BENEVOL 2008 : the 7th Belgian-Netherlands software eVOLution workshop proceedings, December 11-12, 2008, Eindhoven : informal pre-proceedings

Sponsors

BENEVOL 2008 has been made possible with the generous help of the following institutions:

The workshop is supported by the ERCIM Working Group on Software Evolution.

Preface

This volume presents informal pre-proceedings of BENEVOL 2008, the 7th BElgian-NEtherlands software eVOLution workshop hold in Eindhoven on December 11-12, 2008. The aim of the workshop is to bring researchers to identify and discuss important principles, problems, techniques and results related to software evolution research and practice.

Program of the workshop consisted of an invited talk by Prof. Ralf Lämmel (University of Koblenz) on *Tracking the evolution of grammar-like knowledge in software* as well as of presentations of nineteen technical papers, gathered and bound in this volume.

December 2008

Alexander Serebrenik
Program Chair
BENEVOL 2008

Organization

BENEVOL 2008 is organized by the department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands.

Organizing Committee

Mark van den Brand
Christine van Gils
Erik Scheffers
Alexander Serebrenik

Acknowledgements

The organisers are very grateful to Luc Engelen for designing the BENEVOL poster.

Table of Contents

Relationship between Size, Effort, Duration and Number of Contributors in Large FLOSS projects

Juan Fernandez-Ramil[1,3], Daniel Izquierdo-Cortazar[2] and Tom Mens[3]

[1] The Open University, Milton Keynes, U.K. j.f.ramil@open.ac.uk
[2] Universidad Rey Juan Carlos, Madrid, Spain dizquierdo@gsyc.urjc.es
[3] Université de Mons-Hainaut, Mons, Belgium {j.f.ramil,tom.mens}@umh.ac.be

1 Introduction

This contribution presents initial results in the study of the relationship between size, effort, duration and number of contributors in eleven evolving Free/Libre Open Source Software (FLOSS) projects, in the range from approx. 650,000 to 5,300,000 lines of code. Our initial motivation was to estimate how much effort is involved in achieving a large FLOSS system. Software cost estimation for proprietary projects has been an active area of study for many years (e.g. [1][2][3]). However, to our knowledge, no previous similar research has been conducted in FLOSS effort estimation. This research can help planning the evolution of future FLOSS projects and in comparing them with proprietary systems. Companies that are actively developing FLOSS may benefit from such estimates [4]. Such estimates may also help to identify the productivity 'baseline' for evaluating improvements in process, methods and tools for FLOSS evolution. Table 1 shows the projects that we have considered, together with the programming language(s) primarily used for each system and the time from the first known commit.

Table 1. FLOSS systems studied and some of their characteristics

name	primary language	description	time from 1st commit (years)
Blender	C/C++	cross-platform tool suite for 3D animation	6
Eclipse	Java	IDE and application framework	6.9
FPC	Pascal	Pascal compiler	3.4
GCC	C/Java/Ada	GNU Compiler Collection	19.9
GCL	C/Lisp/ASM	GNU Common Lisp	8.8
GDB	C/C++	GNU Debugger	9.5
GIMP	C	GNU Image Manipulation Program	10.9
GNUBinUtils	C/C++	collection of binary tools	9.4
NCBITools	C/C++	libraries for biology applications	15.4
WireShark	C	network traffic analyser	10
XEmacs	Lisp/C	text editor and application development system	12.2

The measurements extracted for this study are listed in Table 2. We used the SLOCCount[4] tool to measure the size of the software in lines of code. In order to measure size in number of files, duration, effort and team size, we used CVSAnalY[5]. This tool stores information extracted from the version control log (CVS, Subversion or Git) in a MySQL database. Specifically, we indirectly used CVSAnalY by downloading data from FLOSSMetrics[6]. For mesuring $EFFORT$, possibly in a very approximate way, we added one contributor-month for each contributor making one commit or more in a given month. Then, we calculated the number of contributor-years by dividing by 12 the total contributor-month values for each project.

Table 2. Measured attributes for each system

attribute name	description	extracted using:
$KLOC$	physical lines of source code (in thousands)	SLOCCount
$netKLOC$	see explantion in text	
$FILES$	total number of files in code repository	CVSAnalY
$netFILES$	see explanation in text	
$EFFORT$	effort in contributor-years	CVSAnalY
DUR	time length of 'active' evolution in years	CVSAnalY
DEV	number of unique contributors	CVSAnalY

2 Data filtering

In most of the projects, when looking at the total number of files ($FILES$) in the repository per month, we observed a number of relatively large 'jumps'. Figure 1 (left hand side) shows this for project GCL. It is unlikely that the productivity of the team of contributors has increased so suddenly. It is more likely that each 'jump' reflects moments in which the repository receives chunks of externally generated files. It could also be that non-code files were added. We tried to filter out this phenomenon by subtracting the size increments in the 'jumps'. We also subtracted the initial size. We call the result $netFILES$. For GCL, the result is presented on the right hand side of Figure 1. Since $KLOC$ was measured only at one recent date, we also defined $netKLOC$ as $KLOC * \frac{netFILES}{FILES}$.

When measuring project life span or duration DUR since the first commit up to Oct 2008 – for Eclipse the cut off month was April 2008 –, we excluded any apparent periods of no commits or with monthly commits number much lower than during other periods. We did this for two of the projects, GCC and GCL. This can be seen on Figure 1 for GCL. The left-hand-side plot shows the total duration. The right-hand-side shows the period of active work, which determined the value of DUR (4.3 years instead of 8.8). Similarly, for GCC we assumed a value of DUR of 11.2 instead of 19.9 years.

[4] `www.dwheeler.com/sloccount/`
[5] `svn.forge.morfeo-project.org/svn/libresoft-tools/cvsanaly`
[6] `data.flossmetrics.org`

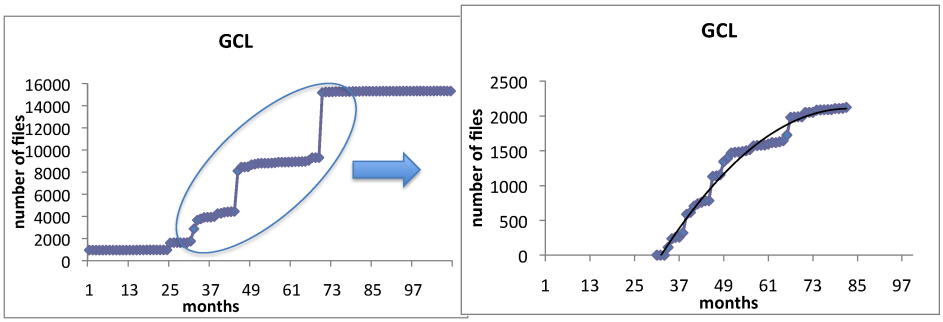

Fig. 1. Trend in $FILES$ (left) and $netFILES$ (right) per month for GCL.

For DEV we counted the number of people making at least one commit since the start of the repository. We computed two variants, one measuring all the contributors, $DEV[100]$, and one measuring the number of developers which provided 80 percent or more of the effort in contributor-months, or $DEV[80]$, in order to exclude contributors who may have made only a few commits.

3 Preliminary results

Initially, we studied the $netFILES$ trends over months. We found a very good fit to linear, quadratic or exponential models, with coefficient of determination[7] (R^2) values ranging between 0.98 to 0.99. Five of the systems had trends that follow linear models, two follow sublinear (quadratic) and three superlinear (2 quadratic and one exponential). The high goodness of fit suggests that size increase in these FLOSS systems take place at a predictable rate. This mixture of different trends is in agreement with previous empirical studies of FLOSS growth trends [5].

As it is common practice in effort estimation models (e.g., COCOMO [2]), we studied the linear correlation between size (measured in $KLOC$ and $FILES$) and $EFFORT$, DUR and DEV. Table 3 shows the parameters of linear models of the form $y = (a * size) + b$ and the corresponding a, b and R^2 values, for the eleven systems and for ten systems (excluding Eclipse) The exclusion of the data of the largest system, Eclipse, following its identification as a possible outlier in some of the scatter plots that we studied, is indicated by an '*' in the attribute's name (e.g. $KLOC^*$). Parameters a and b are not reported when R^2 is less than 0.1 because they are not meaningful. The best models, indicated in bold, are obtained when Eclipse is excluded from the dataset.

Contrary to what was expected, the sole removal of unusual 'jumps' in the monthly growth of the studied systems did not lead to visible improvements in the regression models. It is the exclusion of Eclipse that leads to improvement in the correlation in 13 out of 16 correlations studied. All the best regression results correspond to net

[7] This is a measure of goodness of fit, with values from 0, for a very poor fit, and 1 for a very good fit.

Table 3. Linear regression results - parameters a, b and R^2 values

size in:	$EFFORT$ (a, b, R^2)	DUR (a, b, R^2)	$DEV[100]$ (a, b, R^2)	$DEV[80]$ (a, b, R^2)
$KLOC$	(0.0858, 29.909, 0.339)	(-, - , 0.012)	(0.027, 88.538, 0.101)	(0.0095, 29.57, 0.1331)
$FILES$	(0.0039, 119.58, 0.390)	(-, -, 0.001)	(0.0017, 103.97, 0.219)	(0.0006, 35.142, 0.2845)
$netKLOC$	(0.076, 110.39, 0.326)	(- , - , 0.048)	(0.0287, 106.56, 0.139)	(0.0105, 35.384, 0.196)
$netFILES$	(0.0035, 156.61, 0.313)	(- , - , 8.8E-05)	(0.0016, 119.2, 0.185)	(0.006, 40.066, 0.258)
$KLOC^*$	(0.1327, -53.323, 0.387)	(0.0015, 6.1233, 0.153)	(- , - , 0.088)	(- , - , 0.07)
$FILES^*$	(0.0093, 3123,0.66)	(- , - , 0.06)	(0.004, 62.831, 0.391)	(0.0012, 24.787, 0.367)
$netKLOC^*$	(0.1699, 14.247, 0.499)	(0.0032, 5.4474, **0.525)**	(0.0626, 71.879, 0.196)	(0.0183, 27.401, 0.185)
$netFILES^*$	(0.0139, 51.455, **0.797)**	(- , - , 0.094)	(0.0143, -19.009, **0.565)**	(0.002, 25.998, **0.506)**

size and with Eclipse excluded. The best regression model obtained is the one involving $EFFORT$ as a linear function of $netFILES'$ (R^2 value of 0.797), that is, $EFFORT = 0.0139 * netFILES^* + 51.455$. According to this model, reaching say, 8,000 files would require about $(0.0139 * 8000) + 51.455$ or 163 contributor-years [6]. The six worst regression models correspond to DUR vs size, indicating that FLOSS increase in size at different rates across projects.

With regards to external validity (generalisation), eleven arbitrarily chosen FLOSS constitute a small sample to be able to generalise our conclusions. Regarding threats to internal validity, $EFFORT$ is likely to be an over-estimation of the actual effort. This is so because many FLOSS contributors are volunteers who work part-time rather than full time on the project. One way to improve measuring $EFFORT$ would be to conduct a survey of FLOSS contributors to know better their work patterns and use this knowledge to adjust our measurements. Surveys, however, may require considerable research effort and the willingness of FLOSS contributors to participate. Our $EFFORT$ measurement may be more accurate for FLOSS projects like Eclipse, where a portion of contributors are full-time employees of a sponsoring company (in this case, IBM). Despite all our care, the tools used to extract and analyse the data may contain defects that may affect the results.

4 Conclusion and further work

This paper reports on work in progress in the study of the relationship between size, effort and other attributes for large FLOSS, an issue that does not seem to have been empirically studied. The preliminary results are encouraging and better models may be obtained through refinement and improvement of the measurement methodology. If this were succesful, the models could provide a baseline to study, for example, the possible impact of new or improved processes, methods and tools. The models may also help FLOSS communities in systematic planning of the future evolution of their systems. The best model excludes data from Eclipse. This suggests that independent models for very large systems (greater than 5 million LOC) or for systems based on different technologies (Eclipse was the only Java based system in the sample) are worth exploring.

We have started to compare how well existing cost estimation models based on proprietary projects (in particular, COCOMO [2]) correspond to our FLOSS data [6]. In

order to increase the external validity of the results, we would need to study an additional number of FLOSS projects. The regression models we used are simple. Better results may be achievable by using *robust regression* [7]. Other suitable modelling techniques different than regression [8] could be evaluated against the data. Accuracy may be improved by including other attributes that may impact productivity and duration. One of these is the so-called number of *orphaned lines of code* [9], the portion of the code left behind by contributors who have left a project. Currently, our $FILES$ measurement considers all files in the repository (i.e. code, configuration, data, web pages). It is likely that better results will be achieved by considering code files only. We would also like to exclude any automatically generated files since they will bias the results (e.g., programming productivity may appear higher than it is). We also plan to examine different approaches to extract the outliers from monthly growth trends. In Figure 1 (right) one can identify 'jumps' that were not apparent when looking at the unfiltered data (left). One question is how to define formally what is a 'jump'. In particular, one needs to check manually the version repositories to confirm whether the 'jumps' are actually corresponding to the inclusion of externally generated code or libraries.

Acknowledgements

This contribution has been prepared during the research visits of two of the authors (DI and JFR) to UMH. One of the authors (JFR) acknowledges the Belgian F.R.S.-F.N.R.S. for funding through postdoctoral scholarship 2.4519.05. One of the authors (DI) has been funded in part by the European Commission, under the FLOSSMETRICS (FP6-IST-5-033547) and QUALOSS (FP6-IST-5-033547) projects, and by the Spanish CICyT, project SobreSalto (TIN2007-66172). TM acknowledges partial funding by the *Actions de Recherche Concertées – Ministère de la Communauté française - Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique*.

References

1. Wolverton, R.W.: The cost of developing large-scale software. IEEE Trans. Computers **C-23**(6) (June 1974) 615 – 636
2. Boehm, B.: Software Engineering Economics. Prentice Hall (1981)
3. Jones, C.: Estimating Software Costs - Bringing Realism to Estimating. McGraw Hill (April 2007)
4. Amor, J.J., Robles, G., Gonzalez-Barahona, J.M.: Effort estimation by characterizing developer activity. In: EDSER '06: Proceedings of the 2006 international workshop on economics driven software engineering research, New York, NY, USA, ACM (2006) 3–6
5. Herraiz, I., Robles, G., Gonzalez-Barahona, J.M., Capiluppi, A., Ramil, J.F.: Comparison between SLOCs and number of files as size metrics for software evolution analysis. In: Proc. European Conf. Software Maintenance and Reengineering (CSMR), Bari, Italy (March 2006)
6. Fernandez-Ramil, J., Izquierdo-Cortazar, D., Mens, T.: How much does it take to achieve one megaloc in open source? Submitted for publication (November 2008)
7. Lawrence, K.D., Arthur, J.L.: Robust Regression: Analysis and Applications. CRC Press (1990)
8. MacDonell, S.G., Gray, A.R.: Alternatives to regression models for estimating software projects. In: Proceedings of the IFPUG Fall Conference, Dallas TX, IFPUG. (1996)
9. Izquierdo-Cortazar, D., Robles, G., Ortega, F., Gonzalez-Barahona, J.: Using software archaelogy to measure knowledge loss in software projects due to developer turnover. In: Proceedings of the Hawaii International Conference on System Sciences (HICSS-42), Hawaii, USA, forthcoming (January 2009)

Relationship between Orphaning and Productivity in Evolution and GIMP projects*

Daniel Izquierdo-Cortazar

GSyC/LibreSoft, Universidad Rey Juan Carlos, Móstoles, Spain
dizquierdo@gsyc.es

Abstract. In this paper we try to better understand the metric *orphaning* comparing with some others like regeneration of developers and productivity. By definition, if a developer leaves a project, her source lines of code (SLOC) become *orphaned*. In this study we have focused on two projects with totally different levels of orphaning, Evolution and GIMP, both from the GNOME desktop. We try to understand if high levels of orphaning can be seen as a risky situation in terms of productivity for these projects.
Keywords: libre software, data mining, software evolution

1 Introduction

The turnover in companies and FLOSS projects [1] is a fact. In previous research [2], an approach has been proposed to quantify the impact of that turnover. Such impact is named as *know-how gap* and is measured in number of lines. In other words, we assume this is the knowledge loss when a developer leaves the project.

There are two kinds of knowledge, tacit and explicit. While the first one is not measurable due to its condition of subjective knowledge, the second one remains in the project data sources [3], like mailing lists, bug tracking systems or source code management (SCM in the following). Our assumption is that part of the explicit knowledge is represented in the source code and the unit of knowledge is the line of code. However some other granularity can be taken into account, like functions or files.

The concept of *orphaning* was introduced as a metric to measure that know-how gap. In this paper, we continue the work presented before, trying to go a step ahead and measuring how an abrupt regeneration of developers can affect the orphaning of a project and trying to understand whether high levels of orphaning could entail risky situations for the project, for instance measuring productivity.

* The author would like to thank Dr. Tom Mens from University of Mons-Hainaut (Belgium) and Dr. Juan Fernandez-Ramil from The Open University (United Kingdom) for their comments to improve the current version of the paper and for their invaluable support given during his research stay at University of Mons-Hainaut. This work has been funded in part by the European Commission, under the QUALOSS (FP6-IST-5-033547), FLOSSMETRICS (FP6-IST-5-033547), and by the Spanish CICyT, project SobreSalto (TIN2007-66172).

2 Methodology

The methodology is based on the analysis of source code found in the source SCMs. CVSAnalY and Carnarvon are the tools used in this paper[1]. The first one extracts out of CVS, Subversion or Git repository log and stores it in a MySQL database.

The second one, Carnarvon, extracts the author and the exact date when a line was committed in the system and stores it in a MySQL database.

The selected projects are two, Evolution[2] and GIMP[3]. Evolution is a groupware solution that combines e-mail, calendar, address book and task list managements function. GIMP is a graphics editor.

3 Measurements

The extracted measurements are three. Firstly, orphaning, measured in number of lines. This metric gives the number of *orphaned* lines for a period of time. It represents the number of lines whose committers have not ever committed again after a given date.

It is remarkable that the process of committing in FLOSS projects is irregular. There are periods of intense work and periods where there is no commit for days or even weeks. In order to deal with this, we have defined a year as the period where the lines become orphaned. As an example, if a committer has not made changes since February (less than one year), we can not confirm that her SLOC are orphaned. However, if there is no a change, for a given committer, since February 2006 (more than one year), we can confirm that her SLOC became orphaned after that date.

Secondly, we measure the evolution of the size of the projects in number of SLOC.

And finally, productivity, measured in number of ”handled” files (added, modified and deleted) per committer and month. Also, the productivity results have been filtered, ignoring those committers who are under the 80% of the total number of files handled during the given year. The productivity is measured by month, but the results are presented by year in order to simplify the graphs.

4 Preliminary Results

Figure 1 shows the evolution of SLOC in Evolution and GIMP and also the evolution of orphaned lines detected in each project. Focusing on Evolution, it can be divided in three main parts. One for each big jump in number of orphaned lines. This behaviour shows an abrupt regeneration of developers, where suddenly the core developer left the project, leaving what we named as *know-how gap*. We can see how the know-how gap increases up to 80% of the total SLOC. Regarding the number of SLOC, it is remarkable the period of refactoring where the community carried out two big deletions of lines, even when the number of orphaned lines did not decrease. Thus, the developers just worked on their code previously added and not in the old code (orphaned lines).

[1] `http://forge.morfeo-project.org/projects/libresoft-tools/`
[2] `http://www.gnome.org/projects/evolution/`
[3] `http://www.gimp.org/`

On the opposite, we find GIMP, where a group of main developers left the project at the very beginning, and the number of orphaned lines have continuously decreased during the last years, even when the number of total SLOC is increasing. It is also remarkable that this project also suffered a refactoring process (around 350,000 lines were removed).

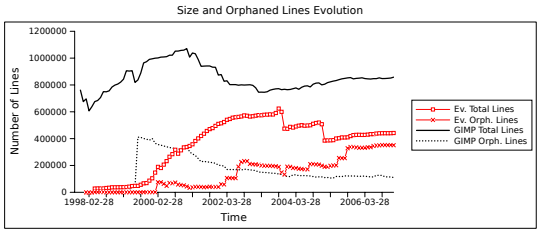

Fig. 1. Evolution of size and orphaned lines in GIMP and Evolution projects.

Figure 2 shows the evolution of the productivity in Evolution and GIMP projects. With regards to Evolution, there is a clear increase during the big jumps detected in the number of orphaned lines. This activity comes from a new set of committers and we can appreciate how it decreases during next years until a new set of developers take the control of the project.

However, GIMP shows a more active productivity. We have detected that the regeneration of developers in this project is not so abrupt. In fact, the core group of developers have remained stable since the first big jump of orphaned lines detected in Figure 1

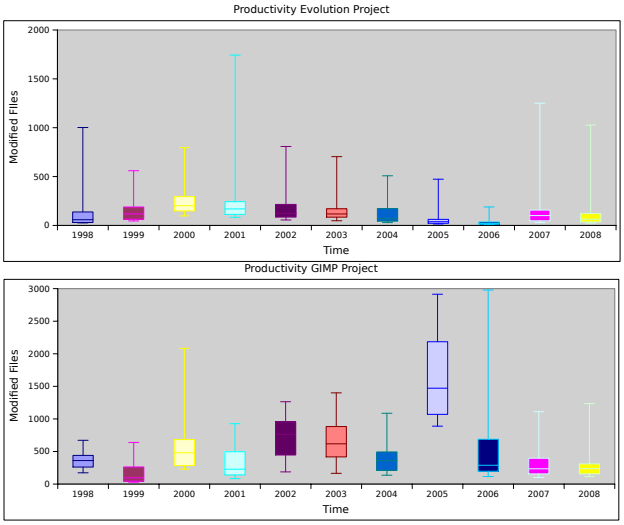

Fig. 2. Distribution of files handled per developer in Evolution and GIMP Projects

5 Conclusions and Further Work

We have not demonstrated a direct relationship between orphaning and productivity, however we think this study can be used as a good basement for future research. A strong regeneration of developers means big jumps on the number of orphaned lines. We have seen how the orphaned code detected in Evolution has not decreased during the stable periods, at least not as faster as in GIMP. It can be seen as a no maintenance activity in that code. The productivity measurements do not show that relationship because they were measured for the whole system, however orphaned areas should be measured in more detail.

Some limitations and further work were addressed in previous work [2], but some extra threats to validity should be added here related to the productivity metric. For instance, the data obtained to calculate productivity take into account the whole repository of code. These repositories contain any kind of file, even no source code file. Thus, it is necessary to focus more on the results on source code having, in this way, more accurate results. Also, it is hard even for really active committers to modify source code in around 3,000 files, this kind of outliers should be measured carefully. For example, automatic generated files or changes in the license should not be included for future research.

Also, studying more FLOSS projects will enrich the results. The two studied systems present a totally different behaviour in the number of orphaned lines. Adding older projects, like GCC, could provide an interesting study of their orphaned lines because of their age (around 20 years since the first commit).

The stability of the core group is also an interesting case of study. GIMP presents a stable core group, but not Evolution. It can be seen as a triumph of the stable core groups, or we can see it as a risky situation for GIMP project. If one of the main developers leaves the project, a big amount of lines will become orphaned.

Finally, it is surprising for us to see how the productivity in Evolution is so irregular. GIMP presents a more active and sometimes regular productivity, at least in number of handled files. Evolution seems to show a strong increase in the productivity when a new set of developers start to work on the project. However it decreases as time goes by until the next set of developers. One of our hypothesis is that in the Evolution new functionality was built over old one, but no maintenance activity was carried out, at least in the orphaned code.

References

1. Robles, G.: Contributor turnover in libre software projects. In: Proceedings of the Second International Conference on Open Source Systems. (2006)
2. Izquierdo-Cortazar, D., Robles, G., Ortega, F., Gonzalez-Barahona, J.M.: Using software archaeology to measure knowledge loss in software projects due to developer turnover. In: Proceedings of the Hawaii International Conference on System Sciences (HICSS-42), Big Island, Hawaii, USA (January 2009)
3. Ward, J., Aurum, A.: Knowledge management in software engineering - describing the process. In: Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04), Melbourne, Australia (April 2004)

Version Control of Graphs

Marcel van Amstel, Mark van den Brand, and Zvezdan Protić

Department of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{M.F.v.Amstel|M.G.J.v.d.Brand|Z.Protic}@tue.nl

Software versioning is an important part of the software development process. A version (of a software artifact) represents a state of an evolving artifact. Versioning of software artifacts is done by assigning unique identifiers to the artifacts that are being versioned. These identifiers are in a form that allows a temporal relation between different versions of an artifact, e.g. version 1.0 is created before version 2.0 etc. Versioning is used in conjunction with revision control. Revision control assumes the existence of a repository and enables efficient storage, manipulation, and retrieval of any version of an artifact in the repository. Software versioning together with revision control allows teams of developers to work on multiple versions of software at the same time, as well as for teams of developers to work on the same version at the same time, and to synchronize (merge) their work when needed.

Data put under version control is usually contained in (text)files [1]. Hence, file is the unit of versioning (UOV) as described by Murta et al. [2]. Software versioning systems have several important features. One of the most important features is the way storage of different versions of a UOV is handled. This is done by storing only the differences between the current and the previous version. In this way it suffices to store only the initial version completely and all other versions as a list of differences with the previous version. In order to describe the differences between units of versioning there should be a way for comparing them. In software versioning systems, the units of versioning (files) are being compared on the level of paragraph (a line of text ending with carriage return). Hence, the unit of comparison (UOC) in software versioning systems is paragraph [2].

Versioning approaches with file as unit of versioning and paragraph as unit of comparison are generic because they handle all file types in the same way. This is not always advantageous. This approach is appropriate for textual documents, where the required UOC is paragraph. It is less appropriate for source code files, where the preferred UOC is statement, when a line of source code contains more than one statement. It is inappropriate for XMI documents representing UML models, where the preferred UOC is a model element, and not a line of text. Figure 1 gives an overview of the unit of version and the unit of comparison in classical software versioning systems for several file types [2].

With the emergence of model driven engineering, there is a shift from developing code to developing models. Since models are becoming the main design artifacts, the problem of model versioning is becoming more apparent. Existing software versioning systems do not provide immediate solutions to this problem. However, in recent years several authors have proposed solutions to the problem of model versioning. Nevertheless, most of the authors focus only on UML models [2] [3] [1] [4]. There are only a

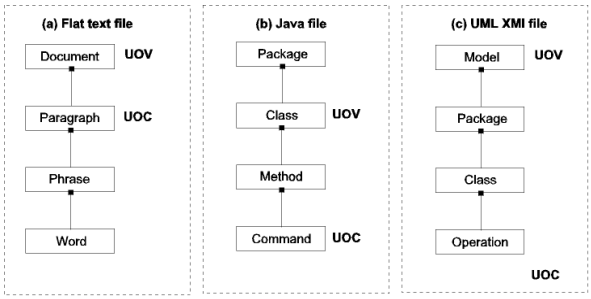

Fig. 1. Units of versioning and comparison for different file types

few approaches in specifying a generic method for versioning models. Rho and Wu [5] present a method for versioning software diagrams. Alanen and Porres [6] present a metamodel independent approach to model versioning. Neither of the two has achieved general public acceptation, most likely because of a lack of a tool that would support those methods.

Our solution to the problem of versioning models is based on the assumption that most models can be transformed into graphs. Therefore, we created a method and system for graph versioning. Figure 2 illustrates our method.

Fig. 2. Schema of a method for putting graphs under version control

This method is based on a bidirectional transformation from graphs to text. The text resulting from the graph-to-text transformation can be put under version control by an existing version control system. The graph to text transformation sets the UOV to a graph, and UOC to a node or an edge. The original graph can be retrieved from the version control system using the text-to-graph transformation.

Next, we propose to use bidirectional transformations between models and graphs. A model can then be put under version control by transforming it into a graph and thereafter into text. Example models include UML models, finite state machine diagrams, flowcharts, Petri-nets, etc. We claim that all models that can be transformed into graphs and back can be versioned using this method.

An example of a class diagram and a possible graph representation of that diagram are depicted in Figure 3. It is easy to give a graph representation of other elements of UML class models, like methods or stereotypes. It is also easy to give a graph representation of features like positioning, color and shape that are not part of the class diagram meta-model, but are introduced by a tool used to create class diagram.

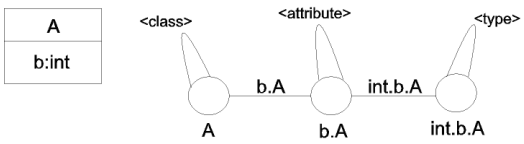

Fig. 3. An example class and its graph representation

References

1. Bartelt, C.: Consistence preserving model merge in collaborative development processes. In: Proceedings of the 2008 international workshop on Comparison and versioning of software models, New York, NY, USA, ACM (2008) 13–18
2. Murta, L., Oliveira, H., Dantas, C., Lopez, L.G., Werner, C.: Odysey-scm: An integrated software configuration managemet infrastructure for uml models. In: Science of Computer Programming, Elsevier (2006) 249–274
3. El-khoury, J.: Model data management: towards a common solution for pdm/scm systems. In: SCM '05: Proceedings of the 12th international workshop on Software configuration management, New York, NY, USA, ACM (2005) 17–32
4. Zündorf, A.: Merging graph-like object structures. In: Proceedings of the Tenth International Workshop on Software Configuration Management. (2001)
5. Rho, J., Wu, C.: An efficient version model of software diagrams. In: APSEC '98: Proceedings of the Fifth Asia Pacific Software Engineering Conference, Washington, DC, USA, IEEE Computer Society (1998) 236
6. Alanen, M., Porres, I.: Difference and union of models. TUCS Technical Report No 527, TUCS Turku Centre for Computer Science (2003)

Towards a general conceptual framework for model inconsistencies

Kim Mens and other members of WP4 work package on ”Consistency Checking and Co-evolution” of the Belgian interuniversity MoVES project on ”Fundamental Issues in Modelling, Verification and Evolution of Software”.

Département d’Ingénierie Informatique
Université catholique de Louvain
Place Sainte Barbe 2,
B-1348 Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be

The problem of handling inconsistencies in and between models is omnipresent in software engineering. It appears at every phase of the software life-cycle, ranging from requirements, analysis, architecture, design and implementation to maintenance and evolution. It deals with a variety of inconsistencies in and between models built with different languages and of different types like requirements models, feature models, use cases or UML design models, software architectures, design guidelines and programming conventions, up to, eventually, program code. In spite of the importance of using models in software development and the need for dealing with inconsistencies in and between them, to our knowledge there exists no single unifying conceptual framework for model inconsistencies that encompasses all these different phases and models and that allows us to focus on what all these approaches for dealing with inconsistencies have in common. In fact, it is almost taken from granted by the inconsistency management community that it is impossible to come up with such a unifying framework, given that inconsistencies vary a lot and require vastly different approaches depending on the software process or language being used and depending on the application domain. Our initial framework circumvents this problem by not focussing on the inconsistency management approach being used, but by define unambiguously and independently of any concrete instantiation to a particular phase or kind of model, the notion of inconsistency. In addition it allows us to identify the potential causes of such inconsistencies. Such a framework would allows us to explain more clearly what certain approaches may have in common and how they vary, and this unique reference frame lets related approaches dealing with inconsistencies learn from each other. We validate and illustrate the framework by providing a concrete instantiation of it for three different application domains: (1) checking inconsistencies when co- evolving source code and structural regularities on that code, (2) inconsistencies between data models and queries over those models (in the context of database schema evolution) and (3) detect inter-model inconsistencies between different UML models describing a same software system from different angles (for example, class and sequence diagrams).

Challenges in Model-Driven Software Evolution

Michael Hoste, Jorge Pinna Puissant, Tom Mens

Université de Mons-Hainaut, Mons, Belgium
{ michael.hoste | jorge.pinnapuissant | tom.mens@umh.ac.be }

Abstract. We report on a research project[1] that started in July 2008, with the aim to scientifically study and advance the state-of-the-art in the use of lightweight formal methods to support the software developer during the process of evolving models while preserving consistency and improving quality. In particular, graph-based and logic-based formalisms will be exploited and combined to provide more generic and more uniform solutions to the stated problem. In this paper we present a number of challenges that need to be addressed to achieve this.

1 Introduction

The use of models in software engineering promises to cope with the intrinsic complexity of software-intensive systems by raising the level of abstraction, and by hiding the accidental complexity of the underlying technology as much as possible. This opens up new possibilities for creating, analysing, manipulating and formally reasoning about systems at a high level of abstraction. Evolution of models can be achieved by relying on sophisticated mechanisms of model transformation. Model transformation techniques and languages enable a wide range of different automated activities such as translation of models (expressed in different modelling languages), generating code from models, model refinement, model synthesis or model extraction, model restructuring etc.

It is generally acknowledged that software that is employed in a real-world environment must be continuously evolved and adapted, else it is doomed to become obsolete due to changes in the operational environment or user requirements (Parnas,1994) (Lehman et al., 1997). On the other hand, any software system needs to satisfy certain well-defined quality criteria related to performance, correctness, security, safety, reliability, and soundness and completeness w.r.t. the problem specification. It is a very challenging task to reconcile these conflicting concerns, in order to develop software that is easy to maintain and evolve, yet continues to satisfy all required quality characteristics. When we look at contemporary support for software evolution at the level of models, however, research results and automated tools for this activity are still in their infancy. There is an urgent need to reconcile the ability to evolve models easily without compromising their overall consistency, and without degrading their quality.

Therefore, we aim to explore scientifically the interaction between model evolution, model quality and model consistency, and the many scientific challenges associated with it. To achieve this, we will resort to formal methods in order to formally

[1] Funded by the *Actions de Recherche Concertées – Ministère de la Communauté française - Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique.*

reason about the artefacts we are dealing with. These formalisms will be sufficiently lightweight and scalable, so as to be applicable into tools that can be applied to large and complex industrial models.

2 Research challenges

To achieve the aforementioned goals, we will tackle the following fundamental research challenges:

Model independence. How can we represent and manipulate different types of models in a uniform way, without needing to change the infrastructure (tools, mechanisms and formalisms) for reasoning about them? Such model independence is of scientific as well as practical importance, because we want our solutions to be sufficiently generic, in order to be applicable beyond mere software models. Indeed, we want to be able to support an as wide range of models as possible.

Language evolution. Not only models evolve, but so do the languages in which the models are expressed, though at a lower pace. In order to ensure that models do not become obsolete because their languages have evolved, we need mechanisms to support the co-evolution between both. In a similar vein, the model transformation languages may evolve in parallel with the model transformations being used, so we also need to support co-evolution at this level.

Model quality. How can we provide a precise definition of model quality? A model can have many different non-functional properties or quality characteristics that may be desirable (e.g., usability, readability, performance and adaptability). It remains an open challenge to identify which qualities are necessary and sufficient for which type of stakeholder, as well as how to specify these qualities formally, and how to relate them to one another.

The next logical question concerns how we can objectively measure, predict and control the quality of models during thier evolution. One possible solution is by resorting to model metrics, the model-level equivalent of software metrics. The challenge here is to define model metrics in such a way that they correlate well with external model quality characteristics.

A more pragmatic way of assessing model quality is by resorting to what we call model smells, being the model-level equivalent of bad smells. Typical model smells have to do with redundancies, ambiguities, inconsistencies, incompleteness, non-adherence to design conventions or standards, abuse of the modelling notation, and so on. The challenge is to come up with a comprehensive and commonly accepted list of model smells, as well as tool support to detect such smells in an automated way.

Model improvement. In order to improve model quality, we will resort to the technique of model refactoring, the model-level equivalent of program refactoring. An important point of attention is the study of the relation between model metrics and model refactoring. In particular, we need to assess to which extent model refactorings affect metric values. A formal specification of model refactoring is required to address these issues.

In a similar vein, we also require a precise understanding of the relation between model smells and model refactoring, in order to be able to suggest, for any given model

smell, appropriate model refactorings that can remove this smell. The other way around, we need to ensure that model refactorings effectively reduce the number of smells.

An important point of attention is the need for a composition mechanism that allows us to reason about composite refactorings in a scaleable way. We also need to study to which extent the formalisms allow us to verify that a given transformation preserves certain properties (e.g. structure-preserving, behaviour-preserving, quality-preserving).

Model inconsistency. In a model-driven development approach, inconsistencies inevitably arise, because a system description is composed of a wide variety of diverse models, some of which are maintained in parallel, and most of which are subject to continuous evolution. Therefore, there is a need to formally define the various types of model inconsistencies in a uniform framework, and to resort to formally founded techniques to detect and resolve these model inconsistencies. A prerequisite for doing so is to provide traceability mechanisms, by making explicit the dependencies between models.

Conflict analysis. Another important challenge has to do with the ability to cope with conflicting goals. During model evolution, trade-offs need to be made all the time:

- When trying to improve model quality, different quality goals may be in contradiction with each other. For example, optimising the understandability of a model may go at the expense of its maintainability.
- In the context of inconsistency management, inconsistency resolution strategies may be in mutual conflict.
- In the context of model refactoring, a given model smell may be resolved in various ways, by applying different model refactorings. Vice versa, a given model refactoring may simultaneously remove multiple model smells, but may also introduce new smells.

It should be clear from the discussion above that uniform formal support for analysing and resolving conflicts during model transformation is needed.

Collaborative modelling. Another important challenge in model-driven software evolution is how to cope with models that evolve in a distributed collaborative setting? This naturally leads to a whole range of problems that need to be addressed, such as the need for model differencing, model versioning, model merging or model integration, model synchronisation, and so on. In our research project, we will not study the problem of collaborative modelling in detail. The solutions that we will provide for the other challenges, however, should be sufficiently easy to combine with solutions to this problem of collaborative modelling.

Choice of formalism. Various formalisms may be suited to specify and reason about model quality and model consistency in presence of continually evolving models. Each type of formalism has its own advantages (in terms of the formal properties they can express), but it is often very difficult to combine them into a uniform formal framework. Graph-transformation and logic-based approaches seem to be the most likely candidates.

Graphs are an obvious choice for representing models, since most types of models have an intrinsic graph-based structure. As a direct consequence, it makes sense to express model evolution by resorting to graph transformation approaches. Many useful theoretical properties exist, such as results about parallellism, confluence, termination

and critical pair analysis. Model independence (i.e., the ability to apply it to a wide range models) can be achieved by resorting to a technique similar to metamodelling in model-driven engineering: each graph and graph transformation needs to conform to the constraints imposed by a type graph, used to describe the syntax of the language.

If we want to reason about the behaviour of software, logic-based formalisms seem to be the most interesting choice. Their declarative or descriptive semantics allow for a more or less direct representation of the behaviour of software artefacts.

Scaleability and incrementality. An important practical aspect of our research will be the ability to provide tool support that is scalable to large and complex models. This scalability is essential in order to allow us, on a medium to long term, to transfer our research results to industrial practice by integrating them into commercial modelling environnements and by validating them on industrial models. Obviously, this requirement imposes important restrictions on the underlying formalisms to be used. As an example, consider existing approaches to formal verification and model checking, typically based on some variant of temporal logics. Their main problem is that they only allow to verify properties on a model in its entirety. Even a smal incremental change to this model typically requires reverification of the proven properties for the entire model. With an incremental approach, the effort needed to reverify a given property would become proportional to the change made to the model. It is therefore essential to find truly incremental techniques, in order to close the gap between formal techniques and pragmatic software development approaches, which are inherently evolutionary in nature.

3 Conclusions

Clearly, most of the challenges identified above interact and overlap. Therefore, they cannot be addressed in isolation. For example, formal solutions that allow us to specify and perform model refactorings, will have to be directly linked to solutions addressing model quality and model consistency, since the main goal of applying a model refactoring is to improve model quality in a well-defined way, while preserving the overall model consistency. Hence, whatever technique and formalism that we come up with needs to guarantee that this is actually the case.

The expected benefits of our research for software-producing and software-consuming companies are obvious. In order to face the increasing complexity of software intensive systems, model-driven engineering technology is rapidly gaining momentum in industry. Without integrated support for evolution of software models, this technology will not be able to deliver its promises. Our formally-founded research aims to advance the state-of-the-art in industrial practice, and to enable the development of tools for improving the quality of models.

System Evolution by Migration Coordination

Suzana Andova[2], Luuk Groenewegen[1], and Erik de Vink[2] ⋆

[1] FaST Group, LIACS, Leiden University
[2] Formal Methods Group, Department of Mathematics and Computer Science
Eindhoven University of Technology

1 Introduction

Collaborations between components can be modeled in the coordination language Paradigm [3]. A collaboration solution is specified by loosely coupling component dynamics to a protocol via their roles. Not only regular, foreseen collaboration can be specified, originally unforeseen collaboration can be modeled too [4]. To explain how, we first look very briefly at Paradigm's regular coordination specification.

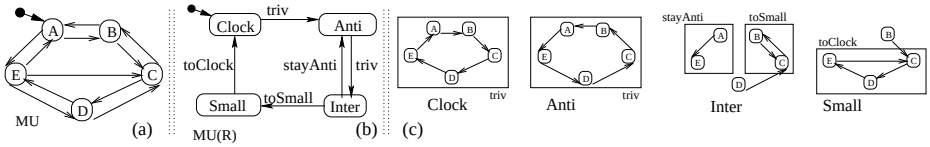

Fig. 1. Example component dynamics, role dynamics by constraints.

Component dynamics are expressed by state-transition diagrams (STDs), see Figure 1(a) for a mock-up STD *MU* in UML style. *MU* contributes to a collaboration via a role $MU(R)$. Figure 1(b) specifies $MU(R)$ through a different STD, whose states are so-called phases of *MU*: temporarily valid, dynamic constraints imposed on *MU*. The figure mentions four such phases, *Clock*, *Anti*, *Inter* and *Small*. Figure 1(c) couples *MU* and $MU(R)$. It specifies each phase as part of *MU*, additionally decorated with one or more polygons grouping some states of a phase. Polygons visualize so-called traps: a trap, once entered, cannot be left as long as the phase remains the valid constraint. A trap having been entered, serves as a guard for a phase change. Therefore, traps label transitions in a role STD, cf. Figure 1(b).

Single steps from different roles, are synchronized into one protocol step. A protocol step can be coupled to one detailed step of a so-called manager component, driving the protocol. Meanwhile, local variables can be updated. It is through a consistency rule, Paradigm specifies a protocol step: (i) at the left-hand side of a $*$ the one, driving manager step is given, if relevant; (ii) the right-hand side lists the role steps being synchronized; (iii) optionally, a change clause [2] can be given updating variables, e.g. one containing the current set of consistency rules. For example, a consistency rule without change clause,

$$MU_2\colon A \to B \;*\; MU_1(R)\colon \mathit{Clock} \overset{\mathit{triv}}{\to} \mathit{Anti},\; MU_3(R)\colon \mathit{Inter} \overset{\mathit{toSmall}}{\to} \mathit{Small}$$

⋆ Corresponding author, e-mail evink@win.tue.nl.

where a manager step of MU_2 is coupled to the swapping of MU_1 from circling clockwise to anti-clock-wise and swapping MU_3 from intermediate inspection into circling on a smaller scale.

2 Migration by constraint manipulation

For modeling unforeseen change, the special component *McPal* is added to a Paradigm model. *McPal* coordinates the migration towards the new way of working, by explicitly driving an unforeseen protocol. During the original, stable collaboration stage of the running Paradigm model, *McPal* is stand-by only, not influencing the rest of the model at all. This is *McPal*'s hibernated form. But, by being there, *McPal* provides the means for preparing the migration as well as for guiding the execution accordingly. To that aim, connections between *McPal* and the rest of the model are in place, realizing rudimentary interfacing for later purposes: in Paradigm terms, an *Evol* role per component. As soon as, via *McPal*, the new way of working together with migration towards it, have been developed and have been installed as an extension of the original model, *McPal* starts coordinating the migration. Its own migration begins, the migration of the others is started thereafter. Finishing migration is done in reversed order. The others are explicitly left to their new stable collaboration phase before *McPal* ceases to influence the others. As a last step, *McPal* shrinks the recently extended model, by removing model fragments no longer needed, keeping the new model only.

It is stressed, migration is on-the-fly. New behaviour per component just emerges in the ongoing execution. Note that no quiescence of components is needed. Additionally, *McPal*'s way of working is pattern-like, as *McPal* can be reused afterward for yet another unforeseen migration.

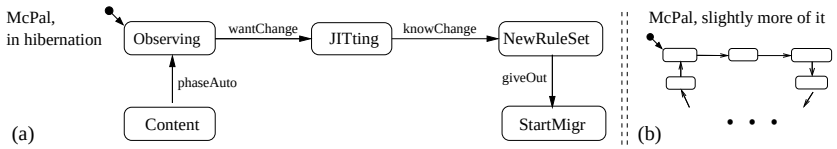

Fig. 2. *McPal*, its hibernated form.

Figure 2(a) visualizes *McPal*'s detailed dynamics in its hibernated form only. In starting state *Observing*, *McPal* is doing nothing in particular, but it can observe, that something should change. State *JITting* is where just-in-time foreseeing and modeling of such a concrete change occurs. The extended model then is available in state *NewRuleSet*. Thus, upon leaving *NewRuleSet* for state *StartMigr*, *McPal* extends its hibernated form with originally unknown dynamics for coordinating the migration. Such an extension is suggested in Figure 2(b).

Figure 3(a) visualizes the ingredients for *McPal*'s role *Evol*. *McPal*'s hibernated form returns here as the phase *Stat*. The other phase *Migr* represents *McPal*'s coordinating a once-only migration. Figure 3(b) visualizes the role STD *McPal*(*Evol*). It says, *McPal*'s hibernation constraint is replaced by the migration constraint, after entering trap *ready* (i.e. once state *NewRuleSet* has been reached). Note, the originally

unforeseen migration dynamics are known by then indeed. Similarly, the hibernation constraint is being re-installed after trap *migrDone* has been entered. So, by returning to starting state *Observing* all model fragments obsolete by then, can be safely removed, including the phase *Migr* of *McPal*. Then, the new stable model is in execution, with *McPal* present in its original, hibernated form.

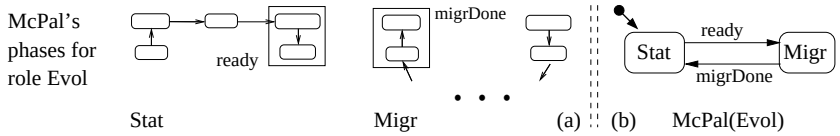

Fig. 3. *McPal*, its phases and global process.

In the style of a horizontal UML activity diagram, Figure 4(a) gives a small part of the coupling between *McPal* and $McPal(Evol)$. Regarding *McPal*, the Paradigm model has initially the following two consistency rules, specifying *McPal*'s first two steps only, the first one without any further coupling.

$$McPal\colon Observing \overset{wantChange}{\rightarrow} JITting$$
$$McPal\colon JITting \overset{knowChange}{\rightarrow} NewRuleSet \ * \ McPal\,[\,Crs := Crs + Crs_{migr} + Crs_{toBe}\,]$$

In the second step from *JITting* to *NewRuleSet*, via a so-called change clause, the set of consistency rules *Crs* for the original stable collaboration is extended with the rules Crs_{migr} for the migration and with the rules Crs_{toBe} for the new, stable collaboration to migrate to. In particular, apart from all other migration coordination details, *McPal* obtains two new consistency rules:

$$McPal\colon NewRuleSet \overset{giveOut}{\rightarrow} StartMigr \ * \ McPal(Evol)\colon Stat \overset{ready}{\rightarrow} Migr$$
$$McPal\colon Content \overset{phaseAuto}{\rightarrow} Observing \ *$$
$$McPal(Evol)\colon Migr \overset{migrDone}{\rightarrow} Stat, \ McPal\,[\,Crs := Crs_{toBe}\,]$$

The first rule says, on the basis of having entered trap *ready*, the phase change from *Stat* to *Migr* can be made, coupled to *McPal*'s transition from state *NewRuleSet* to *StartMigr*. Figure 4(a) expresses this through the left 'lightning' step. As the last migration step, after having phased out dynamics no longer needed for the other components and eventually having entered trap *migrDone* of its phase *Migr*, *McPal* makes its role $McPal(Evol)$ return from *Migr* to *Stat* by making the (coupled) step from state *Content* to *Observing*. Then, also the rule set *Crs* is reduced to Crs_{toBe}, by means of a change clause. See the right 'lightning' in Figure 4(a). Once returned in state *Observing*, *McPal* is in hibernation again, ready for a next migration.

Figure 4(b) suggests how *McPal*, by doing steps between state *StartMigr* and *Content*, may guide other components. Here, one *MU* component migrates from its complete, old dynamics Ph_1 to originally unforeseen dynamics Ph_2, via two intermediate phases $Migr_1$ and $Migr_2$. First, old dynamics is interrupted at trap *triv*. Second, the dynamics

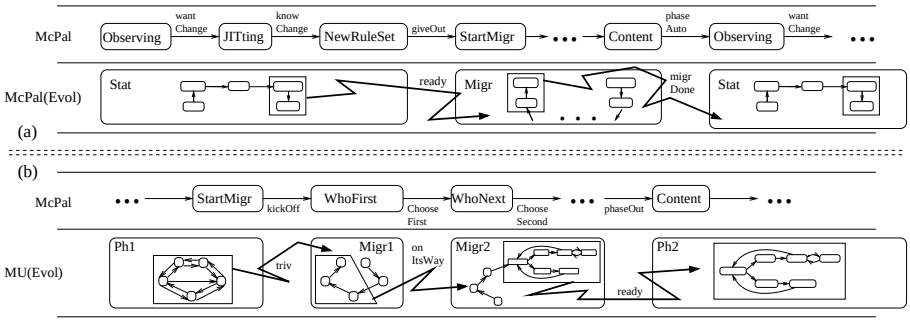

Fig. 4. Migration coordination as constraint manipulation.

is extended after trap *onItsWay* has been entered. Third, finally, the extended dynamics is restricted to that of Ph_2, after trap *ready* has been entered. All this occurs during *McPal*'s migration phase *Migr.*

3 Conclusion

We have sketched how system evolution can be modeled in Paradigm using the migration pattern of *McPal.* New intermediate migration behaviour as well as new target behaviour is added to the relevant components. By restricting the original way of working, components are steered by *McPal* towards a final, stable stage of execution. After removing obsolete model fragments, *McPal* returns to its so-called hibernated form, waiting for a new migration to coordinate.

Paradigm helps structuring software architectures, high-lighting the collaborations that are relevant for separate issues. A prototype environment is reported in [6]. Recently, in [1], a translation of Paradigm into the process algebra ACP is described. This paves the way to state-of-the-art modelchecking using the mCRL2 toolkit [7] developed in Eindhoven, providing support for the verification of invariants and progress properties in Paradigm. Future work is devoted to quantitative analysis of migration, in particular timeliness and performance, envisioning a novel perspective on system migration and evolution. In addition, Paradigm's concept of JIT modeling facilitates that performance triggers *McPal* to update, on-the-fly, the current constraints. Note, Paradigm's constraint handling can be expressed in other languages too, e.g., the UML and ArchiMate.

References

1. S. Andova, L.P.J. Groenewegen, and E.P. de Vink. Dynamic consistency in process algebra: From Paradigm to ACP. In *Proc. FOCLASA'08.* ENTCS, to appear. 19pp.
2. L. Groenewegen, N. van Kampenhout, and E. de Vink. Delegation modeling with Paradigm. In *Proc. Coordination 2005,* pages 94–108. LNCS 3454, 2005.
3. L. Groenewegen and E. de Vink. Operational semantics for coordination in Paradigm. In *Proc. Coordination 2002,* pages 191–206. LNCS 2315, 2002.

4. L. Groenewegen and E. de Vink. Evolution-on-the-fly with Paradigm. In *Proc. Coordination 2006*, pages 97–112. LNCS 4038, 2006.
5. L.P.J. Groenewegen and E.P. de Vink. Dynamic system adaptation by constraint orchestration. Technical Report CSR 08/29, TU/e, 2008. `CoRR abs/0811.3492`.
6. A.W. Stam. *ParADE – a Conceptual Framework for Software Component Interaction*. PhD thesis, LIACS, Leiden University, 2009. Forthcoming.
7. `http://www.mcrl2.org`.

Exploring Source-Code using Visualized Program Queries

Johan Brichau

Département d'Ingénierie Informatique
Université catholique de Louvain
Place Sainte Barbe 2,
B-1348 Louvain-la-Neuve, Belgium
johan.brichau@uclouvain.be

The growing amount of program query languages - of which ASTLog, SOUL, JQuery, CodeQuest and PQL are only some examples - is testament to the significant momentum on the investigation of a program's structure and/or behaviour by means of user-defined queries. Such queries serve the identification of code exhibiting features of interest which range from application-specific coding conventions over refactoring opportunities and design patterns to run-time errors. Similarly, software visualisations such as Codecrawler, Mondrian, Codecity, Chronia, X-Ray, NDepend, Seesoft, etc... are becoming increasingly popular for exposing features of interest in the implementation of a software application. This presentation will demonstrate a prototype that combines program queries and software visualisations to explore features of interest in source code. In particular, we present a composition of the SOUL program-query language and the Mondrian scripting visualisations that combines logic-based user-defined program queries with the visual feedback of polymetric views. Using this tool, developers can write expressive logic program queries to detect conventions, design patterns, bad smells, etc... in Smalltalk or Java source code and present a query's results succinctly using a comprehensive software visualization. The tool is part of the ongoing work in the context of the IntensiVE tool suite, that focuses on the verification of design structure in the implementation of evolving software applications. The screenshot below shows a (simple) query that finds Java classes following a naming convention (prefixed 'AG') and the interface that they implement. The found classes and their interfaces are shown using red and green colors respectively in a modular system complexity view, exposing that these classes and interfaces are organised in two different hierarchies.

Visual Analytics for Understanding the Evolution of Large Software Projects

Alexandru Telea[1] and Lucian Voinea[2]

[1] Institute for Math. and Computer Science, University of Groningen, the Netherlands
[2] SolidSource BV, Eindhoven, the Netherlands

Abstract. We present how a combination of static source code analysis, repository analysis, and visualization techniques has been used to effectively get and communicate insight in the development and project management problems of a large industrial code base. This study is an example of how visual analytics can be effectively applied to answer maintenance questions in the software industry.

1 Introduction

Industrial software projects encounter bottlenecks due to many factors: improper architectures, exploding code size, bad coding style, or suboptimal team structure. Understanding the causes of such problems helps taking corrective measures for ongoing projects or choosing better development and management strategies for new projects.

We present here a combination of static data analysis and visualization tools used to explore the causes of development and maintenance problems in an industrial project. The uncovered facts helped developers to understand the causes of past problems, validate earlier suspicions, and assisted them in deciding further development. Our solution fits into the emerging *visual analytics* discipline, as it uses information visualization to support the analytical reasoning about data mined from the project evolution.

2 Software Project Description

The studied software was developed in the embedded industry by three teams located in Western Europe, Eastern Europe, and India. All code is written in Keil C166, a special C dialect with constructs that closely supports hardware-related operations [2]. The development took six years (2002-2008) and yielded 3.5 MLOC of source code (1881 files) and 1 MLOC of headers (2454 files) in 15 releases. In the last 2 years, it was noticed that the project could not be completed on schedule, within budget, and that new features were hard to introduce. The team leaders were not sure what went wrong, so an investigation was performed at the end.

3 Analysis Method

The investigation had two parts: a process analysis and a product (code) analysis. We describe here only the latter. We analyzed the source code, stored in a Source Control

Management (SCM) system, using a C/C++ analyzer able to handle incorrect and incomplete code by using a flexible GLR grammar [3], and a tool to mine data from SCM systems. We were able to easily modify this grammar (within two work days) to make it accept the Kyle C dialect. The analyzer creates a fact database that contains static constructs *e.g.* functions and dependencies, and several software metrics.

We applied the above process to all files in all 15 releases. Given the high speed of the analyzer, this took under 10 minutes on a standard Linux PC. After data extraction, we used several visualizations to make sense of the extracted facts. Each visualization looks at different aspects of the code, so their combination aims to correlate these aspects in one coherent picture. The visualizations (presented next) were interactively shown to the developers, who were asked to comment on the findings, their relevance, and their usefulness for understanding the actual causes of the development problems.

3.1 Modification Request Analysis

Figure 1 shows two images of the project structure, depicted as a three-level treemap (packages, modules and files). Treemaps have been used in many software visualization applications to show metrics over structure [1]. The smallest rectangles are files, colored by various metrics, and scaled by file size (LOC)[3]. The left image shows the number of modification requests (MRs) per file. Files with more than 30 MRs (red) appear spread over all packages. The right image shows the same structure, colored by team identity. We see that most high-MR files (red in the left image) are developed by Team A (red in the right image), which is located in India. As this team had communication issues with the other two teams, a better work division may be to reassign Team A to work on a single, low-MR-rate, package.

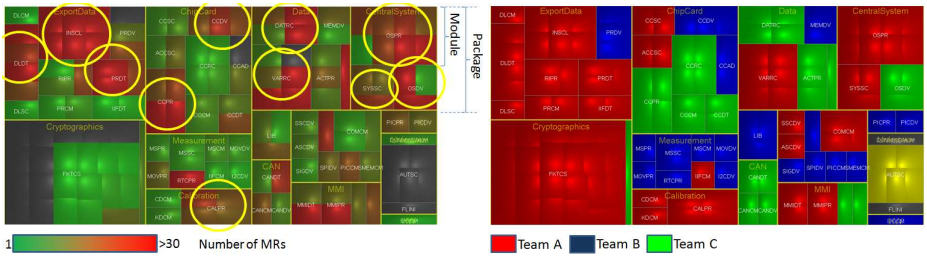

Fig. 1. Team assessment: number of MRs (left) and team structure (right)

We further analyzed the MR records. Figure 2 left shows the MR distribution over project and time. Files are drawn as gray horizontal pixel lines, stacked in creation order from bottom to top. The x axis shows time (years 2002 to 2008). Red dots show the location (file, time) of the MRs. We see that less than 15% of the files have been created in the second project half, but most MRs in this half address *older* files, so the late work mainly tried to fix old requirements. The right image supports this hypothesis:

[3] We recommend viewing this paper in full color

each horizontal bar indicates the number of file changes related to one MR range (MRs are functionally grouped per range), the x axis shows again time. We see that older MRs (at top) have large activity spreads over time. For example, in mid-2008, developers still try to address MRs introduced in 2005-2006. Clearly, new features are hard to introduce if most work has to deal with old MRs.

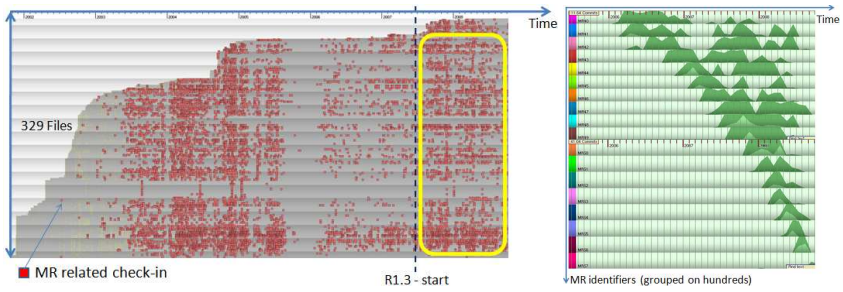

Fig. 2. MR evolution per file (left) and per range of MRs (right)

Figure 3 shows the MRs versus project structure. The left image shows MR criticality (green=low, red=high). We see a correlation with the MR count and team distribution (Fig. 1): many critical MRs are assigned to Team A, which had communication problems. The right image indicates the average closure time of a MR: critical MRs, involving Team A, took long to close. This partially explains the encountered delays and further supports the idea of reassigning critical MRs to other teams.

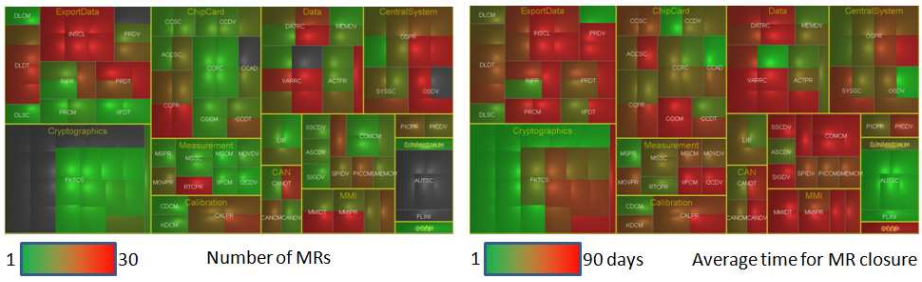

Fig. 3. MR criticality: MRs vs project structure (left); MR closure time (right)

3.2 Structural Analysis

We next analyzed the evolution in time of several software metrics (see Fig. **??**). The graphs show a relatively low increase of project size (functions and function calls) and, roughly, and overall stable dependency count (fan-in) and code complexity. The fan-out and number of function calls increases more visibly. Hence, we do not think that maintenance problems were caused by an explosion of code size or complexity, as is

the case in other projects. Additional analyses such as call and dependency graphs (not shown here) confirmed, indeed, that the system architecture was already well-defined and finalized in the first project half, and that the second half did barely change it.

4 Conclusions

Correlating the MR analysis with the static code analysis, our overall conclusion is that the most work in this project was spent in finalizing early requirements. Correlating with the developer interviews, this seems to be mainly caused by a suboptimal allocation of teams to requirements and packages. The project owners stated that they found this type of analysis very useful from several perspectives: supporting earlier suspicions, providing concrete measures for assessing the project evolution, presenting these measures in a simple and easy way for discussion, and supporting further project management decisions. An attractive element was the short duration of the analysis: the entire process lasted three days, including the developer interviews, code analysis, and results discussion. Also, the stakeholders stated that using such types of analyses continuously, and from the beginning of, new projects is of added value in early detection and discussion of problems.

References

1. M. Balzer and O. Deussen. Voronoi treemaps for the visualization of software metrics. In *Proc. ACM Softvis*, pages 165–172, 2005.
2. Keil, Inc. The Keil C166 compiler. 2008. `http://www.keil.com`.
3. A. Telea and L. Voinea. An interactive reverse engineering environment for large-scale C++ code. In *Proc. ACM SoftVis*, pages 67–76, 2008.

Extraction of State Machines of Legacy C code with Cpp2XMI

Mark van den Brand, Alexander Serebrenik, and Dennie van Zeeland

Technical University Eindhoven, Department of Mathematics and Computer Science,
Den Dolech 2, NL-5612 AZ Eindhoven, The Netherlands
m.g.j.v.d.brand@tue.nl, a.serebrenik@tue.nl,
d.h.a.v.zeeland@student.tue.nl

Introduction Analysis of legacy code is often focussed on extracting either metrics or relations, e.g. call relations or structure relations. For object-oriented programs, e.g. Java or C++ code, such relations are commonly represented as UML diagrams: e.g., such tools as Columbus [1] and Cpp2XMI [2] are capable of extracting from the C++ code UML class, and UML class, sequence and activity diagrams, respectively.

New challenges in UML diagram extraction arise when a) additional UML diagrams and b) non-object-oriented programs are considered. In this paper we present an ongoing work on extracting state machines from the legacy C code, motivated by the popularity of state machine models in embedded software [3]. To validate the approach we consider an approximately ten-years old embedded system provided by the industrial partner. The system lacks up-to-date documentation and is reportedly hard to maintain.

Approach We start by observing that in their simplest form UML state machines contain nothing but states and transitions connecting states, such that transitions are associated with events and guards. At each moment of time the system can be in one and only one of the states. When an event occurs the system should check whether the guard is satisfied, and, should this be the case, move to the subsequent state. Observe, that implementing a state machine behaviour involves, therefore, a three-phase decision making:

- What is the current state of the system?
- What is the most recent event to be handled?
- Is the guard satisfied?

Based on this simple observation, our approach consists in looking for *nested-choice patterns*, such as "`if` within `if`" or "`switch` within `switch`". As guards can be omitted we require the nesting to be at least two. As we do not aim to discover all possible state-machines present in the code, validation of the approach will consist in applying in the case study and comparing the state-machines detected with the results expected by the domain experts.

Implementation We have chosen to implement the approach based on the Cpp2XMI tool set [2]. Since Cpp2XMI was designed for reverse engineering C++, we first had to adapt the tool for C. Second, we added a number of new filters to detect the nested-choice patterns in the abstract syntax trees. Finally, we had to extend the visualisation component to provide for state machine visualisation.

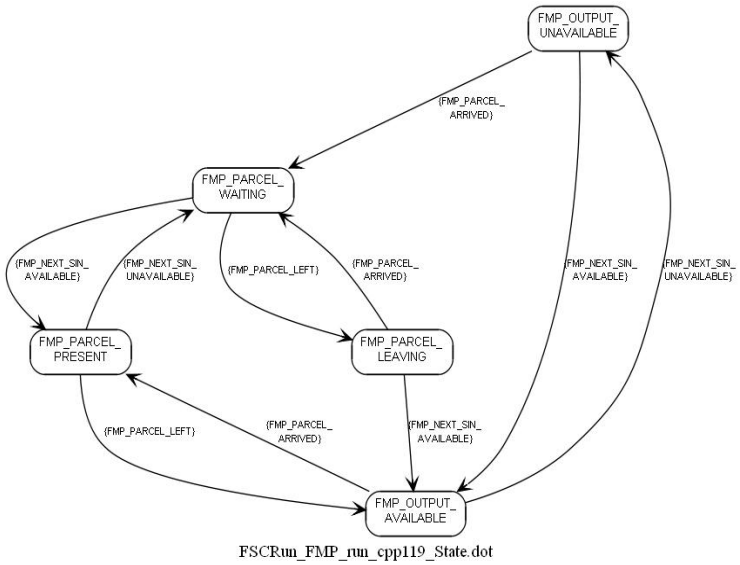

Fig. 1. A state machine discovered.

Case study As the case study we consider an approximately ten-year old system, developed for controlling material handling components, such as conveyer belts, sensors, sorters, etc. Up-to-date documentation is missing and the code is reportedly hard to maintain. While a re-implementation of the system is considered by the company, understanding the current functionality is still a necessity.

It turned out that the original software engineers have quite consistently used `switch` statements within `switch` statements to model the state machines. Therefore, already the first version of the implementation based solely on the "`switch` within `switch`" pattern produced a number of relevant state machines.

At the moment more than forty state machines have been extracted from the code. The size of the extracted state machines varied from 4 states up to 25 states. One of the extracted state machines is shown on Figure 1, the transitions are decorated with conditional events. All the machines extracted were presented to the (software) engineers of the company and their correctness as well as importance were confirmed by them.

Conclusions and future work. In this abstract we presented an ongoing effort on extracting UML state machines from legacy non-object-oriented code. We have observed that UML state machines are useful for the developers and maintainers, and that they can be derived automatically even from a non-object-oriented code. The approach proved to be very successful in the case study and, is in general, promising. As the future work we consider:

- including the "`switch` within `if`" and "`if` within `switch`" patterns;
- analysing the extracted state machines for overlap;
- combining the extracted state machines to nested state machines.

References

1. Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus - reverse engineering tool and schema for c++. In *ICSM*, pages 172–181. IEEE Computer Society, 2002.
2. E. Korshunova, M. Petkovic, M. G. J. van den Brand, and M. R. Mousavi. Cpp2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code. In *WCRE*, pages 297–298. IEEE Computer Society, 2006.
3. Jürgen Mottok, Frank Schiller, Thomas Völkl, and Thomas Zeitler. A concept for a safe realization of a state machine in embedded automotive applications. In Francesca Saglietti and Norbert Oster, editors, *SAFECOMP*, volume 4680 of *Lecture Notes in Computer Science*, pages 283–288. Springer, 2007.

Using graph transformation to evolve software architectures

Dalila Tamzalit[1,2] and Tom Mens[1]

[1] Université de Mons-Hainaut, Mons, Belgium
tom.mens@umh.ac.be
[2] Université de Nantes, France
Dalila.Tamzalit@univ-nantes.fr

Abstract. In our research, we explore the problem of how to evolve software architectures in a disciplined way. The main force of software architectures is their ability to describe software systems at a high level of abstraction, ignoring code details. This is typically ensured by specifying them in a formal way using an Architecture Description Language (ADL). Unfortunately, these ADLs have not found widespread adoption in industry. UML, on the other hand, is a de facto industry standard for software modeling, and also includes a way to specify software architectures. Unfortunately , seen as an ADL, UML is much more ambiguous and informal than the "traditional" ADLs. In addition, an ADL must determine not only how the architecture should be specified, but it should also be used to guide its evolution.
The goal of our research is to provide a solution to the above problem. We formalise the ADL UML 2 using the theory of graph transformation. This allows us to specify the structure and behaviour of an architecture, to impose architectural styles constraining the architecture, and to specify and execute typical architectural evolution scenarios. More in particular, we focus on two types of architectural evolutions: structural and behavioural refactorings. The formalism of graph transformation allows us to verify whether such refactorings preserve the constraints imposed by the ADL and existing architectural styles. As a proof of concept, we have performed experiments using the graph transformation engine AGG.

1 Introduction

Software architectures become inescapable for specifying various and complex software systems. They offer a high abstraction level for describing software systems, by defining their global architecture, thanks to Architecture Description Languages (ADLs). These ADLs must *control and guide architectural evolutions* [1], [2]. Over the past ten years, formal or less formal Architecture Description Languages (ADLs) and supporting methods and tools have been proposed. The most important ones are: Rapide [3], ACME [4] , [5], Wright [6], C2SADEL [7], Darwin [8] and UML 2 [11].

Regarding UML 2, even if the proposed concepts for software architecting remain awkward, there is a wide acceptance (in both industry and academia) to consider and use UML 2 as an ADL. We explore how to evolve UML 2.x software architectures thanks to graph transformations.

2 UML 2.x architecture evolutions with graph transformations

2.1 Specifications

We focus on the specifications only of the main concepts of the ADL UML 2: a *component* has one or several *ports*. Each port can have several *provided/required interfaces* as interacting points and representing *provided/required operations* (or *services*). In order to guide and achieve architectural evolutions properly, we also consider architectural constraints and the behaviour of components. All constraints that must be fulfilled and that guide any evolution scenario represent an *architectural style* [14]. We rely on the graph transformation tool AGG [12] to specify the dedicated type graph (figure 1): (i) *concepts*: using named and attributed nodes and edges, (ii) *constraints*: using constraints and multiplicities, (iii) *behaviour of a component*: using a visual variant of input-output automata [13], inspired by [9].

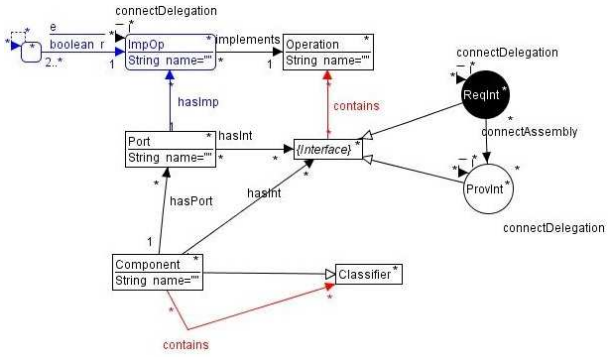

Fig. 1. Type Graph of the main architectural concepts of UML 2

Let us focus on the behavioural automaton. To represent a *component behaviour*, we use the *ImpOp* node as the implementation of exactly one operation. The automaton behaviour is an directed graph of *states* (unlabeled rounded rectangles). *Traces* between states are represented by dotted edges.

Fig. 2. The UML 2 Booking component with its provided interface *Booking* and required interface *Book* and its ports: *p1*, *p2* and *p3*.

2.2 Evolution scenarios on an example

Figure 3 represents a flight&hotel booking component (the whole architecture is not presented here) as a graph that conforms to the type graph of figure 1. Thanks to this component, we can book a flight or a hotel or both, and cancel existing bookings. The UML 2 Booking component of this architecture is represented in figure 2.

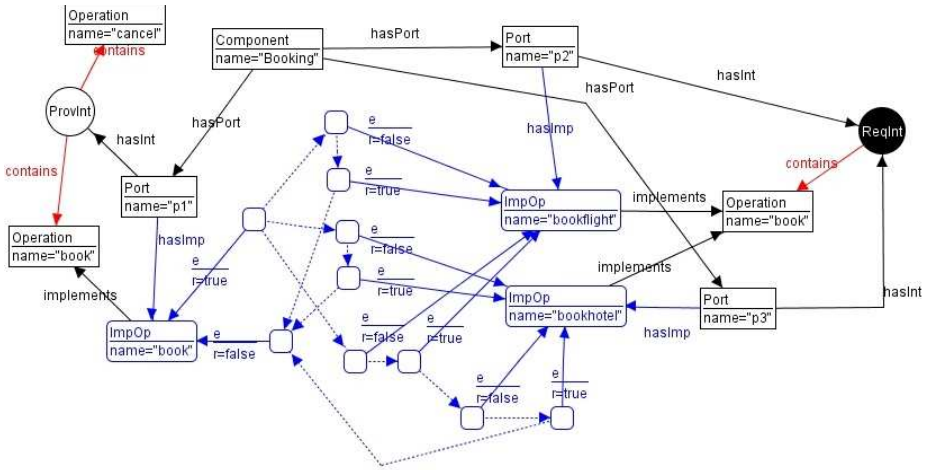

Fig. 3. Booking architecture represented as a graph conforming the type graph in figure 1

We consider two evolution scenarios for this *Booking* component: refactoring the Booking component by splitting a required interface and refactoring by creating a subcomponent of the Booking component. The subcomponent provides the complex behaviour of booking both hotels and flights, leaving simple bookings to the containing component. The result of this last refactoring is presented in figure 4. The graph transformations that formally express this refactoring are left out, due to space considerations.

3 Conclusions and future work

We have outlined the contributions of graph transformations to: specify an ADL such as UML 2, attach constraints and express component behaviour, but also to guide some refactoring situations. Another important contribution is the benefits of analysing formal properties of graph transformations, like causal (sequential) dependencies between transformations and conflicting (mutually exclusive) transformations [15].

As a future work, we aim to define some interesting *behavioural properties* such as: checking if each node in the automaton is reachable; checking if all possible paths (corresponding to different scenarios) in the automaton are complete; checking if, for each implementation of a provided operation: (a) there is always exactly one event

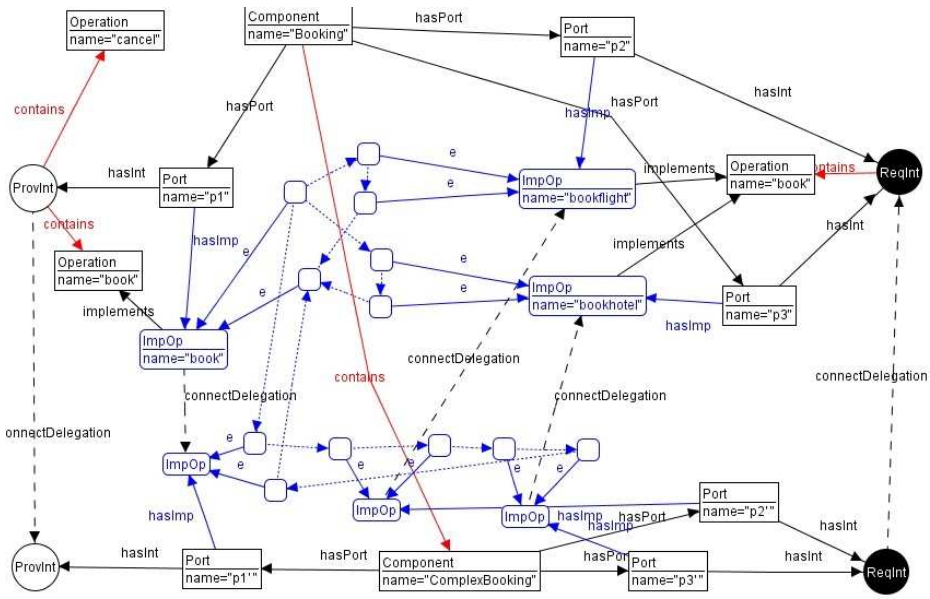

Fig. 4. Refactoring the Booking component and its behaviour by creating a subcomponent.

that starts the automaton, (b) there is always exactly one event that ends the automaton. Another future work is to deal with architectural styles: to specify different architectural styles and to check and execute evolutions according to them: each architecture must always respect, even if it evolves, its architectural style. In addition, if the architectural style evolves, its architectures must evolve accordingly. We have already applied this work to the well-known *pipe and filter* architectural style, and will explore more styles in the future.

Acknowledgements: this research was partly funded by the *Actions de Recherche Concertées – Ministère de la Communauté française – Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique*.

References

1. M. Jazayeri, On architectural stability and evolution, In Reliable Software Technlogies-Ada-Europe 2002, 2002, pages 13–23,Springer Verlag
2. D. Tamzalit, N. Sadou, M. Oussalah, Evolution problem within Component-Based Software Architecture,SEKE 2006, Kang Zhang and George Spanoudakis and Giuseppe Visaggio, pages 296-301, isbn 1-891706-18-7,
3. D. C. Luckham, J. J. Kenney, Larry M. Augustin, J. Vera, D. Bryan, W. Mann, specification and analysis of system architecture using Rapide, IEEE Transactions on Software Engineering, 1995, volume 21, pages 336–355,
4. David Garlan and Robert Monroe and David Wile, ACME: An Architecture Description Interchange Language, in Proceedings of CASCON1997, 1997, pages 169–183,

5. D. Garlan, R. T. Monroe, D. Wile, ACME: Architectural Description of Component-Based Systems, Foundations of Component-Based Systems, 2000, Gary T. Leavens and Murali Sitaraman Ed., Cambridge University Press, pages 47-68,
6. R. Allen, D. Garlan,The Wright architectural specification language, Technical report, CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science, 1996.
7. N. Medvidovic, D. S. Rosenblum, R. N. Taylor, A language and environment for architecture-based software development and evolution, ICSE'99, ISBN 1-58113-074-0, pages 44-53, Los Angeles, IEEE Computer Society Press, Los Alamitos, CA, USA,
8. J. Magee, N. Dulay, S. Eisenbach, J. Kramer, Specifying distributed software architectures, 5th European Software Engineering Conference, Sitges, Spain, 1995, pages 137-153, Springer-Verlag,
9. O. Barais, A.F. Le Meur, L. Duchien, J. Lawall, Software architecture evolution, pages 233-262, in [10]
10. T. Mens, S. Demeyer (editors), "Software Evolution, " Springer-Verlag, 2008
11. OMG, UML 2.1.2 infrastructure specification, Technical report, formal/2007-11-02, OMG, 2007,
12. G. Taentzer, AGG: A Graph Transformation Environment for Modeling and Validation of Software, Proc. AGTIVE 2003, Lecture Notes in Computer Science, Vol.3062, pages 446-453, 2004, Springer-Verlag
13. N.A. Lynch, M.R. Tuttle, An introduction to input, output automata. Technical Report MIT/LCS/TM-373, MIT Laboratory for Computer Science, November 1988.
14. M. Shaw and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, ISBN:0-13-182957-2, Prentice Hall, 1996.
15. T. Mens, G. Taentzer, O. Runge, Analyzing Refactoring Dependencies Using Graph Transformation Software and Systems Modeling Journal, Springer, vol. 6, num. 3, september 2007.

Metrics for Analyzing the Quality of Model Transformations — Extended Abstract*

Marcel van Amstel[1], Mark van den Brand[1], and Christian Lange[2]

[1] Department of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{M.F.v.Amstel|M.G.J.v.d.Brand}@tue.nl
[2] Federal Office for Information Technology
Barbarastraße 1, 50735 Cologne, Germany
mail@christian-lange.com

Model Driven Engineering [2] is an emerging software engineering discipline in which models play a central role throughout the entire development process. MDE combines domain-specific modeling languages for modeling software systems and model transformations for synthesizing them. Similar to other software engineering artifacts, model transformations have to be used by several developers, have to be changed according to changing requirements and should preferably be reused. Because of the prominent role of model transformations in today's and future software engineering, there is the need to define and assess their quality. Quality attributes such as modifiability, understandability and reusability need to be understood and defined in the context of MDE, i.e., for model transformations. The goal of our research is to make the quality of model transformations measurable. Currently, we focus on model transformations created using the ASF+SDF [3] term rewriting system, but we expect that our techniques can be applied to model transformations created using different transformation engines such as ATL [4] as well.

We identified seven quality attributes relevant for model transformations, viz. understandability, modifiability, reusability, modularity, conciseness, consistency, and completeness. Most of these quality attributes have already be defined for other software artifacts. We describe why they are specifically relevant for model transformations. We also identified a set of approximately forty metrics and related these to the quality attributes to define how the quality attributes should be assessed. Furthermore, we created a tool that can extract (most of) the metrics we defined from model transformations specified in the ASF+SDF formalism. We used this tool to evaluate a number of transformations. The same transformations were also manually evaluated by ASF+SDF experts to validate the relationship between metrics and quality attributes we established.

The next step in our ongoing research is to define a set of metrics for ATL and use these to assess the same quality attributes. We also plan to define a quality model in which we define a relationship between the quality attributes. Once we have identified quality problems in model transformations, we can propose a methodology for improving their quality.

* This is an extended abstract of [1]

References

1. van Amstel, M.F., Lange, C.F.J., van den Brand, M.G.J.: Metrics for analyzing the quality of model transformations. In Falcone, G., Guéhéneuc, Y., Lange, C., Porkoláb, Z., Sahraoui, H., eds.: Proceedings of the 12th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Paphos, Cyprus (July 2008) 41–51
2. Schmidt, D.C.: Model-driven engineering. Computer **39**(2) (2006) 25–31
3. van Deursen, A.: An overview of ASF+SDF. In van Deursen, A., Heering, J., Klint, P., eds.: Language Prototyping: An Algebraic Specification Approach. Volume 5. World Scientific Publishing (1996) 1–29
4. Jouault, F., Kurtev, I.: Transforming models with ATL. In Bruel, J.M., ed.: Satellite Events at the MoDELS 2005 Conference. Number 3844 in LNCS, Montego Bay, Jamaica, Springer (October 2005) 128–138

Dynamic Analysis of SQL Statements for Reverse Engineering Data-Intensive Applications

Anthony Cleve and Jean-Luc Hainaut

Laboratory of Database Applications Engineering
University of Namur, Belgium
21 rue Grandgagnage 5000 Namur
{acl,jlh}@info.fundp.ac.be

SQL statements control the bi-directional data flow between application programs and a database through a high-level, declarative and semantically rich data manipulation language. Analyzing these statements brings invaluable information that can be used in such applications as program understanding, database reverse engineering, intrusion detection, program behaviour analysis, program refactoring, traffic monitoring, performance analysis and tuning, to mention some of them. SQL APIs come in two variants, namely static and dynamic. While static SQL statements are fairly easy to process, dynamic SQL statements most often require dynamic analysis techniques that may prove more difficult to implement.

In this talk, we will address the problem of dynamic SQL query analysis in the context of software and database reverse engineering. We will explore the use of dynamic analysis techniques for extracting implicit information about both the program behavior and the database structure.

In the first part of the talk, we will describe and compare several possible techniques to capture the SQL statements that are executed in a data-intensive application program. Those techniques include, among others, code instrumentation, static analysis, aspect-based tracing, API overloading, API substitution, DBMS logs and tracing triggers.

The second part of the talk will elaborate on the analysis of SQL execution traces and its various applications in the context of reverse engineering. We will show, in particular, how SQL traces may help in discovering *implicit* schema constructs like undocumented foreign keys. We will then report on a recent case study in which SQL trace analysis was used for reverse engineering a data-intensive web application.

Static Estimation of Test Coverage

Tiago L. Alves[1,2], and Joost Visser[2]

[1] Universidade do Minho, Portugal
[2] Software Improvement Group, The Netherlands
{t.alves,j.visser}@sig.nl

Test coverage is an important indicator for unit test quality. Tools such as Clover can compute it by first instrumenting the code with logging functionality, and then log which parts are executed during unit test runs. Since computation of test coverage is a dynamic analysis, it presupposes a working installation of the software.

In the context of software quality assessment by an independent third party, a working installation is often not available. The evaluator may not have access to the required software libraries or hardware platform. The installation procedure may not be automated or documented.

We investigate the possibility to estimate test coverage at system, package and class levels through static analysis only. We present a method using slicing of static call graphs to estimate the actual dynamic test coverage and we identify the sources of imprecision. The method estimates test coverage at method level, by computing all methods reachable from tests. Coverage at method level is then used as basis for estimating class and package coverage.

We validate the results of static estimation by comparison to actual values obtained through dynamic analysis using Clover. The comparison is done using 12 systems both proprietary and open-source, with sizes ranging from small to large. Additionally, we apply our technique to 52 releases of a proprietary system.

We report that static estimation of test coverage at system level is highly correlated with dynamic coverage, demonstrating that static estimation can be a good predictor for the actual coverage.

A Formal Semantics for Multi-level Staged Configuration*

Andreas Classen**, Arnaud Hubaux, and Patrick Heymans

PReCISE Research Centre,
Faculty of Computer Science,
University of Namur
5000 Namur, Belgium
{acs,ahu,phe}@info.fundp.ac.be

Feature diagrams are a common means to represent, and reason about, variability in Software Product Line Engineering [2]. In this context, they have proved to be useful for a variety of tasks such as project scoping, requirements engineering and product configuration [2]. The core purpose of a feature diagram is to define concisely the set of legal *configurations* – generally called *products* – of some (usually software) artefact.

Given a feature diagram, the *configuration process* is the process of gradually making the choices defined in the feature diagram with the purpose of determining the product that is going to be built. In a realistic development, the configuration process is a small project itself, involving many stakeholders and taking up to several months [3]. Based on this observation, Czarnecki *et al.* [4] proposed the concept of *multi-level staged configuration* (MLSC), which splits the configuration process up into different levels that can be assigned to different stakeholders. This makes configuration more scalable to realistic environments.

However, MLSC never received a formal semantics, the primary indicator for precision and unambiguity, and an important prerequisite for reliable tool-support. We intend to fill this gap with a denotational semantics for MLSC that builds on our earlier work on formal semantics for feature diagrams [5] and extends it with the concepts of *stage*, *configuration path* and *level*.

The contribution is a precise and formal account of MLSC that makes the original definition [4] more explicit and reveals some of its subtleties. The semantics makes it possible to define the model-checking problem for MLSC and allowed us to discover some important properties that a feature model and its staged configuration process must possess. Our contribution is primarily of a fundamental nature, clarifying central concepts and properties related to MLSC. Thereby, we intend to pave the way for safer, more efficient and more comprehensive automation of configuration tasks.

---

* Full paper submitted to the *Third International Workshop on Variability Modelling of Software-intensive Systems* (VaMoS'09), Sevilla, Spain. An extended version is available as a technical report [1].

** FNRS Research Fellow.

Acknowledgements

This work is sponsored by the Interuniversity Attraction Poles Programme of the Belgian State of Belgian Science Policy under the MoVES project and the FNRS.

References

1. Classen, A., Hubaux, A., Heymans, P.: A formal semantics for multi-level staged configuration. Technical Report P-CS-TR SPLBT-00000002, PReCISE Research Center, University of Namur, Namur, Belgium (November 2008) Download at www.fundp.ac.be/pdf/publications/66426.pdf.
2. Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (July 2005)
3. Rabiser, R., Grunbacher, P., Dhungana, D.: Supporting product derivation by adapting and augmenting variability models. In: SPLC '07: Proceedings of the 11th International Software Product Line Conference, Washington, DC, USA, IEEE Computer Society (2007) 141–150
4. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged configuration through specialization and multi-level configuration of feature models. Software Process: Improvement and Practice **10**(2) (2005) 143–169
5. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Feature Diagrams: A Survey and A Formal Semantics. In: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06), Minneapolis, Minnesota, USA (September 2006) 139–148

On the classification of first-class changes

Peter Ebraert* and Theo D'Hondt

Computer Science Department
Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussel, Belgium
{pebraert,tjdhondt}@vub.ac.be

Abstract. Feature-oriented programming (FOP) is the research domain that targets the encapsulation of software building blocks as features, which better match the specification of requirements. Recently, we proposed change-oriented programming, in which features are seen as sets of changes that can be applied to a base program, as an approach to FOP.
In order to express features as sets of changes, those changes need to be classified in different sets that each represent a separate feature. Several classification strategies are conceivable. In this paper we identify three kinds of classification strategies that can be used to group the change objects. We compare them with respect to a number of criteria that emerged from our practical experience.

1 Introduction

Feature-oriented programming (FOP) is the study of feature modularity, where features are raised to first-class entities [1]. In FOP, features are basic building blocks, which satisfy intuitive user-formulated requirements on the software system. A software product is built by composing features. Recently, we proposed a bottom-up approach to FOP which consists of three phases [2, 3]. First, the change operations have to be captured into first-class entities. Second, those entities have to be classified in features (= separate change sets that each implement one functionality). Finally, those feature modules can be recomposed in order to form software variations that provide different functionalities.

In previous work, we already elaborated on two techniques to capture change objects. A classic way is to take two finished versions of a software system and to execute a Unix `diff` command on their respective abstract syntax trees [4], revealing the changes. This approach, however, only works *a posteriori*, and at a high level of granularity (the version level). A more subtle alternative is to log the developer's actions as he is performing the changes. The latter approach is based on change-oriented programming (ChOP) and was proven to provide a more complete overview of the history of development actions [5].

In this paper, we focus on the classification of changes into features. Classification has two aspects: the classification model and the classification technique, which is embodied by the different software classification strategies.

* Research funded by the Varibru research project initiated in the framework of the Brussels Impulse Programme for ICT supported by the Brussels Capital Region

2 Classification model

The classification model is a metamodel that consists of two parts: the change model and the actual classification model. Each part focuses on another level of granularity. The change model describes how the changes are modeled. Figure 1 shows that the change model separates between four kinds of changes, which can be composed. Atomic changes have a `subject`: the program building block affected by the change and defined by the Famix metamodel [6].

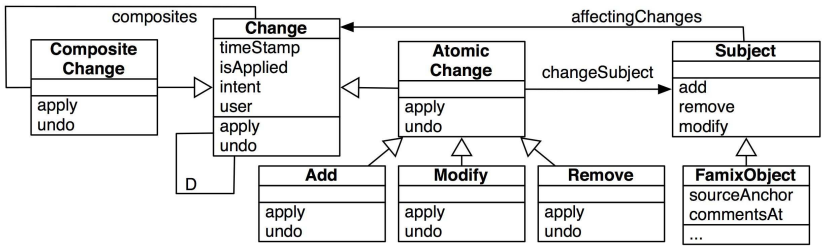

Fig. 1. Change Model

The actual classification model defines and describes the entities of the superstructure which is a flexible organisational structure based on feature and change objects. Figure 2 shows that the model contains three relations: D (the structural dependencies between the change objects), $CF4$ (which changes are grouped together into which feature) and Sub (which features are contained within another feature). The $cardinality$ of $CF4$ and Sub specifies whether or not the sons (changes and/or features) have to be included in a composition that includes the parent (a feature). This information can afterwards be used to validate feature compositions (as in Feature Diagrams [7]).

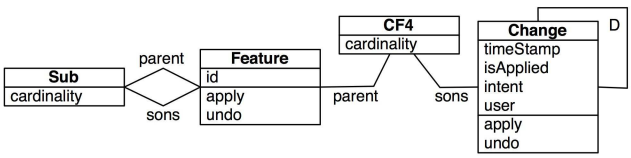

Fig. 2. Classification Model

3 Classification techniques

A classification strategy is a method for setting up classifications. Many classification strategies can be devised ranging from setting up classifications manually to generating

classifications automatically. We present three classification strategies: *manual* classification, *semi-autmatic* classification through clustering and *automatic* classification through forward tagging.

3.1 Manual classification

Manual classification is the simplest classification strategy: manually putting change objects in features. The strategy can be used by the software engineer to group changes according to his wishes. Since our classification model states that a change can only be classified in one feature, this strategy should be supported by a tool which enforces that rule.

The advantages of this strategy are twofold. First, it is a very straightforward technique which can easily be implemented. Second, it can be applied on change objects that were obtained both with a *diff* and *logging* strategy. The main disadvantage is the tediousness that comes with the manual effort of this strategy.

3.2 Semi-automatic classification

Change objects contain information about *by whom*, *when*, *why* and *where* the operations they reify were carried out. Using clustering techniques [8] based on metrics on these properties, change objects can be grouped. This classification is basically a manual classification strategy. Based on the the clusters of changes, the developer decides on how the changes must be classified.

The main advantage of this strategy is that it can be used to assist the developer doing a manual strategy. The disadvantages of this strategy are threefold. First, it is more difficult to implement (clustering should be supported). Second, different parameters in the metrics might give different clustering results. Extra research is required to find adequate parameters. Third, this success of this strategy largely depends on the amount of information available in the change objects and is consequently not recommended to be used in combination with a *diff* strategy.

3.3 Automatic classification

In many cases, a manual classification strategy is not a feasible option. For large software systems it would take a long time to classify all classes by hand. Often classification of a software system is an activity that cannot be done by one software engineer alone since one software engineer seldom knows the whole system. When manual classification is not a valid option for the classification problem at hand automatic classification may provide a solution.

The idea behind automatic classification is that when software engineers carry out a development operation, for example implementing a new or changed specification or fixing a bug, they usually know the context in which changes are made. Moreover, the IDE knows the exact time and in what part of the software, the operation is performed. In stead of keeping this knowledge implicit in the heads of the developers, it is *tagged* into the changes partially by the developer and partially by the IDE. Afterwards, these tags can be processed automatically to generate tag-based classifications. Since

software engineers are usually lazy when source code documentation is concerned, relying on discipline is not realistic. It is up to the IDE to make sure that classification knowledge about the software is recorded.

Advantages of this approach are that it can be used on the biggest of systems as it does not require manual labour, and that it is relatively easy to implement. The sole inconvenience is that it can only be used in combination with a *logging* strategy, which enforces developers to do forward tagging.

4 Conclusion

We introduced a model and three strategies to classify changes and/or features in sets that represent features. The model consists of two parts which respectively model the change objects and the actual classification. The first strategy is straightforward: *manual* classification is a strategy to put together classifications manually. *Semi-automatic* classification is based on clustering changes together based on properties such as *by whom*, *when*, *why* and *where* the changes were applied. *Automatic* classification is based on forward tagging, and automatically groups changes together. Our findings are summarised as follows:

	Manual	Semi-Auto	Automatic
Capturing Changes	diff, logging	logging	logging
Amount of manual labour	high	average	low
Error probability	high	high	low

Only the automatic strategy is usable in a context of large-scale software systems. As that strategy requires logging as a technique to capture changes, we conclude that the development environment should support logging and enforce forward tagging, so that the changes can automatically be classified in recomposable feature modules.

References

1. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. In: Proceedings of the 25th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2003) 187–197
2. Ebraert, P., Van Paesschen, E., D'Hondt, T.: Change-oriented round-trip engineering. Technical report, Vrije Universiteit Brussel (2007)
3. Ebraert, P., Vallejos, J., Costanza, P., Van Paesschen, E., D'Hondt, T.: Change-oriented software engineering. In: ICDL '07: Proceedings of the 2007 international conference on Dynamic languages, New York, NY, USA, ACM (2007) 3–24
4. Xing, Z., Stroulia, E.: Umldiff: An algorithm for object-oriented design differencing. In: Proceedings of the 20th International Conference on Automated Software Engineering. (2005)
5. Robbes, R., Lanza, M.: A change-based approach to software evolution. Electronic Notes in Theoretical Computer Science (2007) 93–109
6. Demeyer, S., Tichelaar, S., Steyaert, P.: FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Berne (1999)

7. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
8. Romesburg, C.: Cluster Analysis for Researchers. Number 978-1-4116-0617-3. Krieger (1990)

Towards an automated tool for correcting design decay

Sergio Castro

Département d'Ingénierie Informatique
Université catholique de Louvain
Place Sainte Barbe 2,
B-1348 Louvain-la-Neuve, Belgium
sergio.castro@uclouvain.be

Under the influence of continuous evolution and maintenance tasks, the original design structure of a software system often rapidly erodes away. This observation has already led to a good number of techniques and tools for expressing and checking different kinds of design constraints over the source-code of a system. However, in addition to tools and techniques that support the automatic detection of violated design constraints, tools are needed to support the developers in reconciling the broken implementation structure with these formulated design constraints.

We propose a technique for the detection and semi-automatic correction of user-defined design constraints over the source code. To detect adherence of design constraints in the source code, our technique relies on the use of logic rules that reason over the implementation. The corrective actions that are needed to fix detected broken design constraints are deduced using abductive logic reasoning techniques. The resulting technique is a uniform framework for the definition of design constraints as well as their corresponding corrective actions. An early prototype that allows us to experiment with this technique is currently being developed as an extension to the IntensiVE tool suite, thereby adding support for correcting the implementation structure, in addition to the already existing definition and enforcement of structural design constraints over source code.

Enabling Refactoring with HTN Planning to Improve the Design Smells Correction Activity

Javier Pérez

University of Valladolid; Department of Computer Science
jperez@infor.uva.es

Abstract. Refactorings are a key technique to software evolution. They can be used to improve the structure and quality of a software system. This paper introduces a proposal for generating refactoring plans with hierarchical task network planning, to improve the automation of the bad smells correction activity.

1 Introduction

Over the evolution of a software system, its structure deteriorates because the maintenance efforts concentrate more on the correction of bugs and on the addition of new functionalities than on the control and improvement of the software's architecture and design [1]. Bad design practices, often due to inexperience, insufficient knowledge or time pressure, are at the origin of design smells. They can arise at different levels of granularity, ranging from high level design problems, such as antipatterns [2], to low-level or local problems, such as code smells [3].

Design smells are problems encountered in the software's structure, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to true compilation errors and run-time errors in the future. Design Smell management refers to the set of techniques, tools and approaches addressed to detect and to correct or, at least, reduce design smells to improve software quality. Among the activities involved in design smell management, correction and detection are the most significant ones.

2 Detection and Correction of Smells

The detection techniques proposed in the literature mainly consist on defining and applying rules for identifying design smells. Meanwhile, correction techniques often consist on suggesting which transformations could be applied to the source code of the system in order to restructure it, by correcting or, at least, reducing its design problems. There has been an increasing number of works dealing with smell detection, and the most successful ones are those based on metrics [4], and on the jointed use of metrics and structural patterns analysis [5].

The correction activity has not been explored as much as the detection one. Most approaches focus on suggesting which are the best redesign changes to perform, and which are the best structures to remedy the smell and reflect the original design intent.

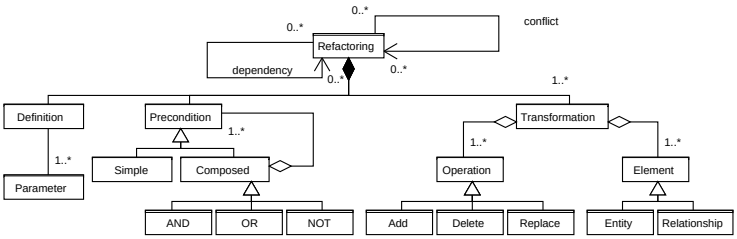

Fig. 1. A model for refactoring operations.

The preferred technique for correction is refactoring [3], because the objective is not to remove bugs or errors. In terms of observable behaviour, we aim at leaving the system untouched. The automation of the correction strategies, based on refactorings, faces the problem of precondition fulfillment, as mentioned in [6].

It is rare that the preconditions of the desired refactorings could be fulfilled by the system's source code at its current state. For example, to allow moving a method from one class to another, one should probably, first, have to move the attributes accessed from it. In these cases, the developer has to plan ahead how to solve this problem. This can be done either by choosing a different refactoring path or by applying other preparatory refactorings to enable the precondition which previously failed. Moreover, violation of preconditions is the most common error developers encounter when trying to apply a refactoring operation [7]. Therefore, suggesting refactorings is not enough to allow for automated correction of design smells.

3 Anatomy of a Refactoring Operation

A refactoring can be seen as a conditional transformation [8], which is composed of a precondition and a set of transformations. The precondition establishes the situations under which the refactoring can be executed, while the transformation part specifies the changes that are to be applied to the source code. If the precondition of a refactoring is fulfilled when it is performed, the system's behaviour is preserved. Figure 1 shows a simplified model for refactoring operations.

Once a smell has been detected and once the refactorings to correct it have been given out, they can't be immediately applied if their preconditions fail. Therefore, refactoring suggestions don't suffice to automate the activity of bad smell correction, we need refactoring plans.

We define a refactoring plan as the specification of a refactoring sequence that matches a system redesign proposal and can actually be executed over the current system's source code. To improve the automation of the smell correction activity, we intend to support the automated generation of refactoring plans.

A variety of techniques can be used to reason about refactorings and assist the generation of refactoring plans. Analysis of dependencies and conflicts can be performed to find out which refactorings can enable or disable other refactoring's preconditions [9].

First-order logic inference can help composing refactoring sequences [8]. Automated planning [10], can integrate all these techniques.

4 Enabling Refactoring with HTN planning

Automated planning [10] is an artificial intelligence technique to generate sequences of actions that will achieve a certain goal when they are performed. We think that automated planning is a technique suitable to be used in the generation of refactoring plans.

For a typical automated planner, the current state of the world is represented as a set of logical terms which are changed through application of operators. Operators are composed of a precondition which specifies the conditions under which they can be applied, and two separate sets of actions which specify how the operator modifies the state of the world. These lists enumerate the terms the operator will add to and delete from the current state. A goal is a list of terms which represents a certain state of the world we want to achieve. A planner computes a plan as a sequence of operator instances that changes the world to achieve the desired goals in the final state.

Among all the existing planning approaches, we think that hierarchical task network (HTN) planning provides the best balance between search-based and procedural-based strategies, for the problem of refactoring planning. We have explored other approaches, such as partial-order backwards planning, only to discover that combinatorial explosion and lack of expressivity disallow their application in the refactoring planning domain.

HTN planning [10], introduces the concept of “task”, which models actions composed by simple operators or by other tasks. Task networks allow to include domain knowledge describing which subtasks should be performed to accomplish another one. HTN planning and forward search allows very expressive domain definitions which lead to very detailed domains with a lot of domain knowledge which can guide the planning process in a very efficient way.

Starting from a set of refactorings, the refactoring dependencies, and the system’s source code and a redesign proposal, an HTN planner can obtain a refactoring plan matching the redesign proposal, while solving the problem of failing preconditions.

To search for refactoring plans we use the representation from [8], which turns the system’s AST into a set of logical terms. This set builds up the planner’s state of the world. Refactorings are modeled with tasks and operators. Tasks hierarchies allow to specify the algorithm of a refactoring along with the dependencies with other refactorings. Thus, using tasks and subtasks dependencies, we model which refactorings should be executed in order to enable the precondition of another one. An HTN planner can be tailored to search for plans which achieve a certain design structure, or which enable application of a desired set of refactorings.

5 Conclusions

This paper introduces a proposal to enable refactoring application through automated planning, more precisely HTN planning. Automation support for refactoring planning can improve any practice which uses them and many software evolution techniques,

particularly the correction of design smells. We are currently preparing experiments to show the feasibility of this approach.

Acknowledgements

I want to thank Tom Mens, Naouel Moha and Carlos López who have helped me reviewing the state of the art in design smells management.

This work has been partially funded by the regional government of Castilla y León (project VA-018A07).

References

1. Frederick P. Brooks, J.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Publishing Company, Reading, MA , USA (1975)
2. Brown, W.H., Malveau, R.C., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley (March 1998)
3. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Object Technology Series. Addison-Wesley (1999)
4. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer (2006)
5. Moha, N.: DECOR : Détection et correction des défauts dans les systèmes orientés objet. PhD thesis, Université des Sciences et Technologies de Lille; Université de Montréal (August 2008)
6. Trifu, A., Reupke, U.: Towards automated restructuring of object oriented systems. Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on (March 2007) 39–48
7. Murphy-Hill, E., Black, A.P.: Breaking the barriers to successful refactoring: observations and tools for extract method. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, New York, NY, USA, ACM (2008) 421–430
8. Kniesel, G.: A logic foundation for conditional program transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn (January 2006)
9. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Software and Systems Modeling **6**(3) (September 2007) 269–285
10. Ghallab, M., Nau, D., Traverso, P.: Automated Planning; Theory and Practice. Morgan Kaufmann (2004)

Locating Features in COBOL Mainframe System: Preliminary Results of a Financial Case Study

Joris Van Geet

University of Antwerp
Joris.VanGeet@ua.ac.be

Abstract. As Lehman's laws indicate, one of the main causes of software evolution is changing business requirements. With the advent of Service Oriented Architectures, more effort is being spent on reorganizing software systems to better match the company and business structures, thereby making the systems more adaptive to these changing requirements. As a first step in this process we need to locate the relevant business functionalities (features) in the system. In an ongoing case study we apply formal concept analysis on execution profiles generated by carefully chosen execution scenarios, as proposed by Eisenbarth et. al.[2003], on a COBOL mainframe system of a Belgian bank.

1 Prerequisites

- Investigate or set up tracing facilities on mainframe sufficiently powerful to provide an execution profile.
- Select features (i.e., functionality provided by the system) that need to be located within the system.

2 Analysis Steps

- Create executable scenarios that trigger the selected features and create a scenario-feature mapping as this is usually no one-to-one mapping. This information should be gathered in close cooperation with domain experts.
- Execute the scenarios while extracting an execution profile for each scenario. This results in a list of computational units (at whatever granularity necessary and available) that were executed during the scenario.
- Construct a concept lattice correlating the computational units with the executed scenarios.
- Combine the concept lattice and the scenario-feature mapping to correlate the computational units with the features.
- Iterate over these steps by either refining/adding/removing scenarios or refining/filtering dynamic analysis results.

References

[2003] T. Eisenbarth and R. Koschke and D. Simon Locating features in source code IEEE Transactions on Software Engineering, 29 210-224 (2003)

Verifying the design of a Cobol system using Cognac

Andy Kellens, Kris De Schutter, and Theo D´Hondt

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussels
{akellens | kdeschut | tjdhondt}@vub.ac.be

1 Introduction

A property of large-scale, industrial systems is that they are intended to be used and maintained over a long period of time. In order to keep such large systems maintainable, it is important that developers respect the various rules that underlie the design of such systems during the subsequent evolutions of the system. These design rules can range from low-level naming conventions and coding guidelines, over the correct use of frameworks to the different constraints that are imposed by the architecture of the system.

Serving as a testimony to this problem is the amount of effort that has been devoted — both in academia and industry — to tools and approaches that aid in verifying design rules with respect to a system's source code. Examples of such tools are low-level code checkers such as Lint [3] and CheckStyle [1], tools such as Ptidej [2] that enforce design patterns, approaches like Reflexion Models [5] that verify a high-level specification (architecture) of a system with the source code and so on.

However, it seems that the vast majority of these tools neglects the Cobol language, which is still one of the most prevalent languages in industry. In this presentation we discuss *Cognac*, our approach that offers a general framework for documenting design rules in Cobol code and verifying their validity with respect to the implementation.

2 Context

The context of our work is a fairly large case study (500KLoc) we are conducting together with the Flemish company inno.com that has recently designed a new Cobol system for a Belgian bank. Their interest in verifying this design with respect to the implementation is three-fold:

- The implementation of the system has been out-sourced, resulting in that our industrial partner is interested in knowing whether the external partner respected the intended design and the provided coding guidelines/naming conventions;
- The system is expected to be in use for 20 to 25 years, resulting in that keeping the system maintainable is a valuable asset;

– The system is being implemented in various phases spread over multiple years, during which novel functionality is added. Our industrial partner wants to assure that during these phases the design is respected and wants to assess possible violations of the design.

3 Outline of our approach

Our tool — Cognac — offers developers a common framework to document and verify design rules in Cobol systems. Cognac is developed as an extension to our IntensiVE tool suite [4]. In a nutshell, the main idea of IntensiVE is to document design rules by grouping source-code entities in so-called *intensional views*: sets of source-code entities that belong conceptually together and that are defined by means of a logic program query (expressed in the SOUL language [6]). Either by specifying multiple, alternative definitions for one intensional view, or by imposing constraints over intensional views, design rules can be expressed using the tool. IntensiVE offers a number of subtools that allow for the verification of these design rules with respect to the source code and offer developers detailed feedback concerning possible violations.

Reasoning about Cobol posed a number of interesting challenges. Therefore, Cognac makes the following extensions to IntensiVE:

– There exist different variants of the Cobol language, each specifying a large amount of different language constructs. In order to deal with this problem, we have implemented a customisable island-based parser. Such an island-based parser allows us to extract only the information that is necessary for the analyses we wish to express from Cobol source code;
– We have implemented a set of SOUL predicates that reason about the Cobol parse tree, such that we can define intensional views (and constraints over these views) over such programs. This set of predicates consists of basic predicates that allow to query the structure of Cobol programs, predicates that retrieve relations between the various source-code entities, as well as predicates that e.g. extract information from embedded SQL statements;
– By reasoning purely over parse trees, we were restricted in the number of interesting design rules that can be expressed. Therefore, we complimented the set of SOUL predicates with two static analyses. One analysis is used to resolve *call* statements in the source code and link these statements to the actual Cobol programs that might get invoked. The second analysis — data field aliasing — conservatively computes aliases between different data fields in Cobol programs.

4 Design rules in the case study

In this section, we take a brief look at three of the design rules that we have documented in the case study. During the presentation, a more in-depth look at these design rules will be given, along with details about how we documented them using Cognac.

Section layering In the case study under investigation, the designers of the system introduced a clear layered structure in the individual Cobol programs as a means to make the control flow more explicit. More specifically, the various sections in each program were divided into separate layers in which sections in one layer are only allowed to invoke sections in the same, or a lower layer. This design rule is reflected in the source code by means of a simple naming convention: each section's name is prefixed with a letter grouping sections at the same level using the same letter. From within each section, only sections may be invoked with the same starting letter, or with a letter that comes later in the alphabet. During the presentation, we show how to document this design rule by creating an intensional view that groups all callers and callees of sections, and by imposing a constraint over the elements of this intensional view specifying that for all pairs of callers and callees, the first letter of the callee should be the same or come later in the alphabet than the first letter of the caller.

Copybook - linkage correspondence A Cobol program that can be called from within another program needs to declare a linkage section that specifies the data definition of the arguments that it expects as input. In our case study, one design rule that needs to be obeyed is that, if a program calls another program, it uses the same data definition for the argument of both caller as well as callee. In order to ease this correspondence, a copybook is used that contains the data definition and that should be included in the linkage section of the called program as well as in the calling program. Since this pattern however is not enforced by the language itself, we have documented it using intensional views.

Database modularity The case study we investigated is designed in a component-oriented fashion. In the system, the various components consist of a top-level program that serves as the component's interface, along with a number of programs to which this top-level program delegates particular requests. Also associated with each component is a set of database tables that contain the persistent data which the module is responsible for. In order not to break this modularity, only programs from within one particular module are allowed writing access to the tables associated with that module. All other programs need to retrieve and manipulate data via the interface program of that module. Preferably also, the number of programs within a module that are allowed to write to the associated tables is limited. As we will show in the presentation, in order to verify this design rule, we opted to use a more pragmatic approach in which we use a visualisation as a means to provide the original designers of the system with feedback concerning the use of database tables in the current implementation.

Acknowledgements

Andy Kellens is funded by a research mandate provided by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen). Kris De Schutter received support from the Belgian research project AspectLab, sponsored by the IWT Vlaanderen.

References

1. Checkstyle, December 2006. `http://checkstyle.sourceforge.net`.
2. Y. Guéhéneuc. Three musketeers to the rescue – meta-modeling, logic programming, and explanation-based constraint programming for pattern description and detection. In *Workshop on Declarative Meta-Programming at ASE 2002*, 2002.
3. S.C. Johnson. Lint, a c program checker. In M.D. McIIroy and B.W. Kemighan, editors, *Unix Programmer's Manual*, volume 2A. AT&T Bell Laboratories, seventh edition, 1979.
4. K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views: A case study. *Elsevier Journal on Computer Languages, Systems & Structures*, 32(2-3):140–156, 2006.
5. G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Symposium on the Foundations of Software Engineering (SIGSOFT)*, pages 18–28, 1995.
6. R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, January 2001.

Author Index