Static analysis on integration level

Heiko Frey

Robert Bosch GmbH, P.O. Box 1661, D-71226 Leonberg, E-mail: heiko.frey@de.bosch.com.

Abstract—Static analysis on integration level is a method of checking the usage of interfaces between software modules (function calls and global variables) without executing the software system. The interfaces are analyzed by a tool without referencing an interface specification or documented software architecture. An interface is classified as suspicious if it is neither written nor read, if it is globally defined but only used locally, or not used at all. In a tool supported analysis, all suspicious interfaces are collected and then evaluated for their validity within the context of a formal review. Invalid findings may result from pointer accesses, future development plans or unused interfaces which are provided by a library component. The static analysis on integration level was executed on a small embedded system pre-development project written in C. The results show - although the number of invalid findings is relatively high - findings which would not have been found by another test method.

1 INTRODUCTION

Static analysis is used to evaluate a software artifact (typically source code) based on its form, structure, content or documentation [1] in a static way, which means without execution. The method is widely used in the industry and subject of research in various characteristics (e.g. [2],[3],[4],[5],[6]). In general, static analysis is performed by a tool (called automated static analysis). Static analysis concentrates on suspicious code elements which could be identified during a review as well. However performing such a review manually would require significant effort from the reviewers. Automating this procedure using a tool provides a more consistent analysis with reduced effort. Potential defects which can be found by an automated static analyzer because they are statically visible are:

- Anomalies in the code structure
  - data flow anomalies, e.g. usage of an uninitialized variable[7]
  - control flow anomalies, e.g. unreachable code
- Violation of coding guideline [4]
- Also, static analysis tools are used to calculated metrics of the software artifact.

For automated static analysis of source code, many tools are available on the market (a good overview can be found on the website of Denis Faught [8]). These tools use different analysis techniques and subsequently they produce different kinds of findings. Automated static analysis highlights findings by attaching the expression "warning" to them. An expert who knows the analyzed code well (e.g. the author of a module) must evaluate each warning whether it is valid or not. Xiao and Pham describe in [2] a tool to store the warnings of different static analyzers as well as an approach to reduce the "noise" of invalid warnings. However, a manual evaluation of each warning to decide about validity or invalidity still must be done before changing the code for the sake of defect removal (in case of a valid warning) or before deactivating the warning (in case the warning is invalid).

In terms of integration testing Leung and White [9] propose static analysis "to check the data type, format and the parameter passing rules of each parameter, and the order and number of parameters". Other publications use static analysis on integration level before running a dynamic integration test in order to detect statically visible defects on integration level and to prepare for the dynamic integration test by retrieving information about the interface structure, control- and data flows. The steps performed during static analysis are according to Spillner [10]:

- Check the syntactical correctness of the interfaces. This check is partly done by a compiler or linker [11], but for some deviations (e.g. compiler tolerates differences of type) additional analysis might be necessary.
- Categorize couplings between modules.
- Uncover hidden dependencies, for example if two modules use a global variable which is managed by a third module.
- Perform intermodular dataflow analysis to uncover intermodular dataflow anomalies and to uncover hidden dependencies (e.g. via a global variable).

Static analysis on integration level supports refactoring which was defined by Fowler [12] as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior". The difference between refactoring and the method described in this article is, that refactoring uses regression testing to assure that no new bugs are added during the refactoring process. Static analysis on integration level doesn't require regression test cases, although they can be used. In fact, the analysis is based on the source code and can be performed without any documented architecture or design. However, the method describes a way to discover objects to be refactored in a C-code project with a poorly (or not) documented architecture using review techniques.

To discover the suspicious objects we enhanced the tool QA-C[1] for static analysis. The enhancement allows

[1] QA-C is a registered trademark of Programming Research Ltd.

to analyze the interfaces and dependencies between several modules automatically and to uncover anomalies by doing so.

2 STATIC ANALYSIS ON INTEGRATION LEVEL

2.1 Alternatives

The verification of the interfaces and dependencies between modules could be performed by a review of the source code and the software architecture and/or a dynamic integration test. However, these two methods have disadvantages for uncovering anomalies in interfaces between modules which are statically visible:

The review has to be performed manually and the effort for reviewing the module interfaces depends on the complexity of the interfaces and the coupling to other modules. The effort on performing a review on source code increases with code complexity for a given quality level of the review. Complexity is often measured by LOC[13] or cyclomatic complexity [14],[4]. However, the number and type of interfaces – i.e. the degree of coupling – also has an impact on complexity. Jin and Offut [15] characterize couplings between functions on 12 levels based on call coupling, control coupling and data coupling. Measures for coupling complexity are for example Akiyama's criterion [16], the number of (distinct) function calls [17], the number of function parameters of a function [18] or the number of used global variables of a function or system. Thus, keeping an overview of all dependencies of many modules is usually hard to achieve for a reviewer without spending a large effort.

A dynamic test relates to a specified architecture of a software system. Not reaching a specified interface coverage by running the test or manual rework to find the reason of the unexpected low coverage can help to uncover anomalies related to unspecified interfaces (see [19] for possible integration test coverage criteria). Additionally, integration test cases have to be specified and implemented. This manual effort depends, as for reviews, on the degree of coupling.

2.2 Description of the static analysis on integration level

As mentioned above, the static analysis tool QA-C is our basis for the static analysis on integration level. The tool collects information about the structure of every analyzed c-file and stores this information in a so called metric output file. The following information is collected in this file [18]:

- Relationship records for file including, internal and external call linkage,
- external reference records,
- define and declaration records,
- control graph records,
- metrics records,
- pragma records (compiler specific commands),
- literal usage records.

The format of the metric output file is published and well-documented [18].

QA-C itself offers the possibility to perform a "cross-module analysis" (CMA) which combines the information of all project metric output files to uncover some structural anomalies. These anomalies are unused external identifiers, function recursion and declaration crosschecks [18]. For performing the static analysis on integration level we use the cross-module-analysis offered by QA-C to detect multiple declarations of global variables and functions.

To provide a deeper analysis of the interfaces, we have developed a program which combines the information from the metric output files to uncover additional anomalies within the interfaces of a software system. The program does the following to identify suspicious interfaces in the software system:

1. *Identify global variables:* All metric output files are parsed for global variables and a list of all global variables carrying a different name is created. If global variables have the same name but a different type, they are treated as two separate variables. Usually the compiler/linker should give an error if the variables have the same name but a different type. However, some differences of type might be tolerated by the compiler/linker (e.g. signed/unsigned int). With the analyzer program, the code is not compiled/linked, but analyzed on source code level and type tolerances are not allowed.
2. *Analyze dependencies between global variables*: For every global variable create a list containing the following information: where is the variable declared (filename, line no.), defined (filename, line no.), written and read (functionname, filename, line number). For every declaration create a list containing the information if the declaration is done by including a file (e.g. a header file) or directly.
3. *Collect metrics for global variables*:
   For every global variable count write and read access in two ways: 1) count every function writing or reading one certain global variable only once (distinct). 2) Count also multiple accesses from a single function (transitive). Furthermore, for every global variable count the number of files which write or read it. (These metrics are used later on to find suspicious interfaces.)
4. *Identify globally visible functions*:
   Collect the globally visible functions from the metric output files into a common list. Treat globally visible functions with the same name but of a different type as two different functions for the same reason as for global variables (see 1.).
5. *Identify dependencies between globally visible functions*:
   For every globally visible function create a list containing the following:
   - Location of declaration (filename, line number),
   - location of definition (filename, line number),
   - whether the declaration was done directly or via a

header file,

- location and name of function calls of globally visible functions (functionname, filename, line number),
- location and name of functions called by globally visible functions (functionname, filename, line number)
- function calls of globally visible and local functions,
- location and name of global variables write or read accesses performed by a function (variable name, filename, line number).

6. *Collect metrics for globally visible functions*:
   For every globally visible function count the number of calls of other globally visible functions: a) count the number of functions which are called (distinct), and b) also count multiple calls of the same function (transitive).
   For every function which is globally visible, count the number of calls by other functions (here globally visible and local functions): a) counts the number of functions by which the function is called (distinct), and b) also count multiple calls by identical functions (transitive).
   Furthermore, for every globally visible function count the number of files from which the function is called.

The information collected is exported in two tables, one with information based on the global variables, and one with information based on the globally visible functions.

2.3 Detectable anomalies by static analysis on integration level

By filtering on the metrics which are calculated and displayed in the two tables, we can uncover the following anomalies:

Control flow anomalies:

- Unused function: The number of functions calling this particular function equals 0. An unused function doesn't usually occupy program memory, as the linker sees that the function is not called. However, this "useless" function can be removed to improve the clearness of the architecture and the understandability of the software system.
- Global function only used locally: The umber of other functions accessing this function from different files equals 1.
- To increase a higher encapsulation of a module, only the functions which are meant to be called by other modules should be externally visible. By applying this filter, functions which are globally visible but called only by local functions are highlighted.

Data flow anomalies:

- *Dead global variable:*
  The number of functions reading one certain global variable equals 0, and the number of functions writing on one certain global variable equals 0. A variable which is neither written nor read carries no functionality. Therefore it can be removed in most cases to improve the data flow architecture and the readability of the code.
- *Global variable written but not read (du-anomaly*): The number of functions reading equals 0, and the number of functions writing is greater zero. An expert needs to evaluate, if the variable is really needed – maybe to store a value for debugging investigations during runtime. If the variable has no use it can be removed to improve the architecture.
- *Global variable not written but read (ur-anomaly):* The number of functions reading is more than 0 and the number of functions writing equals 0. An expert needs to clarify, if it is intended and known that the variable is not written within the regular program flow. In some cases, developers rely on the default initialization of the global variable which is added (typically optional) by the compiler. However, this is risky, if the module is used with a different compiler.
- *Global variables only used locally:*
  The number of variable accesses from different files equals 1. As mentioned for globally visible functions which are only used locally, the same is valid for global variables: Data should be encapsulated and hidden to avoid undesired (miss)-use of the data. This anomaly could be a result from former debugging activities because it is difficult to observe local variables using a debugger.
- *Multiple declarations:*
  This information is taken directly from QA-C cross-module-analysis. It shows identifiers which are declared more than once for the same type or for different types, e.g. a type definition carries the same name as a global variable.

All findings highlighted by filtering for the described metric above have to be evaluated in a formal review to decide about the validity of each finding. The process of this evaluation is described further below.

Possible findings are also subjects for refactoring activities according to the publications of Fowler, Opdyke and Garrido [11], [20], [21]. Refactoring has a wider scope as our focus on interfaces. However, this paper concentrates on the process of evaluating findings concerning inter-modular interfaces and the experience with a sample project.

3 SAMPLE PROJECT

3.1 Project Description

The static analysis on integration level was performed on a small embedded software project, which is the first iteration of a software platform for an automotive control unit. The software system consists of 31 modules, which are arranged in different layers:

- Application layer (APP)
- Application abstraction layer (ASAL)
- Electronic control unit abstraction layer (ESAL)
- Microcontroller abstraction layer (µCAL)
- Network and communication layer (NETCOM)

- Service layer (SERVICE)
- System layer (SYS)

Control- and data flows should occur only horizontally between SERVICE, SYS, NETCOM and either APP, ASAL, ESAL, µCAL or vertically downwards from APP to ASAL to ESAL to µCAL (see Fig. 1.). This strategy is used to achieve a defined functional responsibility of every layer and to avoid a shared responsibility over several layers. See arrows in Fig. 1.

The modules are developed in three different ways:

- *Handwritten code (H):*
  The code is written manually with a code editor.
- *Model-based developed with code generation (M):*
  The module is modelled with a modelling tool (Matlab/Simulink) and code is generated out of the model. It is also possible to include handwritten code sequences within the model.
- *Configured and code-generated library component (L):*
  A library provided by a different external or internal organization is used (e.g. COTS). The configuration of the library may contain handwritten code.

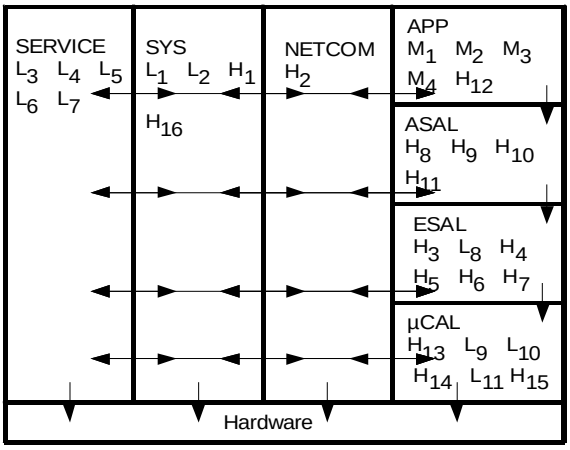

Fig. 1. Arrangement of the modules in architectural layers. The allowed communication paths between the layers is displayed by the arrows.

The software system has the following characterizing metrics:

- number of modules: 31
- number of files (.c and .h-files): 154
- number of function definitions (global and static): 382
- number of static identifiers: 127
- number of variables declared as external: 330
- executable lines of code (LOC): 8580
- cumulated cyclomatic complexity: 1754

A functional specification for the control unit does exist, as well as a rudimentary software architecture and module designs in different qualities.

The developers performed the static analysis on module level with QA-C and evaluated the warnings. The goal was to have no warnings left. Either the code which produced the warning had to be changed, or the warning had to be deactivated locally, if the warning was invalid or a work-around would not have been feasible.

The software has been integrated and the developers have performed basic test to test the functionality of their modules in the integrated environment.

Reviews have been performed for all modules: Design reviews for model-based developed modules. Code reviews for handwritten modules and handwritten parts of model-based modules or handwritten configurations. Integration reviews with the provider of the library components for reviewing the configuration and integration of the library components.

Errors or findings which have been found during these verification steps have been removed and the verification steps have been repeated, if a bigger change in the code was needed. A review was used to decide whether repeating one or more verification steps is necessary or not.

3.2 Process to evaluate warnings

After the two tables are created and the suspicious variables and functions are filtered out, a review has to be performed. Participants are all software developers of the project, the software project manager and the test manager who organizes and moderates the review meeting. It is important that all software developers join the review meeting, because it is highly likely that interfaces between two modules from different authors are suspicious.

The test manager explains the tables and how the findings have to be interpreted. The group walks through the table and decides for each suspicious variable and function if the findings are valid, invalid or if further investigation is required. In case of invalidity, a comment explains the decision. To speed-up this evaluation process, each developer should have easy access to the source code to quickly look up the code location and to be able to explain briefly, why a finding is valid or not. If further investigation is required, a responsible developer is named.

A finding is classified as valid if there is no reason to keep the interface, agreed by the team.

A finding is classified as invalid if there is a reason to keep the interface, agreed by the team. Possible reasons can be:

- An interface (function or global variable) is accessed via a pointer. Such an access is not covered by a static analysis in most cases.
- An interface is intended to be used in the future development.
- A function or global variable is a part of library code or COTS and is not used in the analyzed software system. Library code is usually not intended to be changed and the object will in most cases not appear in the object code due to an optimizing compiler.

4 RESULTS

4.1 Valid and invalid findings

During the review meeting 368 findings have been evaluated for validity or invalidity. The evaluation contained also a classification to which module the finding belongs and if the finding belongs to handwritten, generated or library code. This means, a handwritten configuration of a library component is classified as handwritten code but it belongs to a library module.

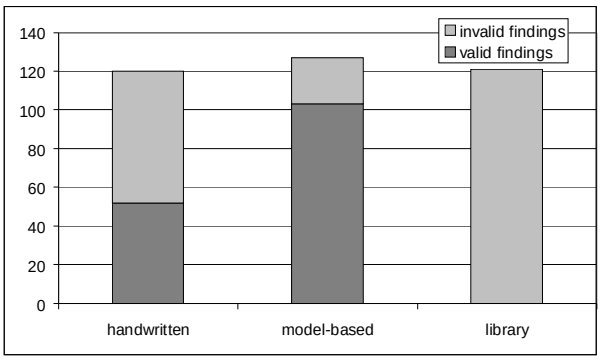

Fig. 2. Distribution of valid and invalid findings over the different ways of code development.

As figure 2 shows, most findings are invalid, especially concerning the library modules. But it is also visible, that the model-based developed and generated code contains many valid findings.

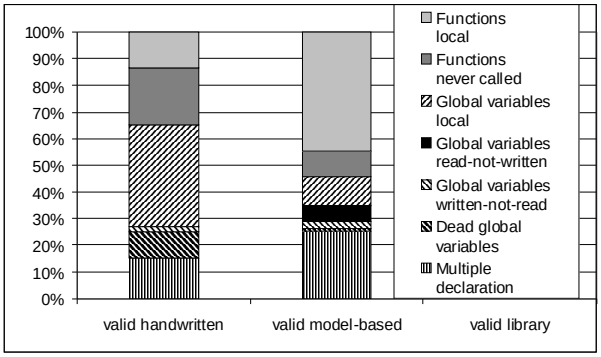

Fig. 3. Distribution of the valid findings (sorted by finding type) over the different ways of code development.

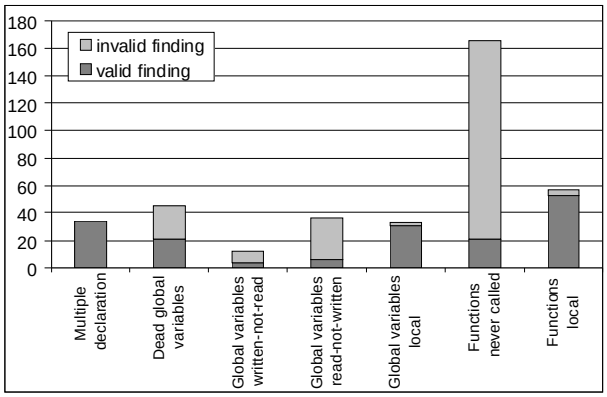

Fig. 4. Valid and invalid findings sorted by the different types of findings

Figure 3 shows that for handwritten modules, the most common anomaly is a write access to a global variable which is never read. For model-based and code-generated modules a common anomaly is that many functions are declared as global but used only locally. This problem was caused due to a wrong configuration of the code generator. It wasn't uncovered by a code review since code reviews are not performed on generated code.

Figure 4 shows that the findings of the type "Functions never called" are in most cases invalid.

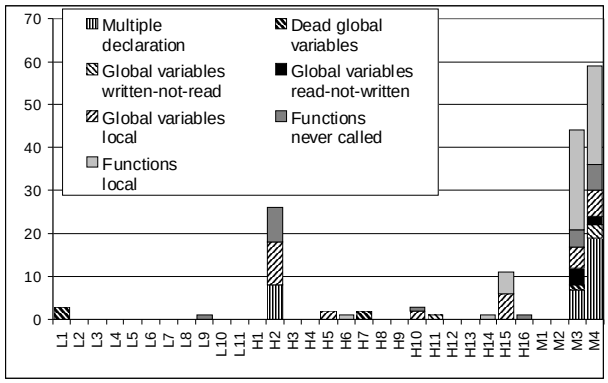

Fig. 5. Distribution of the valid findings over the different modules. (L)ibrary modules, (H)andwritten modules, (M)odel-based developed modules.

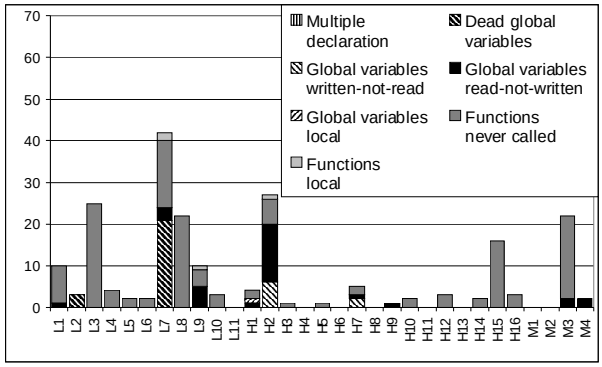

Fig. 6. Distribution of the invalid findings over the different modules. (L)ibrary modules, (H)andwritten modules, (M)odel-based developed modules.

Figure 5 shows the distribution of the valid findings to the different module. Library modules have valid findings as well, although the diagrams before show no valid findings for these modules. The valid findings result from handwritten configuration code for the library modules. The library code itself delivered to the software project had no valid finding.

Figure 6 shows the distribution of the invalid findings to the different module. The most invalid findings are within the library modules. These are in most cases functions never called. The library provides these functions, but the project doesn't use them. They will not be linked to the object code due to the optimizer algorithm of the linker. The other invalid findings have been evaluated and their invalidity is due to the reasons as named in chapter 3.2.

4.2 Value of the static analysis on integration level

The results show a high amount of invalid findings

(noise). This noise results from the high maturity of the library components and the rest of the software modules being in an early phase of the software life cycle. The invalid findings in the library components come from unused functions and unused global variables. Most of these objects will not be in the object code because of optimization by the compiler. The invalid findings in the rest of the modules are result of already provided interfaces which are going to be used in the future. Additional analyses will show, if these interfaces are really used.

The valid findings mainly show a miss-configuration of the generated code, which was developed using a modelling tool and a code generator. Other valid findings result from code which was added for debugging purpose or multiple declarations. All these findings would have been hard to detect by other testing techniques like code reviews or dynamic testing. So, for the valid findings, the static analysis shows a benefit and a chance for improvement of the code structure.

Additionally the review meeting gave the possibility to the developers to see their work from different point of view. Couplings became visible which have not been on their radar before. Developers now can identify those modules their own module stays in contact with and they start to ask questions about dependencies concerning runtime and call sequence. This means the generated tables are a good basis for future reviews concentrating on dependencies.

5 CONCLUSION AND OUTLOOK

Static analysis on integration level describes a method, which makes dependencies between software modules visible. Especially anomalies in the interface become visible, like unused or locally used global variables or globally visible functions. The dependencies and anomalies are listed in a table and a filter can be applied to highlight the different anomaly types (findings). In a review meeting, the findings are evaluated by the software developers to decide which findings are valid and which not.

The method has been applied to a sample project which showed many valid findings in generated model-based developed code and handwritten modules and many invalid findings in library code. Also many invalid findings came from interfaces which have already been implemented for future development of the sample project. However the valid findings show aspects which would be difficult to detect by other test methods, as most test methods (e.g. code reviews or software module testing) concentrate on the structure of single modules and not on the interface structure of inter-modular interfaces. To detect the anomalies by dynamic integration testing, specified test cases basing on the interface definitions would have been necessary, causing much effort to develop these test cases. This was not possible in our example, especially because the architecture was only defined rudimentary.

The high rate of invalid findings shows the high quality of the library code. Future work will therefore concentrate on the following questions: How can we improve the method by reducing the noise produced by invalid findings? How can an algorithm decide on the validity of a finding? Which modules don't need to be considered in an analysis?

Acknowledgment

I want to thank the software development team of the sample project in body electronics development of the Robert Bosch GmbH, Leonberg, Germany. Special thanks go the software project manager, Mr. Kenneth Garvey, for the long and intensive discussions and the support for performing the static analysis on integration level preparation and review.

REFERENCES

[1] IEEE610.12-1990, IEEE standard glossary of software engineering terminology, 1990

[2] S. Xiao, C. Pham, “Performing High Efficiency Source Code Static Analysis with Intelligent Extensions”, in Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC '04), 2004, pp. 346-355

[3] J. Laski, W. Stanley, J. Hurst, “Dependency Analysis of Ada Programs”, in Proceedings of the 1998 annual ACM SIGAda international conference on Ada, 1998, pp. 263-275

[4] T. Tanaka et al., “Software Quality Analysis & Measurement Service Activity in the Company”, in Proceeding of the 1998 International Conference on Software Engineering, 1998, pp. 426-429

[5] A. Aggarwal, P. Jalote, “Integrating static and dynamic analysis for detecting vulnerabilities”, in 30th Annual International Computer Software and Applications Conference, 2006, COMPSAC '06, 2006, pp. 343-350

[6] G. Brat, W. Visser, “Combining Static Analysis and Model Checking for Software Analysis”, in Proceedings of the 16th IEEE international conference on Automated software engineering, 2001, pp. 262-269

[7] R. N. Taylor, L. J. Osterweil, “Anomaly Detection in Concurrent Software by Static Data Flow Analysis”, IEEE Transactions on Software Engineering,Volume SE-6, Issue 3, May 1980, pp. 265-278

[8] D. Faught, “Static Analysis Tools”, Available: http://www.testingfaqs.org/t-static.html, 2008

[9] H. K. N. Leung, L. White, ”A study of integration testing and software regression at the integration level”, in Conference on software maintenance 1990, Proceedings, Nov. 1990, pp. 290-301

[10] A. Spillner, R. Franck, J. Herrmann, “Integrationstest großer Softwaresysteme”, in Software-Entwicklung: Konzepte, Erfahrungen, Perspektiven, W.-M. Lippe (Ed.), 1989, pp.118-132

[11] A. Spillner, T. Linz, H. Schaefer, “Software Testing Foundations”, 2nd edition, Rocky Nook, Santa Barbara, 2007

[12] R. Fowler, “Refactoring: Improving the Design of Existing Code”, Addison-Wesley Longman, Jul. 1999

[13] M. Sheppard, “A critique of cyclomatic complexity as a software metric”, in Software Engineering Journal, Volume 3, Issue 2, Mar 1988, pp. 30-36

[14] *T. J. McCabe, “A Complexity Measure”*, in *IEEE Transactions on Software Engineering*, Band SE-2, 1976, pp. 308-320

[15] Z. Jin, A.J. Offutt, “Integration testing based on software couplings”, in Proceedings of the Tenth Annual Conference on Computer Assurance, 1995, COMPASS '95, Jun. 1995, pp. 13-23

[16] F. Akiyama, "An example of software system debugging," in Proceedings of the IFIP Congress 1971, Aug. 1971, pp. 353-358

[17] D.L. Brandl, “Quality measures in design: finding problems before coding”, ACM SIGSOFT Software Engineering Notes, Volume 15 , Issue 1, Jan. 1990, pp. 68-72

[18] QA-C 7.0, User's guide, Programming Research Ltd., 2007

[19] U. Linnenkugel, M. Müllerburg, “ Test data selection criteria for (software) integration testing”, in Proceedings of the First International Conference on Systems Integration, 1990. Systems Integration '90, Apr. 1990, pp. 709-717

[20] W. F. Opdyke, “Refactoring object-oriented frameworks”, PhD dissertation, University of Illinois at Urbana-Champaign, 1992

[21] A. Garrido, “Software refactoring applied to C programming language”, Master Thesis, University of Illinois at Urbana-Champaign, 2000