Maskenserver: Diese Komponente arbeitet als HTTP-Server und stellt zur Laufzeit die generierten Maskendaten, aber auch andere Ressourcen wie z.B. CSS-Files oder Bilder bereit. Gleichzeitig arbeitet sie als Client für die Middleware, indem sie die vom Browser empfangenen Messages an diese weitergibt, die Antwort abwartet und diese dann wieder zurück an den Browser schickt. Dabei wird der Message-Inhalt nicht verändert, so dass keine Kenntniss über ihren Aufbau erforderlich ist. Prinzipiell kann jeder Webserver entsprechend erweitert werden, wobei jedoch auf eine effiziente Anbindung der Middleware zu achten ist, um keine Performance-Lecks zu erzeugen.

Laufzeitbibliothek: Eine JavaScript-Bibliothek übernimmt die clientseitige Ablaufsteuerung. Dazu gehören z.B. das Senden und Empfangen der Messages, deren Auswertung, das ggf. notwendige Nachladen neuer Masken und Übertragen der Message-Daten in die Maske. Die gesamte Kommunikation wird über Ajax realisiert. Dadurch ist es möglich, Informationen auf dem Client (im Browser) zwischenzuspeichern und nur die sich ändernden Informationen neu zu übertragen. Somit ergibt sich die folgende Ablaufsteuerung: Nach dem Start von MaTriX wird eine initiale Message gelesen, ausgewertet und dann die dazu passende (Start-)Maske nachgeladen und inkl. der aktuellen Messagedaten angezeigt. Nach erfolgten Eingaben in der Maske wird mit diesen Daten eine neue Anfrage-Message erstellt und an den Server gesendet, welcher über die Middleware eine Antwort-Message berechnet und zurück sendet. Der gesamte Ablauf wiederholt sich zyklisch solange die MaTriX-Anwendung aktiv ist.

3 Erfahrungen und Ausblick

MaTriX wurde bereits für die Entwicklung von Masken in einem produktiven Umfeld eingesetzt. Serverseitig kommen bei dieser Anwendung COBOL-Programme unter der Steuerung der Middleware *Tuxedo* zum Einsatz. In einem Pilotprojekt wurden eine Reihe bereits vorhandener Masken mit dem neuen System reimplementiert, um den Anwendungsentwicklern und auch den Anwendern eine Möglichkeit des Vergleichs zu geben. Die ersten Erfahrungen bestätigen die Eignung des gewählten Ansatzes. Insbesondere die neuen Gestaltungsmöglichkeiten, wie z.B. scrollbare Tabellen und die damit verbundene Annäherung der Maskenanzeige an das von Web-Anwendungen her bekannte Design wurden positiv aufgenommen. Gleichzeitig konnten die späteren Anwender damit frühzeitig in das Projekt eingebunden werden und Einfluss auf dessen Ausgestaltung nehmen. Im Ergebnis dessen wurde z.B. eine History eingebaut, welche es gestattet, unter Aktualisierung des Inhaltes auf die 20 zuletzt besuchten Masken zurückzugehen. Dieses Verhalten war mit dem bisherigen System nur sehr umständlich mit Hilfe einer Zwischenspeicherung von Messages in der Datenbank realisierbar.

Die in o.g. Projekt verwandten Masken wurden mit dem Maskeneditor neu erstellt. Dieses Verfahren ist jedoch bei einer größeren Anzahl von zu migrierenden „Alt-Masken“ nicht effektiv. Daher sollen in einem nächsten Schritt die Möglichkeiten einer (teil-)automatischen Konvertierung untersucht werden. Dabei sind kundenspezifisch verschiedene Maskenformate auszuwerten und in die XML-Sprache zu migrieren. Für Masken, die in SCREEN COBOL vorliegen, kann zu diesem Zweck mit *ScreenConv* ein bereits existentes anderes Werkzeug von *pro et con* zum Einsatz kommen. Eine besondere Herausforderung ist die Ablösung der bisherigen Positionierung mit festen X/Y-Koordinaten durch das Grid-Layout. Dabei wird ein hoher Automatisierungsgrad angestrebt, um einen Übergang zu MaTriX so reibungslos wie möglich zu gestalten.

Literaturverzeichnis

[1] Erdmenger, U.; Kaiser,U.; Loos, A.; Uhlig, D.:
Methoden u. Werkzeuge für die Software Migration.
In: Gimnich, R.; Kaiser, U.; Quante, J.; Winter, A. (Eds.): 10th Workshop Software-Reengineering (WSR 2008), 5.-7. May 2008, Bad Honnef.
Lecture Notes in Informatics, (LNI)-Proceedings, Volume P-126, S. 83-97

[2] Bodhuin, T. et al.: Migrating COBOL systems to the Web by using the MVC design pattern.
In: Proceedings Ninth Working Conference on Reverse Engineering WCRE 2002,
ISSN 1095-1350, S. 329–338

[3] Erdmenger, U.: Der Parsergenerator BTRACC2.
11. Workshop Software-Reengineering (WSR 2009), Bad Honnef 4.-6. Mai 2009
In: GI-Softwaretechnik-Trends, Band 29, Heft 2,
ISSN 0720-8928, S. 34 und 35

When Program Comprehension Met Bug Fixing

Jochen Quante

Corporate Sector Research and Advance Engineering
Robert Bosch GmbH, Stuttgart, Germany

Jochen.Quante@de.bosch.com

Abstract

Localizing and fixing bugs requires a certain amount of program understanding to be successful. In this paper, we report about a recently conducted "program comprehension challenge", where the task was to find and fix bugs – but the focus was on program comprehension. Some participants used fault localization techniques, others built on different kinds of static analysis techniques. We present the approaches and results of the challenge, and discuss the relation between program comprehension, fault localization, and bug fixing.

1 Introduction

Last year's International Conference on Program Comprehension (ICPC) gave the program comprehension community the opportunity to analyze some (artificial, but realistic) embedded code and see what they can find out about it with their approaches and tools [1][1]. Bosch provided an artificial robot leg control software, along with a simulation environment and some bug reports. The challenge for the participants was to find and fix all the bugs from the bug reports. The underlying idea was that a certain level of program understanding would be required to solve this task. Our expectation was that people from the program comprehension community would try their approaches on this code – a kind of code that is usually not available to the scientific community. However, we also got some submissions using automated fault localization.

2 The Challenge

The subject software was a robot controller. It was realized as an artificially created piece of C code that contained typical elements of embedded control software, such as filters, ramps, curves, and lots of application parameters [4]. As a part of the story, the documentation of this software was lost a long time ago (by an unfortunate combination of mischances), and the original developers are no longer available. Meanwhile, many people have changed the code, and no one really understands it any more. And this is where our participants stepped in. The participants got the following artefacts to work on:

- The robot leg controller code (∼300 lines of C code, ∼35 application parameters),
- a test environment, consisting of a simulation of the robot leg and a test driver (∼600 lines of C code),
- three bug reports, along with test cases that show the erroneous behaviour and the corresponding log files,
- documentation (i. e., description of general idea and application parameters), and
- an acceptance test script to validate correct controller output.

The challenge was fourfold: Participants were asked to find the bugs, fix them, explain the fix to all stakeholders, and show how they used any techniques or tools for finding and repairing the bug.

The three bugs were the following:

- Exchanged +/− signs in a calculation.
- An erroneous configuration, i. e., two parameters were set in an incompatible way.
- Erroneous use of a library function, or rather a library function that behaves differently from what would be expected.

3 Submissions

We received five submissions. Table 1 shows an overview. The most successful of them (A) was based on classical program analysis techniques such as abstract interpretation, tracing, and slicing. The participant's description showed that he gained a quite good understanding of the code, and that these techniques helped him to focus his attention on the relevant parts of the code. However, using these techniques and using the tool (Frama-C[2]) requires a good amount of expert knowledge.

Participant B used a commercial program analysis tool[3] for extracting projections of the control flow graph. However, he interpreted the tool's output in a

[1] The challenge description and material can be found at `http://icpc2011.cs.usask.ca/conf_site/IndustrialTrack.html`

[2] `http://frama-c.com/`

[3] `http://www.sgvsarc.com/Prods/CREVS/Crystal_REVS.htm`

	A	B	C	D	E
	Frama-C value analysis, abstract interpretation, tracing, slicing	Crystal REVS for C++ flow graphs, plots of trace	Diff of test drivers and configuration, plots	Ochiai bug localization, delta debugging	Test case generation, Ochiai bug localization
Bug #1	Fixed fault	Fixed symptom	Fixed fault	Fixed fault	Inelegant fix, fault still there
Bug #2	Fixed fault	Fixed fault	Fixed fault & superfluous	Fixed fault	"Fix" introduces new faults
Bug #3	Provided workaround	Fixed symptom	Provided workaround	Fixed Symptom	Fixed symptom & superfluous fix
Understanding	Good	Very limited	Manual	None	None

Table 1: Overview of submissions and results.

wrong way: It did not really show data flows (what they thought), but only reduced the control flow graph to nodes that contain a certain variable. This lead to an invalid reduction of the search space, and subsequently, the participant did not find the real underlying fault for two of the bugs. Such a tool could have been useful for understanding the code, but it would have required a better understanding of the capabilities of the tool and of the meaning of its results.

The third submission (C) was based on a quite simple technique and resulted in astonishingly good results. This participant used a simple differencing technique – in fact, a variant of delta debugging. He first checks for differences in the code and configuration of a faulty version compared to a working version, then removes these differences step by step. At some point, the error disappears, therefore the cause has to do with this last removed difference. In a subsequent step, the surroundings of this difference (e. g., the places where the differing parameter is used) are checked manually in detail. This is the key to success: He does not stop when the working replacement has been found, but rather starts detailed program comprehension at this point. The technique delivers some good starting points for that and thus effectively reduces the search space.

Submissions D and E both used the spectrum-based bug localization technique Ochiai [2, 3]. This technique analyzes the execution profiles of positive and negative test cases and then identifies those parts of the program that are primarily executed by failed test cases. Participant D successfully used this technique to locate one of the faults. For the other two bugs, he used delta debugging [5]. This technique was successful in one case, but not successful in the other case. The author of submission E stated that he also used the Ochiai technique. However, he was not successful and provided bug fixes that either did not fix the bugs or introduced new ones. Unfortunately, he did not provide a description of how exactly he proceeded to locate the faults. This shows that the success of using fault localization techniques depends on the people who use it.

Overall, the results are quite diverse. A lot seems to depend on the user: He has to know enough about the used techniques in order to apply them in a successful way. Also, those participants who gained a good understanding of the program – even when it was reduced to the relevant parts – provided the best bug fixes. Others, who did not invest much into program understanding, often fixed only symptoms or even introduced additional errors.

4 Conclusion

The different submissions showed that developers cannot successfully fix bugs when they have not understood the code. From the technical side, (abstract) interpretation, tracing, slicing, and differencing provided the most helpful information, which provides guidance for future program understanding research. Also, some bug localization techniques were successfully applied – but there need to be guidelines and heuristics for how to use these techniques. Blind adoption of advanced techniques is not a good idea and can lead to disastrous results. A certain amount of program understanding is inevitable to assure the correctness of a fix.

References

[1] A. Begel and J. Quante. Industrial program comprehension challenge 2011: Archeology and anthropology of embedded control systems. pages 227–229, 2011.

[2] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proc. of ESEC/FSE 97, LNCS, Vol. 1301*, pages 432–449, 1997.

[3] A. J. v. G. Rui Abreu, Peter Zoeteweij. On the accuracy of spectrum-based fault localization. In *Proc. of Testing: Academic and Industrial Conference – Practice and Research Techniques*, pages 89–98, 2007.

[4] V. Schulte-Coerne, A. Thums, and J. Quante. Challenges in reengineering automotive software. In *Proc. of 13th CSMR*, pages 315–316, 2009.

[5] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

Test der semantischen Äquivalenz von Translatoren am Beispiel von CoJaC

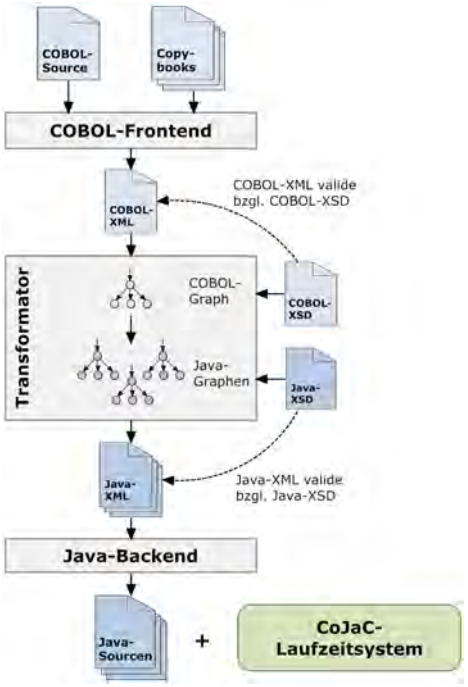
Christian Becker, Uwe Kaiser
pro et con Innovative Informatikanwendungen GmbH, Dittesstraße 15, 09126 Chemnitz
christian.becker@proetcon.de,uwe.kaiser@proetcon.de

Abstract

Im Rahmen des SOAMIG-Projektes[1] wurde u.a. der Translator CoJaC (COBOL to Java Converter) entwickelt. Entwicklungsziele von CoJaC waren, performanten und zum COBOL-Code semantisch äquivalenten Java-Code zu generieren. Der vorliegende Beitrag beschreibt die Testmethodik zum Nachweis dieser semantischen Äquivalenz.

1 CoJaC - COBOL to Java Converter

CoJaC konvertiert ein vollständiges COBOL-Programm (Hauptprogramm und Copybooks) in ein semantisch äquivalentes Java-Programm. CoJaC besteht aus einer Menge unabhängiger Werkzeuge, welche ihre Ergebnisse über Schnittstellen austauschen. Eine detaillierte Beschreibung von CoJaC liefert [1]. Eine für die Beschreibung der Testmethodik vereinfachte Darstellung der Architektur zeigt die nachfolgende Abbildung:



Ein COBOL-Frontend realisiert die aus dem Compilerbau bekannten Funktionen des Präprozessors, des Scanners und des Parsers. Der entstehende COBOL-Syntaxgraph wird in eine XML-Repräsentation serialisiert. Die XML-Schnittstelle entspricht einem definierten XSD-Schema (COBOL-XSD). Diese Repräsentation wurde gewählt, um die Interoperabilität verschiedener Migrationswerkzeuge der Firma pro et con zu gewährleisten. Das generierte COBOL-XML fungiert als Eingabe für ein weiteres Werkzeug (Transformator), welches auf Basis einer vorgegebenen Abbildungsbeschreibung eine *model-to-model*-Transformation aus dem COBOL-Syntaxgraphen in einen oder mehrere Java-Syntaxgraphen ausführt. Dieser Java-Syntaxgraph wird in Form einer serialisierten XML-Datei (Java-XML) dem Java-Backend zugeführt, welches aus dem Java-Generator und dem Java-Formatierer besteht. Ergebnis des Konvertierungsprozesses ist ein Java-Programm, das in seiner Funktionalität dem originalen COBOL-Programm entsprechen sollte. Zur Ausführung nutzen die generierten Java-Programme ein Laufzeitsystem. Dieses stellt Bibliotheksfunktionen für COBOL-Anweisungen und -Daten bereit, für die es in Java keine Entsprechung gibt.

2 Testmethode und Testabdeckung

COBOL besitzt die Eigenschaft, dass zu jeder syntaktischen Einheit (Anweisung, Datendefinition, ...) mehrere, zum Teil optionale Klauseln angegeben werden können. Diese sind in verschiedensten Kombinationen in kommerziellen Programmen enhalten. Deshalb wurde für jede mögliche Kombination von Anweisungen und Klauseln ein minimales Testprogramm (im Weiteren *Testfall*) erstellt, welches nur eine Variante der Anweisung sowie dafür notwendige Datendefinitionen enthält. Die systematische Spezifikation dieser Testfälle erfolgte anhand einer COBOL-Grammatik, welche dem aktuellen ANSI-85-Standard entspricht. Das folgende Listing enthält eine abgeschlossene Regel aus der COBOL-Grammatik für eine DISPLAY-Anweisung:

```
stmt   = DISPLAY id_lit {id_lit} upon;
id_lit= ident | literal;
upon   = [UPON name | env] [NO ADVANCING];
```

Die alternativen Klauseln (UPON, NO ADVANCING, ...) sind ersichtlich. Aus dieser Regel lassen sich u.a. die folgenden Testfälle systematisch ableiten:

- *DISPLAY ident*
- *DISPLAY literal*
- *DISPLAY ident UPON name*
- ...

Die Bezeichner der Grammatik (z.B. *ident*) stehen für einen COBOL-Bezeichner mit beliebigem Datentyp. Das heißt, dass die Anweisung *DISPLAY ident* in Kombination mit numerischen und alphanumerischen Datenelementen und Literalen getestet werden muss. Daraus ergibt sich eine große Anzahl von Testfällen. Die Gesamtheit aller Testfälle wird als Testabdeckung definiert. Diese kann jedoch aufgrund der Vielfalt der möglichen Kombinationen nie 100 % betragen. Im Verlauf der Entwicklung von CoJaC entstand auf diese Weise eine Testsuite mit aktuell mehr als 1.200 Testfällen. Bei neuen Projekten werden ausgewählte, kommerzielle Quellen auf ein Grundgerüst reduziert und ebenfalls konvertiert. Ziel ist, die Abdeckung

eines breiten Spektrums realer Problemstellungen aus historisch gewachsenen Programmen. Die Testsuite wächst folglich ständig mit der Nutzung von CoJaC, da vor allem in kommerziellen Programmen Spezialfälle auftreten, welche bis dahin noch nicht durch einen Testfall abgedeckt waren. Zur Abrundung der Testabdeckung wurden zusätzlich im Internet frei verfügbare, willkürlich ausgewählte COBOL-Programme verwendet.

3 Testrealisierung

Die Realisierung der Tests erfolgt in vier Phasen:
Compilierung und Ausführung der Testfälle in einer COBOL Enterprise Umgebung: Dadurch wird die syntaktische Korrektheit und Ausführbarkeit der Testfälle garantiert. Nur compilierbarer COBOL-Code soll von CoJaC konvertiert werden. Von jedem Testfall werden Dumps (z.B. Bildschirmausgaben) in einem Repository abgelegt.
Konvertierung der Testfälle von COBOL nach Java: In dieser Phase erfolgt die Verifikation der Funktionsfähigkeit des Translators sowie die Korrektheit der Schnittstellen zwischen den Werkzeugen. Der erzeugte Java-Code wird gegen die Konventionen der Abbildungsbeschreibung geprüft.
Ausführung der generierten Java-Programme: Alle generierten Java-Programme werden automatisiert compiliert und in einer Java-Umgebung ausgeführt. Dadurch erfolgt die Verifikation der Funktionalitäten des Laufzeitsystems. Zu jedem Testfall werden wiederum Dumps (z.B. Bildschirmausgaben) in Repositories abgelegt.
Vergleich der Repositories: In der letzten Testphase findet ein Vergleich der COBOL- und Java-Repositories eines Testfalls statt. Auf diese Weise werden semantische Fehler aufgedeckt. Sind die Dumps einer Anweisung in COBOL und in Java identisch, dann war auch die Transformation semantisch äquivalent. Nur in dieser Testphase kann die semantisch äquivalente Transformation von COBOL nach Java für einen Testfall nachgewiesen werden.
Jede Phase wurde über Batch-Prozesse automatisiert und ist somit beliebig oft wiederholbar.

4 Ergebnis des Testprozesses

Die Durchführung der Tests liefert eine Menge von Resultaten, welche zum Lokalisieren von Fehlern im Konvertierungsprozess und im Laufzeitsystem dienen. Die Fehler werden den einzelnen Translatorkomponenten zugeordnet und iterativ beseitigt.
Syntaktische Fehler in den COBOL-Programmen: Syntaktische Fehler treten auf, wenn ein Codefragment nicht vom COBOL-Frontend verarbeitet werden kann. Dieses wird seit Jahren stetig weiterentwickelt und wurde bereits an einer Vielzahl von kommerziellen Quellen ausgetestet. Daher sind echte syntaktische Fehler eher selten. Das Frontend arbeitet restriktiver als der Compiler, wodurch es zusätzlich die Sanierungsphase von COBOL-Programmen vor einer Software-Migration unterstützt.
Syntaktische Fehler in den Java-Programmen: Die erzeugten Java-Programme werden mit Hilfe eines Java-Compilers auf syntaktische Fehler untersucht. Fehler in der *model-to-model*-Transformation des Transformators oder der Codegenerierung des Java-Backends können auf diese Weise erkannt werden.
Korrektheit der COBOL- und Java-Syntaxbäume: Die vom COBOL-Frontend und dem Transformator gelieferten COBOL- bzw. Java-Syntaxbäume (COBOL- bzw. Java-XML) werden nach dem jeweiligen Verarbeitungsschritt gegen ein entsprechendes Schema (COBOL-XSD bzw. Java-XSD) validiert. Dies dient zur zusätzlichen Überprüfung der korrekten Verschachtelung von Anweisungen und Operatoren sowie zur Sicherstellung der fehlerfreien Verarbeitung in den nachfolgenden Transformationsprozessen.
Semantische Äquivalenz: Semantische Unterschiede in den COBOL- und Java-Programmen werden durch den Vergleich der Ergebnis-Repositories aufgedeckt. Erst wenn die Ausgaben der COBOL- und Java-Programme identisch sind, kann der Code als semantisch äquivalent betrachtet werden. Kann eine Anweisung von CoJaC nicht transformiert werden, dann ist das Programm (aus Sicht der gesamten Programmlogik) nicht semantisch äquivalent. Um solche Fälle zu erkennen, werden im Transformationsprozess so genannte „ToDo()“-Funktionsaufrufe für nicht übersetzbare Anweisungen erzeugt. Ein solches „ToDo“ führt zur Laufzeit zu einer Exception. In diesem Fall muss die Anweisung manuell konvertiert werden.
Laufzeitfehler: COBOL-Compiler tolerieren fehlerhafte Konstruktionen, die nicht unmittelbar zu einem Programmabbruch führen. Z.B. kann einer numerischen Variablen in COBOL eine Zeichenkette zugewiesen werden. In Java wird in diesem Fall durch das Laufzeitsystem eine Exception geworfen, da die Zuweisung eindeutig unkorrekt ist. Das Laufzeitsystem arbeitet somit sehr restriktiv.

5 Zusammenfassung

Trotz der umfangreichen Tests kann nicht gezeigt werden, dass für den gesamten COBOL-Sprachumfang semantisch äquivalenter Java-Code erzeugt wird. Dies gilt nur für jene Testfälle, welche den vollständigen Testprozess erfolgreich bestehen. Dazu gehören die Tests aus der Testsuite, die verwendeten kommerziellen Kundenlösungen und die willkürlich ausgewählten COBOL-Programme aus dem Internet. Der durch diese Kombination erreichte Querschnitt des COBOL-Sprachumfangs war für den bisherigen, kommerziellen Einsatz von CoJaC ausreichend.

Literaturverzeichnis

[1] Erdmenger, U.; Uhlig, D.:
Ein Translator für die COBOL-Java-Migration.
13. Workshop Software-Reengineering (WSR 2011),
2.-4. Mai 2011, Bad Honnef.
In: GI-Softwaretechnik-Trends, Band 31, Heft 2,
ISSN 0720-8928, S. 73-74

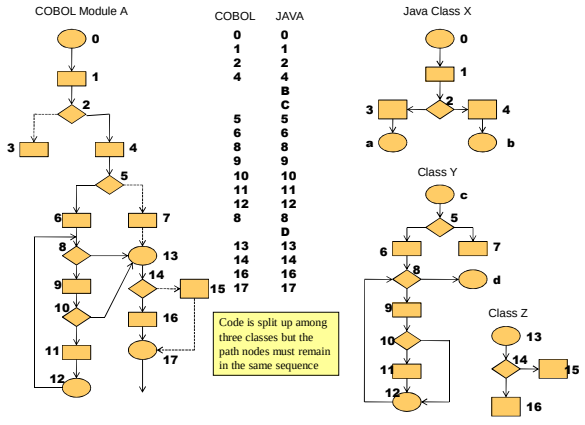
[1] Das vom Bundesministerium für Bildung und Forschung geförderte Projekt SOAMIG beschäftigte sich mit der Migration von Legacy-Software in serviceorientierte Architekturen.

Validierung der funktionalen Äquivalenz konvertierter JAVA Programme durch einen dynamischen Source-Abgleich

Harry M. Sneed
ANECON GmbH, Wien

1. Die zugrunde liegende Theorie

Die Theorie, die diesem Testansatz zugrunde liegt besagt, unabhängig davon, wie die untersten Codebausteine umstrukturiert werden und statisch gegliedert sind, müssen sie dynamisch immer in der gleichen Reihenfolge ausgeführt werden, um funktional äquivalent zu sein. D.h. die COBOL Paragraphen können zerlegt und als Methoden auf diverse Java Klassen verteilt werden. Dennoch müssen die Java Methoden exakt in die gleichen Reihenfolge ausgeführt werden wie die ursprünglichen COBOL Paragraphen bzw. elementare Operationen. Darüber hinaus müssen auch die einzelnen Anweisungen in der gleichen Reihenfolge vorkommen. Es können zwar zusätzliche Methoden und Anweisungen dazwischen kommen, aber die aus dem COBOL Code übernommene Java Methoden müssen in der gleichen Folge vorkommen wie die alten COBOL Bausteine. Die Bedingungen müssen in der gleichen Weise erfüllt oder nicht erfüllt werden und die Schleifen müssen dieselbe Anzahl Wiederholungen haben. (*siehe Abb. 1: Reihung der elementaren Operationen)*



Jegliche Abweichung in der Ausführungsfolge der Grundbausteine deutet auf einen Fehler in der Transformation hin. Demzufolge muss versucht werden, die beiden Komponente - das COBOL Programm und das Java Paket – parallel zu einander ohne Daten auszuführen und die Ablaufpfade miteinander zu vergleichen. Die Methode hierfür heißt symbolische Analyse und geht zurück auf die Forschung von W. Howden in den 70-er Jahren. Danach wird das Programm mit künstlichen Daten ausgeführt. Der Tester steuert den Ablauf durch den Code in dem er bei allen Verzweigungen bestimmt, wie es weitergeht - z.B. wenn er zu einer Alternativanweisung kommt, bestimmt der Tester ob die Bedingung erfüllt oder nicht e4rfüllt wird und wenn es zu einer Schleife kommt, bestimmt er, wann die Schleife beendet wird. Auf dieser Weise werden die unterschiedlichen Pfade durch den Code aufgezeichnet. In der ursprünglichen Forschung von Howden wurden diese Pfade mit dem spezifizierten Pfade abgeglichen um die Korrektheit der Programmlogik zu verifizieren.

2. Aufbau der Anweisungstabellen

Bei der Verifikation konvertierter Komponente wird nicht mit einer Spezifikation, sondern mit den Ablaufpfaden des ursprünglichen Codes verglichen. Der Tester prüft die symbolische Ausführung der beiden Codeversionen – von der COBOL Version wie auch von der Java Version – eine interne Anweisungstabelle generiert. Die Anweisungen sind darin mit ihrer Zeilennummern gekennzeichnet. Neben die Zeilennummer der jeweiligen Anweisung deutet eine zweite Anweisungsnummer auf die folgende Anweisung. Im Falle einer Alternativanweisung gibt es zwei potentielle Nachfolgeranweisungen. Im Falle einer Schleife gibt es die erste Anweisung in der Schleife und die erste Anweisung hinter der Schleife. Im Falle einer Fallanweisung kann es *n* potentielle Nachfolgeranweisungen geben.
Eine dritte Zeilennummer verbindet die Anweisung in dieser Version mit der entsprechenden Anweisung in der anderen. In der Java Tabelle wird auf die COBOL Zeile oder Zeilen hingewiesen, aus der die Java Anweisung stammt. In der COBOL Tabelle wird umgekehrt auf die Java Zeile bzw. Zeilen hingewiesen, die aus dieser COBOL Anweisung abgeleitet wurden. So sind die beiden Tabellen rückwärts und vorwärts miteinander gekettet. (siehe Sample 1: Statement Table)

Neben den zwei Anweisungstabellen wird eine Aufrufstabelle erzeugt, in der in COBOL alle PERFORM und CALL Aufrufe und in JAVA alle Methodenaufrufe eingetragen sind. Sollte in Java die gleiche Methode in mehreren Klassen vorkommen – polymorphy – kann der Tester die gewünschte Klasse auswählen. (Siehe Sample2: Method Table)

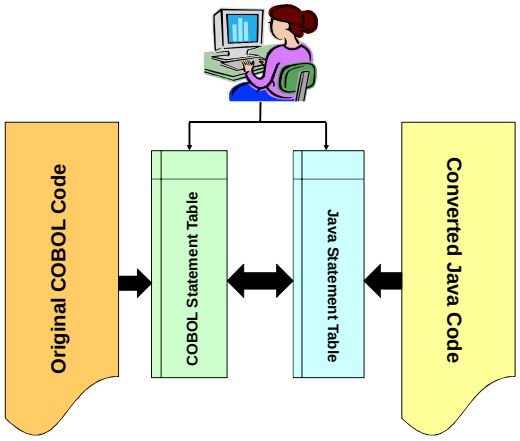
Außer den zwei Anweisungstabellen werden auch zwei Datentabellen generiert, in denen alle globalen und lokalen Variabel in COBOL wie auch in Java registriert sind. Zu jeder Variabel kommt eine Liste der Zeilennummer der Anweisungen, in denen diese Variabel referenziert wird, zusammen mit einem

Kennzeichen für die Art der Verwendung – als Argument, Ergebnis oder als Prädikat bzw. Bedingungsoperanden. Diese Referenzen werden bei der Ausführung der symbolischen Analyse benutzt im Datenflusspfade zu bilden. (seine Sample 3: Data Table)

3. Ausführung der symbolische Analyse

Nach dem Aufbau der Anweisungs- und Datentabellen beginnt der Tester mit der Nach dem Aufbau der Anweisungs- und Datentabellen beginnt der Tester mit der symbolischen Analyse. Als erstes steuert er ausgewählte Pfade durch den Java Code hindurch. Dabei wird ein Trace-Protokoll der durchlaufenen Java Anweisungen sowie ein Trace-Protokoll der referenzierten Java Daten erstellt. Das erste Protokoll dokumentiert den Steuerungsfluss, das zweite Dokument den Datenfluss.

Anschließend wiederholt der Tester die Ausführung der gleichen Pfade durch den COBOL Code. Dabei werden die gleichen zwei Protokolle für die COBOL Version erstellt. (siehe Abb. 2: Symbolische Analyse der beiden Source-Versionen)



Zum Schluss werden die Java Trace-Protokolle mit dem COBOL Trace-Protokollen automatisch abgeglichen. Die Anweisungen sollten in beiden Fällen in der gleichen Reihenfolge kommen. Wenn nicht, oder wenn eine Anweisung fehlt, erfolgt eine Fehlermeldung. Bei den Daten-Trace-Protokollen wird geprüft, ob die Datenreferenzen in der gleichen Reihenfolge erscheinen und ob sie von der gleichen Verwendungsart sind. Wenn nicht, erfolgt auch hier eine Fehlermeldung. Es werden auf diese Weise zweierlei Fehlerarten in der Konversion aufgezeigt:

- Fehler bei der Ablaufsteuerung und
- Fehler beim Datenfluss.

4. Erfahrung und Weiterführung

Dieser Testansatz den Vorteil, dass er die Auflösung des Programmcodes simuliert, ohne den Overhead einer aufwendigen Testumgebung – und das bei einem relativ geringen Aufwand. Der einzige manuelle Aufwand ist die Steuerung der Ablaufpfade durch den Tester, bzw. der Reengineer. Dies hat aber auch ein nützliches Seiteneffekt. Es zwingt den Reengineer sich mit dem Code auseinander zu setzen. dadurch stößt er unweigerlich auf weitere Anomalien im Programmverhalten.

Die Fehler die hier auftauchen, sind fasst alle auf Fehler in dem Konversionswerkzeug zurückzuführen. Meistens sind es ungewöhnliche Konstrukte in COBOL, wie der SEARCH Befehl, die vom Werkzeug nicht richtig behandelt wurden. Wenn diese Konstrukte mehrmals vorkommen, muss die gesamte Codetransformation wiederholt werden. Wenn sie nur in einem oder zwei Sourcen vorkommen, ist es besser sie in dem Java Code manuell nachzubessern.

Der Test des konvertierten Systems bleibt weiterhin der Hauptkostentreiber in allen Migrationsprojekten. Alles, was dazu beitragen kann, den Testaufwand zu reduzieren, ist vom Nutzen. Dieses Verfahren verspricht mindestens einen Teil der Fehler zu entfernen, eher es zum großen Regressionstest kommt. Damit wird der System entlastet, da der Aufwand, den vielen gewöhnlichen Umsetzungsfehlern nachzugehen, fällt weg. Es bleibt genug zu tun mit den Kompatibilitätsfehlern.

Literaturhinweise:

[1] Sneed, H., Wolf, E., Heilmann, H.: “Software Migration in der Praxis”, dpunkt.verlag, Heidelberg, 2010.
[2] Sneed, H.: “ Migrating PL/I Code to Java”, IEEE Proc. of 15th CSMR, IEEE Computer Society Press, Oldenburg, p. 287
[3] Broy, M.: “Zur Spezifikation von Programmen für die Textverarbeitung”, in Wossido, P. (ed.): Textverarbeitung und Informatik, Informatik Fachberichte 30, Springer Verlag, Heidelberg, 1980, p. 75
[4] Howden, W.: "Symbolic Testing with the DISSECT Symbolic Evaluation System" in IEEE Trans. on S.E. Vol. 1, No. 4, July 1977, p. 266
[5] Parrish,A./Borie,R./Cordes,D.: “Automated Flow Graph-based Testing of Object-oriented Software Modules” Journal of Systems and Software, Vol. 23, No. 1,1993, pp. 95-109
[6] Sneed, H.: “Validating Functional Equivalence of reengineered Programs via Control Path, Result and Data Flow Comparison”, Software Testing, Verification, & Reliability, Vol. 4, No. 1, March 1994, p. 33
[7] Howden, W.: "Reliability of the Path Analysis Testing Strategy”, IEEE Trans. on S.E. Vol. 1, No. 4, Sept. 1976, p. 208
[8] Peters, D., Parnas, D.: “Using Test Oracles generated from Program Documentation”, IEEE trans. On S.E., Vol. 24, No. 3, March 1998, p. 161
[9] Sneed, H.: “Source Animation as a means of Program Comprehension for object-oriented Systems” Proc. of 8th IEEE International Workshop on Program Comprehension, Computer Society Press, Limerick, June 2000, p. 179
[10] Cornelissen, B., Zaidman, A., van Deursen, A.: „A controlled Experiment for Program Comprehension through Trace Visualization“, IEEE Trans. on S.E., Vol. 37, No. 3, p. 341

Systemtest im agilen Entwicklungsprozess

Uwe Hehn, Sebastian Kern
Method Park Software AG
Wetterkreuz 19a, 91058 Erlangen, Germany
Email: {uwe.hehn, sebastian.kern}@methodpark.de

Abstract: *In der Vergangenheit wurden medizintechnische Projekte typischerweise klassisch gemäß V-Modell durchgeführt.*
Wir untersuchen, wie die klassische Vorgehensweise mit agilen Elementen kombiniert werden kann, so dass man die Flexibilität des agilen Ansatzes nutzen kann und gleichzeitig die strengen Anforderungen an die Abnahme von Medizinprodukten erfüllt werden können.
Das vorgestellte Konzept zum „agilen Systemtest für medizintechnische Produkte“ beruht auf unseren Erfahrungen in einem großen Medizintechnik-Unternehmen.

1 Einleitung

Bei der Zulassung medizintechnischer Geräte wird nicht nur das Produkt sondern auch der Prozess geprüft; das betrifft insbesondere das Vorliegen von Entwicklungsdokumenten in hoher Qualität. Viele Unternehmen kombinieren die klassische Vorgehensweise gemäß V-Modell mit agilen Elementen. Man hofft, dadurch sowohl die strengen Anforderungen an die Abnahme von Medizinprodukten erfüllen zu können als auch die Flexibilität nutzen zu können, die der agile Ansatz verspricht.
Die folgenden Überlegungen zur Zulässigkeit eines agilen Systemtest für medizintechnische Produkte gründen sich auf aktuelle Erfahrungen in einem großen Medizintechnik-Unternehmen, bei dem ein auf SCRUM [1] basierender agiler Systemtest zur Anwendung kam.

2 Systemtest nach dem V-Modell

Wir betrachten im Folgenden den Fall System = Softwaresystem. (Im Falle allgemeiner Systeme überträgen sich die Überlegungen auf den Systemtest der Software-Subsysteme.)

Der Systemtest dient dazu, die Erfüllung der Systemanforderungen zu prüfen [3].

Die Norm ISO15504-5, ENG.10 (SPICE) [2] definiert genauer:

- *Der Zweck des Systemtestprozesses ist die Sicherstellung, dass die Implementierung aller Systemanforderungen auf deren Erfüllung getestet wird, und dass das System zur Auslieferung bereit ist.*

Allgemeine Grundlagen der Qualitätssicherung fordern, dass das Systemtest-Team unabhängig von der Entwicklung sein muss (Vier-Augen-Prinzip).

Der Zeitbedarf für den Systemtest ist typischerweise sehr groß. Geschickterweise kann man Aktivitäten wie z.B. den Testentwurf, den Aufbau einer Testumgebung bzw. die Realisierung einer Testautomatisierung schon während der Entwicklung des Systems umsetzen. Die Durchführung des Systemtests ist erst dann möglich, wenn das System vollständig ist (Abbildung 1). Folglich ist in der Praxis die Zeit für den Systemtest oft sehr knapp.

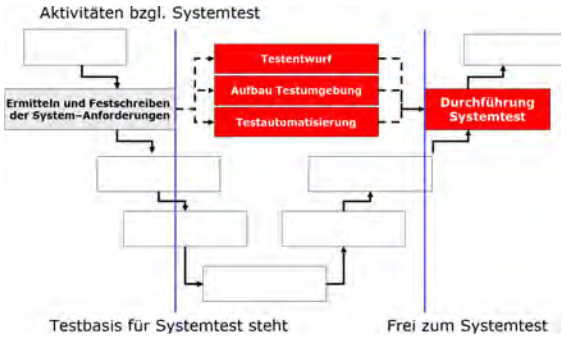


Abbildung 1: Aufteilung der Aktivitäten des Systemtests in Phasen

Der Einsatz agiler Methoden sollte helfen, dass die Vorbereitung der Durchführung des Systemtests möglichst früh erfolgt. Außerdem erhofft man sich, dass das zeitliche Bottleneck der „Durchführung des Systemtests“ verringert wird.

3 SCRUM und Systemtest

3.1 Fachliche Anforderungen an den Systemtest

Selbstverständlich können die folgenden fachlichen Aspekte des Systemtests nicht außer Acht gelassen werden:

- Funktioniert die Software?
- Testspezifikation (Was wurde in welchem Umfang getestet?)
- Testergebnisse (Ergebnis der Prüfung)
- Reagieren auf geänderte Anforderungen
- Regressionstest-Strategie (Veränderungen ohne negativen Einfluss?)

3.2 Die Kombination SCRUM und Systemtest

Die von uns vorgeschlagene und beim Kunden eingeführte Vorgehensweise sieht vor, den Systemtest anteilig bereits in den N Entwicklungssprints zu berücksichtigen. In den abschließenden Systemtest-Sprint N+1 wird dafür im Gegenzug auch das Beheben von in den Systemtest-Sprints gefundenen Fehlern eingeplant. Gefundene Fehler werden so bald wie möglich zur Behebung eingeplant (Abbildung 2). Im Idealfall sind im Systemtest-Sprint im zu testenden System nur noch sehr wenige Fehler vorhanden.

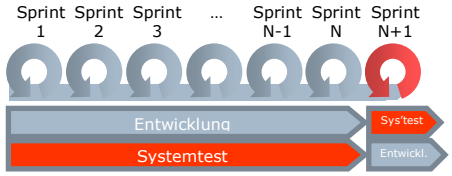


Abbildung 2: Smarte Aufteilung der Systemtest-Aktivitäten auf die Folge von Sprints

Wie eine effektive Umsetzung im Detail aussieht, wird im folgenden Kapitel erläutert.

4 Einbindung von Systemtests in SCRUM

In der Makrosicht (Roadmap) werden N+2 Sprints vorgesehen. Wir betrachten im Folgenden die Sprints aus dem Blickwinkel des Systemtests; natürlich werden in diesen Sprints außerdem die Entwicklung sowie die Aufgaben der anderen Teststufen angesiedelt sein.

4.1 Überblick über die Aufteilung der Sprints aus Sicht des Systemtests

Der initiale Sprint 0 dient dazu, alle Vorbereitungen zu treffen, damit der Systemtest in den folgenden N Produktiv-Sprints effektiv durchgeführt werden kann (Systemtest-Initialisierung).

Die folgenden N Sprints 1-N (Produktiv-Sprints) liefern als Ergebnis ein bereits lauffähiges und getestetes Teilsystem (potentially shippable system). Neben der (Weiter-)Entwicklung des Produkts im Sprint X findet jeweils mindestens ein Regressionstest statt (Abdeckung der Anforderungen der Sprints 1-(X-1)). Schließlich erfolgt der Systemtestentwurf für die im aktuellen Sprint X neu umgesetzten Anforderungen.

Das Besondere an den Produktiv-Sprints ist, dass einerseits der Systemtest aller bis zum Vorgänger-Sprint umgesetzten Anforderungen erfolgt, andererseits die Testvorbereitung für den aktuellen Sprint durchgeführt wird.

Durch diese enge Verzahnung der Aktivitäten des Systemtests mit der Folge der Sprints ergibt sich eine (potentiell) optimale Zuordnung der Systemtestaktivitäten.

Im Abschluss-, Cleanup- oder QS-Sprint (N+1) findet der abschließende Systemtest statt (Testen aller bis inkl. Sprint N umgesetzten Anforderungen).

5 Herausforderungen

Eine zentrale Herausforderung ist die Automatisierung des Systemtests, da nur dadurch genügend Zeit für die verbleibenden manuellen Aktivitäten übrig bleibt.

Die vollständige Integration eines vormals unabhängigen Systemtest-Teams in den Entwicklungsprozess ist eine „kulturelle Herausforderung“. Die notwendige Unabhängigkeit kann jedoch auch hier organisatorisch gewahrt bleiben.

Referenzen

[1] Roman Pichler: SCRUM, dpunkt.verlag, 1. Auflage 2008, Heidelberg

[2] ISO/IEC 15504-5, International standard, Part 5: An exemplar Process Assessment Model, San Francisco, First edition, 2006-03-01, Switzerland

[3] ISTQB: Foundation Level Syllabus (2011), www.istqb.org

Fuzzing: Testing Security in Maintenance Projects

Frank Simon, Daniel Simon
SQS Software Quality Systems AG, Stollwerckstraße 11, 51149 Cologne, Germany
Email: frank.simon|daniel.simon@sqs.com

Abstract: New trends in IT industry impose increasingly requirements on openness and interoperability via networks to enterprise software systems. As a consequence, more and more legacy applications are made available via interfaces more openly through mobile and insecure networks, thereby inducing security risks the initial designs have never had to account for. In this paper, we show how a highly automatable black-box method called *fuzzing* for testing security can be integrated into testing processes to increase interfaces of legacy application in terms of security profiles.

1 Introduction

Several IT technology trends (SOA, cloud computing, and the ever present mobility) contribute consciously to increasing networking readiness of applications. Following the new trends, everyone is expecting legacy applications to be accessible via mobile and internet connections rather than closed and secure enterprise intranets. At the same time, more and more business critical data are made available through such channels and as such increase business risks with regards to security. Long-living software systems and application never designed for the access via unsecured and open networks now have to be made ready to interact and interoperate through channels unforeseen at the time of initial development. To this end, we elaborate in this paper how the widely-known ideas of *fuzzing* can be integrated into standard test processes in order to increase the security of legacy applications that have undergone extension by many new interfaces.

2 Testing and Security

According to ISTQB [1], testing is *"The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects."*

Testing is not only mandatory for new IT systems. Whenever a change to a system is implemented (e.g. by adding interfaces) it is the tester's task to assure "correctness" of the implementation of the change. This includes regression testing to demonstrate unchanged requirements' stability as well as testing new requirements. New requirements can induce adjustments for the regression testing as well: The requirement to open an existing IT system for mobile communication – as example – has not only to be tested for its own but might motivate deeper testing of directly connected components.

For a more systematic view on these implicit testing adjustments testing can be refined into four steps (a more general approach can be found in [2]:

1. Identification of test objects (What artefacts relevant for project success?)
2. Identification of quality attributes (What properties should the artefacts have?)
3. Determination of corresponding test activities to ensure artefacts having particular attributes
4. Clustering of test activities into test stages that can be executed in conjunction

This paper focuses the following aspect: Adding new interfaces creates new test objects as well as it produces new or at least adjusted priorities for quality attributes requiring additional test activities on all test stages. Quality attributes for software can be taken from ISO 25000 family of standards. [3] In particular when adding new service interfaces to legacy applications the first time, security should be seen as one of the top priorities. Security is defined in the ISO 25010 standard as the *Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.*

3 Fuzzing of Software Interfaces

Fuzzing was developed at the University of Wisconsin in 1989 [4], [5]. Takanen et al [6] define fuzzing as follows:

„A highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. The name comes from modem applications' tendency to fail due to random input caused by line noise on `fuzzy` telephone lines."

From the tester's perspective, fuzzing is

- a black-box test for interfaces as test objects, as it does not require knowledge of the underlying implementation;
- a test method for the quality attribute security, as it tries to identify errors in a system that compromise confidentiality, integrity, and availability;
- a negative test method, as it does not try to verify expected system behaviour.

- a brute force method, as it makes use of excessive, sometimes random, test data to exploit the interfaces;
- a boundary test, as it derives test data from specified valid input data and bombards the interfaces accordingly;
- an automated test method, as its use of mass data can only be deployed effectively driven by a machine.

As legacy systems are migrated into Service Oriented Architectures and open their interfaces to the "outside" world, in some cases through the provisioning of web service interfaces, fuzzing is gaining more and more relevance. System owners have started to realise the risks associated with widely (and sometimes through uncontrolled networks) accessible interfaces and the need to make those interfaces "bullet proof" also with regards to security aspects.

To date however, a systematic or holistic approach towards fuzzing has not been observed. Surprisingly, both modern practise oriented process models (e.g., Microsoft Development Lifecycle Model) mention as well as established standards such as ISTQB/ISEB [1] give little attendance to security tests and fuzzing so far.

To utilise fuzzing for legacy systems undergoing some enhancements by adding new interfaces the following can be stated as basis:

1. regression testing and testing the new requirements is done.
2. security as new attribute must be revisited and considered adequately as it might not have been in the legacy system's original setup and it has significant influence on a wide range of additional test objects as well.

In the following, a generic approach is drafted to integrate fuzzing into a generic test process covering a wide range of project types in a way to easily account for new security requirements.

4 Integration of fuzzing into the standard test process

In order to make use of fuzzing in software maintenance projects, we have to integrate the fuzzing methodology in the test process to link it with statement (1) mentioned above. However, in practice there exist several different standard test processes like ISTQB fundamental test process [7], TMap process [8], SCRUM [9], and ISO 29119 [10]). For a generic fuzzing integration these different processes were analysed and a generic test process for testing was derived. This meta process was designed as to define the integration points for fuzzing and make available the fuzzing methods for a wide range of software projects.

Traditional test processes applied for retesting legacy systems can be easily aligned with this generic process. The artefacts in red define the integration points of fuzzing.

The advantages of this approach are:

- Existing test processes remain unchanged (or even better: a generic process is modelled that can be reused).
- Hotspots for integrating fuzzing are highlighted.
- This process complements "traditional" retesting with security testing

One point is not solved by this: The overall awareness that security gets a more important quality attribute motivating to apply this process. This creation of awareness has to be done separately in advance of deploying an IT-system.

5 Summary and Outlook

With the first results, we have integrated *fuzzing* into generic test processes and thereby have made available a method for security testing for general use in software development and maintenance. Fuzzing has already proven its success in many projects, however without any opportunity to report about tangible figures. Future work lies in a scientific case study, collecting relevant numbers (effort, findings, etc.) to demonstrate the positive ROI of this overall concept.

6 Acknowledgements

Parts of this work have been sponsored by SesamBB: *Security and Safety made in Berlin-Brandenburg e.V.* The full result is available to SesamBB members.

7 References

[1] Tilo Linz Andreas Spillner, *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard*, 4th ed.: dPunkt Verlag, 2010.

[2] Frank Simon and Daniel Simon, *Qualitätsrisikomangement*. Berlin: Logos Verlag, 2010.

[3] ISO, DIN ISO/IEC 25000 Software-Engineering – Qualitätskriterien und Bewertung von Softwareprodukten (Software product Quality Requirements and Evaluation), 2010.

[4] B.P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM* , vol. 33, no. 12, December 1990.

[5] Wikipedia. (2011, August) http://de.wikipedia.org/wiki/Fuzzing

[6] Ari Takanen, Charles Miller, and Jared J. Demott, *Fuzzing for Software Security Testing and Quality Assurance*. Norwood: Artech House, 2008.

[7] ISTQB, "Standard glossary of terms used in Software Testing," 2007.

[8] L. van der Aalst, B. Broekman, M. V. T. Koomen, *TMap® Next - Ein praktischer Leitfaden für ergebnisorientiertes Softwaretesten*. Heidelberg: dpunkt, 2008.

[9] Wikipedia. (2011, Nov.) http://de.wikipedia.org/wiki/Scrum.

[10] (2011, Nov.) ISO/IEC 29119 Software Testing. http://softwaretestingstandard.org/

Code Museums as Functional Tests for Static Analyses

Daniel Speicher, Jan Nonnen, Andri Bremm
University of Bonn, Computer Science III, Bonn, Germany
{dsp, nonnen, bremm}@cs.uni-bonn.de

Growing your software guided by tests [2] has the benefit of thoroughly tested implementations of the right functionality. If you are developing static analyses your "test data" consist of *code* that has a few lines to a few classes. How can tests based on this "data" be kept expressive and maintainable? After exploring a variety of different other approaches we decided to embed the expectations into the data instead of embedding the data into test cases. Figure 1 presents an overview of our tool set. The functional tests are within Java code, the tested static analyses are implemented as logic programs.[1]

1 Code Museums

To keep test data and expecatations collocated and to leverage the assistance of the compiler to check the syntactical correctness, we collect the test data in compilable projects and add the expectations as annotations. We call these projects "museums". As the purpose of this code is not to be executed there is a subtle change in the meaning of annotations (1.1). These annotations serve as descriptions of the code (1.2), as functional tests of the static analyses (1.3), and as the basis for education and cultivation of developers and static analyses (1.3).

1.1 Labels in Life and in Museums

Labels added to things used in life are mainly used to guide the handling of the thing.[2] Once we put a thing into a museum the purpose of the thing changes: It is meant to be viewed and not to be used. A label next to the thing therefore does not guide the usage but describes it. Similar to labels on things in life annotations on code can describe how the code is executed or at least compiled[3]. The annotations which we add to our code in a museum are not meant to change the compilation or execution but to describe the code.

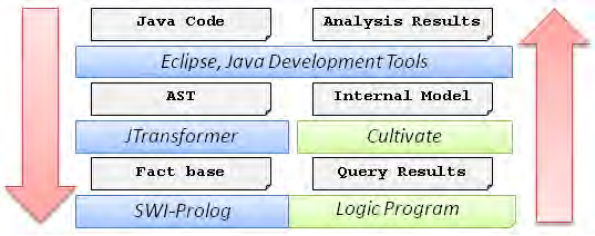


Figure 1: Data flow through the layers of our tools. Static analyses are implemented as logic programs in Cultivate.

1.2 Description of the Code

Figure 2 shows an example of a class in a code museum. Several annotations are added to the class and its methods. The annotation `@HasMetricValues` lists the values of three cohesion metrics[4] for this class. `@HasQualities` lists certain qualities of the methods. `@Expressed` lists concepts that the code expresses[5] In our example all method names start with "get" and may therefore be considered as "getter method". Still, a concept that is expressed in the code does not need to be fulfilled, therefore `@Fulfilled` lists the concepts that the method fulfills. The method `getCount` is clearly a getter method, while the method `getNextCount` is not.

1.3 Functional tests

The code together with the annotations is used as a functional test of the logic program. The logic program contains implementations of metrics and qualities. A test fails if the implementation does not produce the metric value or the quality that is annotated. Similarly the logic program contains a definition of when a program element is meant to express a certain concept and when it actually fulfills it. These definitions are tested by comparing them with the annotations of expressed and fulfilled concepts.

[1] http://sewiki.iai.uni-bonn.de/cultivate

[2] A label on a door may decide about whether women or man use it, or if we use the door to enter or to leave a building; a label on a bottle may decide whether we are willing to drink the content or will be even preventing any contact with our skin.

[3] E.g. `@Inject` identifying fields that are automatically initialized; `@SurpressWarning` surpressing compiler warnings.

[4] LCOM1 is the number of method pairs that do not access a common field. LCOM5 is the ratio of pairs of field and methods for which the method does not access the field. TCC is the ratio of pairs of public methods that do not (directly or via further methods) access a common field. [1]

[5] E.g. by following a naming convention, placing a type in a certain package, following a structure that is a known design pattern or implementation pattern.

1.4 Education and Cultivation

The annotations explain the code not only to the logic program but as well to developers or students. With annotations design pattern, implementation pattern or programming styles can be described.[6] Our example demonstrates an aspect of the calculation of cohesion metrics: Accesses to fields via trivial getter methods like `getCount()` should be considered equivalent to a direct field access. [7] Our example illustrates that basic concepts like "getter method" are not always as clear as say may seem. Judgement of design quality involves the understanding of similar concepts that appear to be fuzzy, once explored in detail. Examples are a good basis for such clarification. As functional tests for our logic program, a new failing example triggers the evolution of more refined judgement implemented in the logic program.

2 Experiences

While working with students we encourage a development style that is guided by tests. Indeed we did invest between 10% and 30% of time in the creation of different test frameworks. There had been successful steps like mastering plugin tests (start up eclipse, programmatically create Java projects, trigger compilation, execute the fact creation and the analyses), performance improvements by precompiling the fact bases, and finally a set of Hamcrest[8] matchers that allowed to write JUnit tests for the analyses fluently. Nevertheless because of the separation of "test data" and test expectations the creation of functional test cases never felt easy. The test cases per metric or smell barely covered all subtleties.[9] Once we introduced a first prototypical implementation of museums to our students the number of test cases created increased[10] Besides the pure number of test cases the most important benefit was the possibility to discuss the cases.

[6]There are annotations expressing that a program element plays a role in a design pattern or that there is a relation to another program element. References in the definition follow the rules of Java identifiers but ignoring modifiers that reduce visibility. Java does not allow to annotate statements (besides local variable declarations). Therefore we allow to add these annotations with a line comment next to the statement. The first token in a line comment is taken as identifier of the statement so that references to lines can be made. There are annotations that express that a program element has a "bad smell" (i.e. can be improved by refactoring) or that it does not.

[7]This explains the results of the cohesion metrics. `getCount()` is ignored. The two other methods access `logger` and are thus connected an LCOM1=0. Only `count` is not accessed by both remaining methods, so that LCOM5=1/4. Finally there is only one public method remaining, so that trivially TCC=1.

[8]http://code.google.com/p/hamcrest/

[9]Number of test cases between 1 and 5.

[10]Number of examples/counter examples: Overhaul of the implementation of the class version of the "Law of Demeter" 36/17; Android performance guidelines "avoid internal use of accessor methods" 7/7; "make private fields public if accesses from an inner class" 4/14; "declare constants static final" 2/14; "favor static methods of virtual methods" 30/32.

```
@HasMetricValues({ "lcom1=0", "lcom5=0.25", "tcc=1.0" })
public class CountGenerator {

    private int count = 0;
    private Logger logger;

    @Expressed({ "getter_method" })
    @Fulfilled({ "getter_method", "trivial_getter_method" })
    @HasQualities({ "returns_field", "side_effect_free" })
    public int getCount() {  return count;  }

    @Expressed({ "getter_method" })
    @Fulfilled({ "getter_method", "lazy_intializer" })
    @NotFulfilled({ "trivial_getter_method" })
    @HasQualities({ "returns_field" })
    protected Logger getLogger() {
        if (logger == null) { logger = new Logger(); }
        return logger;
    }

    @Expressed({ "getter_method" })
    @NotFulfilled({ "getter_method", "trivial_getter_method" })
    @HasQualities({ "returns_field" })
    public int getNextCount() {
        count += 10;
        getLogger().log("Next count is: " + count);
        return count;
    }
}
```

Figure 2: Code with annotations describing the code.

3 Limitations and Opportunities

As we integrate our expectations into the test data, these modifications may have an influence on the test result. Still, the distinction between annotated expectations and the main Java code are obvious to test developers, so possible interferences can be identified. The approach offers the opportunity to integreate the implementation smoothly into the design. To "test", whether a certain smell applies where it is expected, an analysis is implemented that verifies for each annotation of an expected smell that the implementation of the smell heuristic (another analysis) delivers the annotated element. Similarily consistency checks of the annotations, e.g. verifying the presence of the referenced analyis can be added.

4 Conclusion

Museums are collections of annotated code as "test cases" for static analyses that can be even used for educational purposes or as the basis to discuss judgments about code quality in the developer team. Developers may consult these collections of code examples to explore the coding culture and to refine their judgment. That is, well maintained museums offer developers the same service as museums offer to citizens.

References

[1] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[2] S. Freeman and N. Pryce. *Growing object-oriented software, guided by tests.* Addison-Wesley Professional, 2009.

An Integrated Tool Suite for Model-Driven Software Migration towards Service-Oriented Architectures

Andreas Fuhr, Tassilo Horn, Volker Riediger

University of Koblenz-Landau, Germany

{ afuhr | horn | riediger }@uni-koblenz.de

Abstract

Model-driven approaches as well as migration projects rely on a strong tool support. As part of the SOAMIG project, a tool suite has been developed, supporting the model-driven migration of legacy Java and COBOL systems towards Service-oriented Architectures (SOAs). The tool suite integrates a global repository (representing business processes, code and architecture) and capabilities for (i) parsing legacy artifacts (code and project specific architecture constructs), (ii) analyzing legacy code (static and dynamic analysis) and (iii) transforming artifacts.

1 Introduction

Model-driven approaches heavily rely on tools. Without a strong tool support for modeling as well as for querying and transforming models, model-driven techniques are not applicable on real-life projects. Considering the tools, software migration projects have similar dependencies. Without powerful tools for analyzing and migrating legacy systems, these projects are doomed to failure. As a consequence, a combination of both disciplines – called model-driven software migration approaches – requires the development of a strong tool suite for model-driven analysis and migration of legacy systems.

As part of the SOAMIG project[1], a tool suite has been developed, supporting the model-driven software migration of legacy systems towards Service-Oriented Architectures (SOAs). The suite integrates tools for the following tasks:

1. Metamodeling and generation of repository structure; model persistence, querying and transformation
2. Parsing various legacy artifacts (e.g., Java/Cobol code, business processes or project-specific architecture constructs)
3. Analyzing legacy artifacts (static and dynamic analyses)
4. Transforming legacy artifacts
5. Generating code (Java, isolated service code)

In this paper, we describe the SOAMIG tool suite covering the tasks mentioned above. In addition, we present two industrial case studies the tool suit was applied to.

The remainder of this paper is structured as follows. Section 2 introduces the integrated tool suite. Section 3 briefly describes the two case studies, the tool suite was applied to. Finally, Section 4 concludes the paper.

2 The SOAMIG Tool Suite

This section describes the SOAMIG tool suite as shown in Figure 1.

2.1 Repository Technology – The TGraph Approach

One of the core elements of the tool suite is the integrated repository. Artifacts that are used during migration are stored as models in this repository.

The main part of the repository is implemented by TGraphs. TGraphs are typed, attributed, directed and ordered graphs. They are specified by grUML (graph UML) and due to their generic nature used to represent the artifacts of the migration.

With the *Graph Repository Querying Language (GReQL)*, a general-purpose querying language for TGraphs is provided. GReQL is used in static and dynamic analyses to retrieve information from the repository.

With the Graph Repository Transformation Language (GReTL), a generic transformation language for TGraphs is provided. GReTL is used for graph/model transformations.

For accessing and manipulating TGraphs, the Java framework JGraLab (Java Graph Laboratory)[2] can be used.

2.2 Extractor Tools

Various artifacts have been stored as model during the SOAMIG project. For parsing legacy source code,

[1]This work is partially funded by the German Ministry of Education and Research (BMBF) grant 01IS09017C/D. See http://www.soamig.de for further information.

[2]TGraphs, grUML, GReQL, GReTL and JGraLab are developed by the Institute for Software Technology

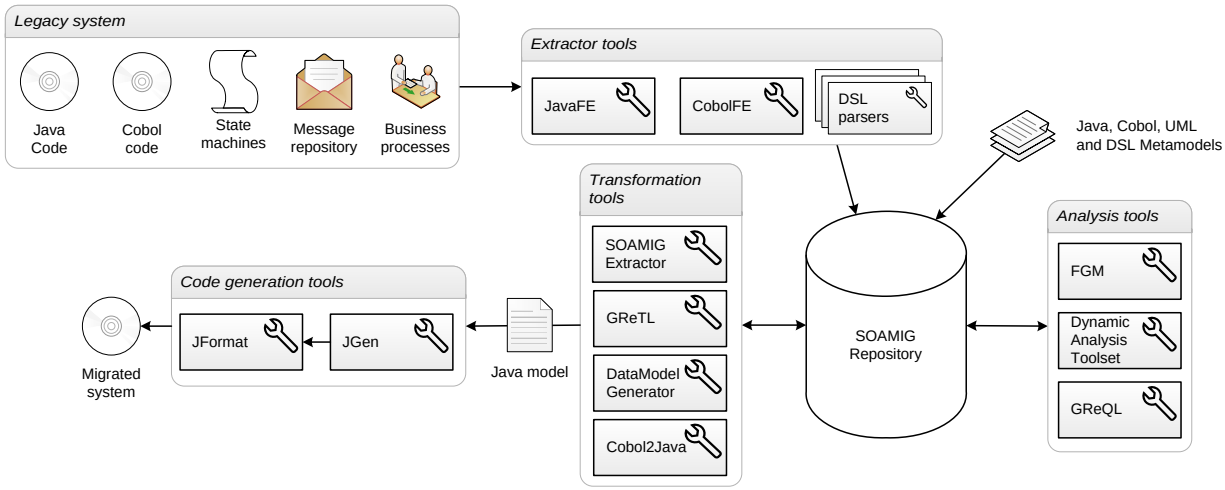


Figure 1: The SOAMIG tool suite.

the *Java Frontend (JavaFE)* and the *Cobol Frontend (CobolFE)*[3] have been used. Both parsers generate fine-grained syntax-graphs which are stored in the repository.

Redocumented business processes (modeled as UML 2.0 activity diagrams), state machines controlling GUI behavior (modeled as UML 2.0 state machines) and an XML message repository (project-specific model) have been parsed using specialized DSL parsers.

2.3 Analysis Tools

For understanding and redocumenting legacy code, the *Flow Graph Manipulator (FGM)*[4] was used. Further static analyses were supported by GReQL.

In addition, a dynamic analysis tool-set-up was developed, tracing which legacy code was executed during a business process. This information was used to integrate the business and code model and to identify legacy code for service implementations.

2.4 Transformation Tools

For Cobol to Java language migration, the tool Cobol2Java[5] was used. Based on sophisticated transformation rules, Cobol2Java transforms Cobol models into Java models.

To enable message exchange in SOA environments, the *Data Model Generator (DMG)* tool was used to generate service-specific data structures. The DMG combines dynamic traces and message repository information.

The *SoamigExtractor*[6] tool provides a graphical user interface for Java model transformations, e.g., pruning generalization hierarchies, slicing of multi-class Java models based on execution traces or establishing traceability links between source and target models. Model transformations may be supported by GReTL.

2.5 Code Generation Tools

Two tools are used for Java code generation: JGen and JFormat[7]. *JGen* is used to unparse a Java model to Java source code, supporting only basic formatting. *JFormat* is an adaptable Java source formatter based on the Eclipse JDT formatter.

3 Case Studies

The tool suite was developed during and tested on two case studies: LCOBOL and RailClient.

LCOBOL was a language migration from Cobol to Java. The tool suite was used to parse various Cobol dialects, transform the Cobol models to Java models (semantics-preserving) and to unparse the Java models to formatted Java source code.

RailClient was a architecture migration aiming at migrating a legacy monolithic Java client to a Service-Oriented Architecture. The tool suite was used for program understanding and for service identification and service migration.

4 Conclusion

As part of the SOAMIG project, a comprehensive tool suite was developed and integrated, supporting model-driven migration towards Service-Oriented Architectures. The tool suite was tested on two case studies. Experiences of these two case studies indicate, that the integrated tool suite is able to provide the required tool support for model-driven software migration projects.

[3] JavaFE and CobolFE are developed by pro et con
[4] FGM is developed by pro et con
[5] Cobol2Java is developed by pro et con
[6] The Data Model Generator and the SoamigExtractor are developed by the Institute for Software Technology
[7] JGen and JFormat are developed by pro et con

A Toolchain for Metrics-based Comparison of COBOL and Migrated Java Systems

Jan Jelschen, Andreas Winter
Carl von Ossietzky Universität, Oldenburg, Germany
{jelschen,winter}@se.uni-oldenburg.de

Abstract

Migrating COBOL legacy systems to Java results in functional equivalent systems expressed in the new language, while the programming paradigm remains that of the old systems. The quality of translated code is therefore assumed to be inferior, if held to the standards of the target language's paradigm. This paper presents an integrated toolchain enabling metrics-based comparisons of original and translated systems to substantiate or refute this hypothesis, characterize the change in code quality, and gain insights for the improvement of translation tools.

1 Introduction

Migrating legacy systems written in old programming languages like COBOL to a more modern language like Java often becomes a necessity, e.g. due to required hard- or software platforms being phased out, or a lack of qualified personnel to maintain the system. Such migrations have to be supported by tools automating translation. However, changing between fundamentally different programming languages also incurs a paradigm shift: while COBOL is procedural, Java is designed to support object-orientation. Tools can map a COBOL program to a functional equivalent Java program, but usually cannot introduce an object-oriented design into a program conceived procedural (cf. Terekhov and Verhoef [7]). Therefore, the code quality of the translated system (e.g. in terms of maintainability) is expected to be lower than that of the original system, at least when evaluated under quality criteria of the target language's paradigm.

In this paper, a toolchain enabling evaluation of code quality metrics on COBOL and translated Java is presented, allowing direct comparisons of quality characteristics. It facilitates the quantification and characterization of quality differences, providing insights to enhance translation tools.

In Section 2, requirements for the toolchain and its composition are described. Section 3 discusses how to approach comparisons of COBOL and translated Java systems using the toolchain and appropriate metrics. The paper concludes with an outlook in Section 4.

2 Toolchain

The toolchain has to support three workflows: *translating COBOL to Java*, *evaluating metrics on COBOL*, and *evaluating metrics on Java*. In the latter case, metrics on Java systems which have *not* been translated from COBOL also have to be evaluable. This extends the scope and flexibility of the toolchain, particularly allowing to use other Java systems as a reference, when direct comparisons with COBOL systems is infeasible (this is further discussed in Sec. 3).

Apart from this, it has to be easy to *extend additional functionality*, e.g. visualization of metric results. An interesting experiment would be to compare different COBOL-to-Java-translators on the basis of metrics evaluated over their respective output. This requires the ability to smoothly *substitute different translator implementations* while leaving the rest of the toolchain unchanged.

Finally, metrics to be evaluated are not known in advance, depending on each specific experiment, necessitating *facile addition and modification of metrics*.

To facilitate the required flexibility, the toolchain is conceived in terms of *services*, abstracting from concrete tools.

2.1 Design and Implementation

Substantial parts of the infrastructure and individual tools have been developed and used in the context of the SOAMIG project [3]. As a basic data structure to hold representations of the processed systems, TGraphs have been chosen. Properties of TGraph instances can be examined using the graph query language GReQL [5], used here to express code metrics.

Data and control flows inside the toolchain, and services involved are depicted in Figure 1 as activity diagram, and explained in the following: *translation* subsumes services concerned with converting Java to COBOL, while *transformation* services provide the required data structures for *metrics* to be evaluated.

Translation. The actual COBOL-to-Java migration is realized by a series of services to *parse COBOL*, translate *COBOL to Java*, and *generate Java* source code. The implementation is provided through the "CoJaC" tools by *pro et con* [1].

The result of parsing COBOL is ouput as an XML file containing an abstract syntax tree (AST). Translating to or *parsing Java* results in a similar XML (parsing Java is, again, provided by a *pro et con* tool.). In Figure 1, these files are shown as datastores *COBOL-AST* and *Java-AST*, respectively.

Transformation. The XML output of the translation tools is TGraph-compatible, and is read into JGraLab's internal (in-memory) representation (depicted twice in the diagram as *XML2TG*).

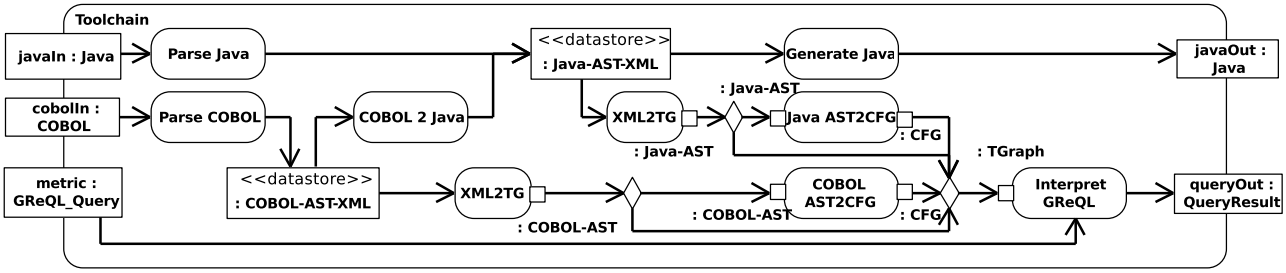


Figure 1: Data and control flow between services of the toolchain.

Metrics can be evaluated directly on the abstract syntax tree representations of either COBOL or Java code. However, many relevant metrics, like *cyclomatic complexity* [6], are more easily evaluated over the control flow graph (CFG) of a program. Therefore, two additional services (*Java AST2CFG* and *COBOL AST2CFG*) to convert Java and COBOL ASTs to CFGs have been implemented for the toolchain. The CFG-representation has the added benefit of being programming language-independent. This allows to express each metric in a single GReQL statement, whereas AST-based metrics have to be formulated over each languages' meta-model separately.

Metrics evaluation. Metrics are not "hard-wired" into the toolchain. Instead, they are provided as an input, expressed in GReQL. Any metric expressible in terms of abstract syntax trees or control flow graphs can be added without changing the infrastructure at all, making the approach very flexible in this regard.

Measures of complexity or structuredness can often be expressed over the CFG [2, pp. 167]. This makes this class of metrics language-independent, and thereby suitable for a direct comparison of original and translated software systems. In addition, the CFG-meta-model is very simple, with only two fundamental concepts: *basic blocks*, connected by *control flows*. Following is an example, showing the *cyclomatic complexity* metric as graph query over the CFG.

```
count(from e : E{ControlFlow} report e end) −
count(from n : V{BasicBlock} report n end) +
count(from p : V{Procedure} report p end) * 2
```

It sums up all basic blocks and all control flows, and subtracts the former from the latter. Then, the number of connected components in the graph times two is added to that (the detection of connected components is simplified here by the fact that there is an "auxiliary" *procedure* node for each of them).

The selection of metrics suitable for comparing COBOL and Java systems has not yet been completed. It is discussed in the following section.

3 Experiments

Due to lack of industrial scale COBOL systems, so far, only small code examples were tested, whose size severely limit the validity of evaluated metrics.

A challenge for further experiments will be to actually achieve comparability: On the one hand, language-independent metrics like *cyclomatic complexity* can be evaluated on both the original and the translated system. The informative value of such metrics – e.g. whether one can directly infer code quality from them – is disputable, though [2, p. 181].

On the other hand, it is desirable to measure the quality of the translated system with respect to the programming paradigm of the target language, i.e. evaluating object-orientation metrics on Java code. There are, however, no canonical COBOL-counterparts for such metrics, leaving open the question of a meaningful comparison.

One approach could be to establish mean values for object-oriented metrics to compare against, by running a large body of representative Java systems through the metric evaluation. While this would still not allow a direct comparison, the quality of a translated system could be rated to be of better, similar, or worse quality than other systems written in the same programming language.

4 Outlook

The toolchain presented here has been conceived with the additional purpose of serving as a case study for tool interoperability. Issues identified in this regard include overhead caused by *data format conversion* and the amount of *glue code* which needs to be written, as well as *platform-specific barriers*. These insights will guide ongoing work on service-based tool interoperability, finally aimed at producing a service-oriented, component-based interoperability framework for software evolution tools [4].

References

[1] U. Erdmenger and D. Uhlig. Ein Translator für die COBOL-Java-Migration. *Softwaretechnik-Trends*, 31(2), 2011.

[2] N. E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman and Hall, 1991.

[3] A. Fuhr, A. Winter, et al. Model-Driven Software Migration - Process Model, Tool Support and Application. To appear in: A. Ionita, M. Litoiu, G. Lewis. *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*, IGI Global, 2012.

[4] J. Jelschen and A. Winter. Towards a Catalogue of Software Evolution Services. *Softwaretechnik-Trends*, 31(2), 2011.

[5] B. Kullbach and A. Winter. Querying as an enabling technology in software reengineering. In *CSMR 1999*, pages 42–50. IEEE CS, 1999.

[6] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.

[7] A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, 2000.

Delta Analysis

Nils Göde, Florian Deissenboeck

CQSE GmbH
Lichtenbergstr. 8, 85748 Garching bei München, Germany

{goede, deissenboeck}@cqse.eu

Abstract

We use various kinds of static analyses to identify problems that decrease the quality of our system. In many cases, however, the number of reported problems is huge—preventing us from solving these problems due to a lack of resources or motivation. We suggest a technique called "delta analysis" together with a simple behavioral rule that allows to deal with large numbers of problems and gradually improves the quality of our system.

1 Introduction

Improving the quality of our software is usually a two-step process. First, we have to identify existing problems and then, we have to remove them. Various kinds of static analyses exists to automatically detect and report existing problems in our software. But while the analyses are often fully automated and easy to apply, the second step—actually removing the problems—is a much harder task.

Independent from the specific type of analysis, tools are likely to report thousands of problems for real-world software which is usually long-lived and contains much legacy code. Since not all of these "problems" have to be real problems and many of them may be more of an indication, we use the more general term *finding* for the remainder of this paper. Especially if no quality control measures have been taken in the past, we often find ourselves confronted with a huge and unmanageable pile of findings when running an initial analysis. Removing all findings at once is in almost all cases infeasible, due to the limitation of resources and the risk of introducing new defects when removing existing findings. In addition, the huge number of findings itself and the comparatively small progress we make in removing them may decrease our motivation and make us resign.

In the remainder of this paper, we summarize our idea of an incremental improvement strategy and explain how it is supplemented by a technique named "delta analysis". The combination of both results in a continuous reduction of findings and ultimately leads to higher quality.

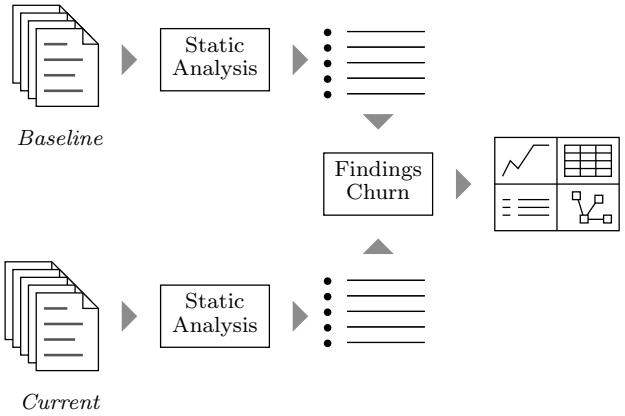


Figure 1: Delta Analysis

2 Incremental Improvement

Instead of solving all findings in a single session, which is in most cases infeasible, we suggest to incrementally remove findings. The central concept of our approach is *whenever you change a file, leave it in a better state than it was before*—inspired by Robert C. Martin's interpretation of the American Boy Scout rule *leave the campground cleaner than you found it* [5].

In our case "better state" or "cleaner" means less findings detected by static analyses. Our assumption is that when you have to change a file, you are forced to read and understand its contents. Given that you have read and understood the file, you are in a very good position to solve at least some of the other findings within that file. Furthermore, the original change and any additionally removed findings can be tested in a single run and do not require separate test runs. If the simple rule is obeyed in any case, the number of findings will naturally go down whereas the quality of our system improves.

On the one hand, we do deliberately not insist on leaving the file in a perfect state, because that may—depending on the number of findings within that file—result in developers not removing any findings due to limited time or a lack of motivation. On the other hand, we require at least a little improvement to ensure a gradual increase of the overall quality.

3 Delta Analysis

At the heart of our delta analysis is comparing the findings found within two successive snapshots of our system. We refer to the earlier snapshot as "baseline" as it defines the state of our system which we would like to improve. Findings within the baseline are seen as legacy problems for which we accept that they exist. We then execute the static analyses for both snapshots separately and compare the detected findings. The result is the *findings churn*, telling us about which findings have been removed, which findings remain, and which findings have been newly introduced. The findings churn allows us to decide whether the state of a file is better after it has been changed. We suggest to integrate the results into a quality dashboard to make them accessible to developers and project managers. The process is illustrated in Figure 1.

In comparison to the traditional static analysis with thousands of findings, we are now confronted with only a comparatively small and manageable number of findings. Furthermore, all findings are related to our latest activity as the delta analysis "hides" all legacy findings which already existed in the baseline. We can now combine this information with our rule of leaving each file in a better state than before to ensure that the overall quality or our system is gradually improving.

4 Advantages

One of the advantages of delta analysis is that the acceptance of the developers is high since they can chose how many findings they actually remove. They are not confronted with an unmanageable number of findings, but can use the results of the delta analysis to inspect those findings that are related to their recent activity. Furthermore, the overall progress of improving the system's quality is measurable by counting the number of removed findings. Being able to actually see that the number of findings is continuously reduced may also increase the developers' motivation and acceptance of quality control measures.

Another positive aspect of delta analysis is that it is not limited to specific types of analyses, but works for all kinds of findings. These include, for example, exceeded metric thresholds, architecture violations [2], and code clones [4]. The concept of delta analysis is independent from the actual source of findings and can also be used for findings detected by tools like *PMD* or *FindBugs*.

Delta analysis also allows to integrate new checks smoothly. Using simple static analyses, integrating new checks usually results in a huge number of findings in the initial analysis and all too often causes the abandonment of the new rule. Using delta analysis makes the integration of new checks much easier, because it focuses on those findings that were introduced since the baseline.

From a technical perspective, results are not stored in a database but the analyses are run for the baseline and the current snapshot of the system every time. This provides a great deal of flexibility, because the baseline can easily be changed. In addition, multiple baselines can be used to analyze the stepwise improvement of the system's quality. Running the delta analysis for the baseline and the current snapshot also allows to change the static analysis algorithms or their configuration while maintaining the comparability of the snapshots.

5 Experience

We have implemented the delta analysis using the ConQAT [1, 3] toolkit. We have integrated the delta analysis as part of the continuous quality control in about two dozen projects of two of our customers, *Munich RE* and *ABB*. The delta analysis has been used in these projects for some time and the feedback from the users is consistently positive. In particular, the users appreciate the possibility to focus on relevant findings which are related to their latest activity. Some of them even regard the delta analysis as an inevitable building block of quality control without which static analyses would not be applicable to large and long-lived systems.

6 Conclusion

In summary, we conclude that delta analysis is a crucial part of continuous quality control as it adds significant value to static analyses as our customers confirmed. Especially for long-lived systems, delta analysis helps to focus on relevant findings by abstracting from the unmanageably huge pile of legacy problems.

References

[1] ConQAT. `www.conqat.org`.

[2] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. Flexible architecture conformance assessment with conqat. In *Proceedings of the International Conference on Software Engineering*, pages 247–250. ACM, 2010.

[3] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. Mas y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Software*, 25(5):60–67, 2008.

[4] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the International Conference on Software Engineering*, pages 485–495. IEEE Computer Society, 2009.

[5] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

How Strict is Your Architecture?

Moritz Beller
Technische Universität München,
Institut für Informatik, D-85748 Garching
beller@in.tum.de

Elmar Jürgens
CQSE GmbH
Lichtenbergstr. 8, D-85748 Garching
juergens@cqse.eu

Abstract

Software architecture has an important impact on maintainability. One of its key functions is to restrict dependencies between system components. However, there are few objective criteria to quantify how well a given architecture does so. In this paper, we propose an objective measure for architecture strictness and report the findings of our case study on architecture strictness with nine real-world software systems.

1 Introduction

Every software system has an architecture [5]. It comprises components, which structure the system into smaller, coherent parts, and policies, which define the allowed and denied dependencies between the components.

A system architect has to design and maintain the architecture specification. During the evolution of a software system, the architecture specification is subject to change [4].

Architecture conformance analysis detects deviations between the architecture specification and the architecture present in the source code. Violations of an intended architecture abound in practice. To remove the violations, the system architect often has to change the architecture specification.

As for other software artifacts, one promising way to perform quality assurance is to review these changes. However, it is difficult to rate the impact of a change, even for an experienced architect with profound knowledge about the system.

As a consequence, during a review it often remains unclear how strong the ability of the architecture to prevent certain dependencies is affected by a change. Thus, we need a tool which allows us to capture this aspect of the architecture design. To support the quality assessor, we propose the architecture strictness ρ and evaluate its relevance in practice in a case study with nine real-world industrial systems answering the following research questions.

2 Architecture Strictness

Architecture strictness ρ is the probability that for two arbitrarily chosen types a and b ($a \neq b$), type a may not access type b in the underlying architecture.[1]

A value of $\rho = 0$ characterises a lax system, where all types can access each other. $\rho = 1$ characterises a restrictive system, in which every type can only access itself.

A type pair is a tuple $< t_1, t_2 >$ of two types. If t_1 may depend on t_2 according to the architecture specification, then the pair is an allowed pair. To calculate ρ on an architecture, we use the formula

$$\begin{aligned} \rho &= \frac{\text{Number of Denied Pairs}}{\text{Number of All Pairs}} \\ &= 1 - \frac{\text{Number of Allowed Pairs}}{\text{Number of Possible Pairs}}. \end{aligned} \quad (1)$$

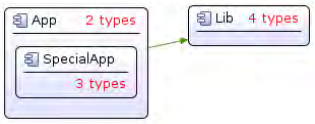


Figure 1: Example ConQAT architecture with one allowed policy from App to Lib. The number of classes mapped to the component is denoted behind the component's name.

As an example, we calculate ρ on figure 1 by summarising all allowed and then all disallowed type pairs.

Reason	Allowed type pairs
Types in same component	$2 \cdot 1 + 3 \cdot 2 + 4 \cdot 3 = 20$
Policy from *App* to *Lib*	$(2 + 3) \cdot 4 = 20$
Sub or super type (*SpecialApp* and *App*)	$(2 \cdot 3) \cdot 2 = 12$
Total	52

Denied pairs are from *Lib* to *App*, which are $(2 + 3) \cdot 4 = 20$ type pairs. Thus, the number of all possible pairs is $52 + 20 = 72$. Using equation 1, we receive $\rho = 1 - \frac{52}{72} \approx 0.278$.

3 Study Objects and Procedure

We performed a case study on nine industrial systems developed and maintained by Munich Re. Munich Re is an international reinsurance company with over 40,000 employees worldwide. The study objects are nine heterogeneous C#-written projects (cf. table 1). A prerequisite for projects at Munich Re is to manage architectural knowledge with a ConQAT architecture specification.

We implemented an automatic calculation of the architecture strictness with the open source quality assessment toolkit ConQAT[2]. The details of ConQAT architecture specifications are described in [2, 3]. The

[1] In the context of C# and Java, types are classes.

[2] Available from http://www.conqat.org

System Characteristics			System Metrics		Strictness Analysis	
System Name	#Components	#Types	LoC	CC	ρ	Change Range
System A	41	1,732	322,870	27.8%	0.673	$[-0.077; 0.033] = 11.0\%$
System B	34	499	56,247	12.4%	0.555	$[-0.064; 0.064] = 12.8\%$
System C	37	660	53,155	27.2%	0.799	$[-0.069; 0.015] = 8.4\%$
System D	39	1,072	160,916	6.2%	0.522	$[-0.131; 0.116] = 24.7\%$
System F	98	4,479	328,975	7.2%	0.611	$[-0.075; 0.075] = 15.0\%$
System H	31	2,034	526,513	20.0%	0.462	$[-0.116; 0.116] = 23.2\%$
System I	71	8,436	1,417,578	21.7%	0.724	$[-0.124; 0.024] = 14.8\%$

Table 1: Excerpt of the nine study objects System A-I.

detailed results to the research questions are given in table 1.

4 Q1: How does ρ differ between systems?

We found that at a range of 0.462 to 0.799 ρ is well-spread.

5 Q2: How do low-restrictive architectures differ from high-restrictive ones?

We manually examined the architecture of the systems with the lowest strictness (Systems D and H), and compared them to System C with the highest strictness in the study. Systems D and H have two and three top-level components, while System C has seven top-level components. Allowed accesses between the top-level components in Systems D and H lower their architecture strictness, but are absent in System C.

The results suggest that for a higher ρ the architecture ought to have several unconnected top-level components of equal size (like System C).

6 Q3: Does ρ correlate with established metrics?

We observed no correlation between LoC (Lines of Code, including blank lines and comments) and strictness: Pearson's $r = 0.18$. However, there is a moderate correlation between CC (Clone Coverage, the probability that a source code statement is covered by at least one clone, an indicator of duplication and quality defects in the code) and strictness at $r = 0.64$.

Together with Q1 this indicates that strictness is largely independent of the system size, allowing us to compare different-sized systems. Generally, a high strictness equals few allowed dependencies in relation to the possible dependencies in the system (cf. equation 1). However, allowing only few dependencies to other types could hinder the principle of code reuse. Therefore, in a restrictive systems, code duplication could be a consequence. This might be a reason why we observed a moderate correlation between strictness and CC.

7 Q4: How does the range of atomic changes to ρ differ between systems?

An atomic change is the removal or addition of an arbitrary policy in an architecture. The variables change range and ρ show a strong negative correlation $r = -0.74$. This supports the interpretation of Q2: High ρ architectures are designed so that a modification to the architecture does not change its ρ so much: An atomic change between small to medium components never has such an effect on ρ as between large components (like Systems H and D).

We found that change range in ρ varies. Moreover, it is negatively correlated with ρ.

8 Related Work

Bouwers et al. [1] propose the Component Balance Metric to capture the analyzability of an architecture. The metric is a function of the number of top-level components and their relative sizes.

Wermelinger et al. [6] perform a case study on the Eclipse SDK's architectural changes from version 1.0 to 3.5.1. They compute a series of related metrics, among which are the coupling and cohesion metrics.

While these are computationally similar architectural metrics, to the best of our knowledge there are no measures that focus on an explicit reference architecture and measure its ability to prohibit dependencies.

9 Conclusion

Based on the results of our case study, we found that strictness differs among systems. We explained outliers by manual inspection and extracted some theses for system designs that lead to strict architecture specifications. Other known metrics do not encompass the architecture strictness. The topic remains an interesting field for future research: We will make a larger case study with (open source) non-C# projects from a different domain. A more elaborate study on code duplication and its implications on the architecture strictness could clarify the correlation.

References

[1] E. Bouwers, J.P. Correia, A. van Deursen, and J. Visser. Quantifying the Analyzability of Software Architectures. 2011.

[2] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. Mas y Parareda, and M. Pizka. Tool Support for Continuous Quality Control. *IEEE Software*, 25(5):60–67, 2008.

[3] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Elmar Juergens. Flexible Architecture Conformance Assessment with ConQAT. In *ICSE'10*, 2010.

[4] M. Feilkas, D. Ratiu, and E. Jurgens. The loss of architectural knowledge during system evolution: An industrial case study. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 188–197. IEEE, 2009.

[5] M.W. Maier, D. Emery, and R. Hilliard. ANSI/IEEE 1471. *Systems Engineering*, 7(3):257–270, 2004.

[6] M. Wermelinger, Y. Yu, A. Lozano, and A. Capiluppi. Assessing architectural evolution: A case study. *Empirical Software Engineering*, pages 1–44, 2011.