Ákos Kiss (Ed.)

13th Symposium on Programming Languages and Software Tools

SPLST'13

Szeged, Hungary, August 26–27, 2013

Proceedings

University of Szeged

13th Symposium on Programming Languages and Software Tools
SPLST'13
Szeged, Hungary, August 26–27, 2013
Proceedings

Edited by Ákos Kiss

University of Szeged
Faculty of Science and Informatics
Institute of Informatics
Árpád tér 2., H-6720 Szeged, Hungary

ISBN 978-963-306-228-9 (printed)
ISBN 978-963-482-716-0 (PDF)

Preface

On behalf of the steering and program committees, welcome to the 13th Symposium on Programming Languages and Software Tools (SPLST'13). The series started in 1989 in Szeged, Hungary, and since then, by tradition, it has been organized every second year in Hungary, Finland, and Estonia, with participants coming from all over Europe. This year, the thirteenth edition of the symposium is back again in Szeged on August 26–27, 2013.

The purpose of the Symposium on Programming Languages and Software Tools is to provide a forum for software scientists to present and discuss recent researches and developments in computer science. The scope of the symposium covers ongoing research related to programming languages, software tools, and methods for software development.

This volume contains the 20 full papers that were accepted by the program committee based on an anonymous peer review process. We hope that the diversity of the papers will lead to stimulating discussions.

As the organizers of the symposium, we would like to thank all the authors and reviewers for bringing together an interesting program for this year's SPLST.

Ákos Kiss
General Chair

Organization

SPLST'13 was organized by the Department of Software Engineering, University of Szeged.

General Chair

Ákos Kiss (University of Szeged, Hungary)

Steering Committee

Zoltán Horváth (Eötvös Loránd University, Hungary)
Kai Koskimies (Tampere University of Technology, Finland)
Jaan Penjam (Institute of Cybernetics, Estonia)

Program Committee

Hassan Charaf (Budapest University of Technology and Economics, Hungary)
Tibor Gyimóthy (University of Szeged, Hungary)
Zoltán Horváth (Eötvös Loránd University, Hungary)
Pekka Kilpeläinen (University of Eastern Finland, Finland)
Ákos Kiss (University of Szeged, Hungary)
Kai Koskimies (Tampere University of Technology, Finland)
Tamás Kozsik (Eötvös Loránd University, Hungary)
Peeter Laud (Cybernetica, Institute of Information Security, Estonia)
Erkki Mäkinen (University of Tampere, Finland)
Jyrki Nummenmaa (University of Tampere, Finland)
Jukka Paakki (University of Helsinki, Finland)
András Pataricza (Budapest University of Technology and Economics, Hungary)
Jari Peltonen (Tampere University of Technology, Finland)
Jaan Penjam (Institute of Cybernetics, Estonia)
Attila Pethő (University of Debrecen, Hungary)
Margus Veanes (Microsoft Research, Redmond, USA)

Additional Referees

Zoltán Alexin, Márk Asztalos, Vilmos Bilicki, István Bozó, Dimitrij Csetverikov, Péter Ekler, Rudolf Ferenc, Zsolt Gazdag, Ferenc Havasi, Zoltán Herczeg, Judit Jász, Róbert Kitlei, Tamás Mészáros, Zoltán Micskei, Ákos Szőke, Zalán Szűgyi, Zoltán Újhelyi, András Vörös

Table of Contents

Monitoring Evolution of Code Complexity in Agile/Lean Software Development

A Case Study at Two Companies

Vard Antinyan[1], Miroslaw Staron[1], Wilhelm Meding[2], Per Österström[3], Henric Bergenwall[3], Johan Wranker[3], Jörgen Hansson[4] Anders Henriksson[4]

Computer Science and Engineering [2] Chalmers | [1] University of Gothenburg
[3] Ericsson AB, Sweden[4] AB Volvo, Sweden
SE 412 96 Gothenburg

Abstract. One of the distinguishing characteristics of Agile and Lean software development is that software products "grow" with new functionality with relatively small increments. Continuous customer demands of new features and the companies' abilities to deliver on those demands are the two driving forces behind this kind of software evolution. Despite the numerous benefits there are a number of risks associated with this kind of growth. One of the main risks is the fact that the complexity of the software product grows slowly, but over time reaches scales which makes the product hard to maintain or evolve. The goal of this paper is to present a measurement system for monitoring the growth of complexity and drawing attention when it becomes problematic. The measurement system was developed during a case study at Ericsson and Volvo Group Truck Technology. During the case study we explored the evolution of size, complexity, revisions and number of designers of two large software products from the telecom and automotive domains. The results show that two measures needed to be monitored to keep the complexity development under control - McCabe's complexity and number of revisions.

Keywords: complexity; metrics; risk; Lean and Agile software development; code; potentially problematic; correlation; measurement systems;

1 Introduction

Actively managing software complexity has become an important aspect of continuous software development in large software products. It is generally believed that software products developed in a continuous manner are getting more and more complex over time, and evidence shows that the rising complexity drives to decreasing quality of software [1-3]. The continuous increase of code base and incremental increase of complexity can lead to large, virtually unmaintainable source code if left unmanaged.

A number of methods have been suggested to measure various aspects of software complexity, e.g. [4-10], accompanied with a number of studies indicating how adequately the proposed methods can relate to software quality. One of the well-known complexity measures, McCabe's cyclomatic complexity has been shown to be a good quality indicator although it does not reveal all aspects of complexity [11-14].

Despite the considerable amount of research conducted about the influence of complexity on software quality, little results can be found on how complexity influences on a continuously developed software product and how to effectively monitor small yet continuous increments of complexity in growing products. Therefore a question remains how the previously established methods can be as efficiently used for software quality evaluation:

How to monitor complexity changes effectively when delivering feature increments to the main code branch in the product codebase?

The aim of this research is to develop methods and tool support for actively monitoring increments of complexity and drawing the attention of product managers, project leaders, quality responsible and the teams to the potentially problematic trends of growing complexity. In this paper we focus on the level of self-organized software development teams who often deliver code to the main branch for further testing, integration with hardware and ultimate deployment to end customers.

We address this question by conducting a case study at two companies which develop software according to Agile and Lean principles. The studied companies are Ericsson AB in Sweden which develops telecom products and Volvo Group Truck Technology which develops trucks under four brands – Volvo, Renault, Mack and UD Trucks.

Our results show that using a number of complementary measures of complexity and development velocity – McCabe's complexity and number of revisions per week – support teams in decision making, when delivering potentially problematic code to the main branch. By saying potentially problematic we mean that there is a tangible chance that the delivered code is fault prone or difficult to understand and maintain. Monitoring trends in these variables effectively draws attention of the self-organized Agile teams to a handful of functions and files which are potentially problematic. The handful of functions are manually assessed, and before the delivery the team formulates the decision whether they indeed might cause problems. The initial evaluation in two ongoing software development projects shows that using the two measures indeed draws attention to the most problematic functions.

2 Related Work

2.1 Continuous Software Evolution

A set of measures useful in the context of continuous deployment can be found in the work of Fritz [15] in the context of market driven software development organization. The metrics presented by Fritz measure such aspects as continuous integration pace or the pace of delivery of features to the customers. These metrics complement the two indicators presented in this paper with a different perspective important for product management.

The delivery strategy, which is an extension of the concept of continuous deployment, has been found as one of the three key aspects important for Agile software development organizations in a survey of 109 companies by Chow and Cao [16]. The indicator presented in this paper is a means of supporting organizations in their transition towards achieving efficient delivery processes.

Ericsson's realization of the Lean principles combined with Agile development was not the only one recognized in literature. Perera and Fernando [17] presented another approach. In their work they show the difference between the traditional and Lean-Agile way of working. Based on our observations, the measures and their trends at Ericsson were similar to those observed by Perera and Fernando.

2.2 Related Complexity Studies

Gill and Kemerer [8] propose another kind of cyclomatic complexity metric – cyclomatic complexity density and they show its usefulness as a software quality indicator. Zhang and Zhang [18] developed a method based on lines of code measure, cyclomatic complexity number and Halstead's volume to predict the defects of a software component. Two other studies provided evidence that files having large number of revisions are defect prone and hard to maintain [19], [20].

2.3 Measurement Systems

The concept of an early warning measurement system is not new in engineering. Measurement instruments are one of the cornerstones of engineering. In this paper we only consider computerized measurement systems – i.e. software products used as measurement systems. The reasons for this are: the flexibility of measurement systems, the fact that we work in the software field, and similarity of the problems – e.g. concept of measurement errors, automation, etc. An example of a similar measurement system is presented by Wisell [21] where the concept of using multiple measurement instruments to define a measurement system is also used. Although differing in domains of applications these measurement systems show that concepts which we adopt from the international standards (like [22]) are successfully used in other engineering disciplines. We use the existing methods from the ISO standard to develop the measurement systems for monitoring complexity evolution.

Lowler and Kitchenham [23] present a generic way of modeling measures and building more advanced measures from less complex ones. Their work is linked to the TychoMetric [24] tool. The tool is a very powerful measurement system framework, which has many advanced features not present in our framework (e.g. advanced ways of combining metrics). A similar approach to the TychoMetric's way of using metrics was presented by Garcia et al. [25]. Despite their complexity, both the TychoMetric tool and Garcia's approach can be seen as alternatives in the context of advanced data presentation or advanced statistical analysis over time.

Meyer [26, pp. 99-122] claims that the need for customized measurement systems for teams is one of the most important aspects in the adoption of metrics at the lowest levels in the organization. Meyer's claims were also supported by the requirements that the customization of measurement systems and development of new ones should be simple and efficient in order to avoid unnecessary costs in development projects. In our research we simplify the ways of developing Key Performance Indicators exemplified by a 12-step model of Parmenter [27] in the domain of software development projects.

3 Design of the Case Study

This case study was conducted using action research approach [28-30] where the researchers were part of the company's operations and worked directly with product development units of the companies. The role of Ericsson in the study was the development of the method and its initial evaluation, whereas the role of Volvo Group Truck Technology was to evaluate the method in a new context.

3.1 Ericsson

The organization and the project within Ericsson, which we worked closely with, developed large products for the mobile telephony network. The number of the developers in the projects was up to a few hundreds[1]. Projects were executed according to the principles of Agile software development and Lean production system, referred to as Streamline development (SD) within Ericsson [31]. In this environment, different development teams were responsible for larger parts of the development process compared to traditional processes: design teams (cross-functional teams responsible for complete analysis, design, implementation, and testing of particular features of the product), network verification and integration testing, etc.

The needs of the organization had evolved from metric calculations and presentations (ca. 7 years before the writing of this paper) to using predictions, simulations, early warning systems and handling of vast quantities of data to steer organizations at different levels and providing information from teams to management.

3.2 Volvo Group Truck Technology (GTT)

The organization which we worked with at Volvo Group developed Electronic Control Unit (ECU) software for trucks for such brands like Volvo, Renault, UD Trucks and Mack. The collaborating unit developed software for two ECUs and consisted of over 40 designers, business analysts and testers at different levels. The process was iterative, agile, involving cross functional teams.

The company used measures to control the progress of its projects, to monitor quality of the products and to collect data semi-automatically, i.e. automatically gathering of data from tools with the manual analysis of the data. The metrics collected at the studied unit fall into the categories of contract management, quality monitoring and control, predictions and project planning. The intention of the unit was to build a measurement system to provide stakeholders (like project leaders, product and line managers or the team) with the information about the current and predicted status of their products.

3.3 Process

According to the principles of action research we adjusted the process of our research with the operations of the company. We worked closely with project teams with dedicated designers, architects and managers being part of the research team. We conducted the study according to the following pre-defined process:

- Obtaining access to the source code of the products and their different releases
- Calculate complexity of all functions in the code
- Identify functions which changed complexity through 4 main releases
- Identify functions which changed complexity in 5 service releases between the two main releases
- Identify drivers for complexity changes in a subset of these functions
- Add new measures to the study:
  - Complexity per file
  - # revisions – to explore files which were changed often
  - # designers – to explore files which were changed by many designers in parallel

[1] The exact size of the unit cannot be provided due to confidentiality reasons.

  - # Number of lines of code (size) – to explore large files and functions
- Correlate measures to explore their dependencies
- Develop a measurement system (according to ISO 15939) to monitor the potentially problematic files.
- Monitor and evaluate the product during two releases

The above process was used during the development of the method at Ericsson and replicated at Volvo Group Truck Technology.

3.4 Units of Analysis

During our study we analyzed two different products – software for a telecom product at Ericsson and software for one electronic control unit from Volvo GTT from the automotive domain.

Ericsson: The product was a large telecommunication product composed by over one million lines of code with several tens of thousands C/C++ functions. Most of the source code was developed using C. The product had a few releases per year with a number of service releases in-between them. All versions of the source code of the product including the main and service releases were stored in version control system, IBM/Rational ClearCase. The product was a mature telecommunication product with a stable customer base. The product has been in development for a number of years.

The measures specified in the previous section were collected from different baseline revisions of the source code in ClearCase. In order to increase the internal validity of data collection and the quality of data we communicated closely with a reference group during bi-weekly meetings over a period of 8 months. The reference group consisted of 2 senior designers, one operational architect, one research engineer from the company, one manager and one metric team leader. The discussions considered the suitability of measures, measurement methods and functions (according to ISO/IEC 15939), validity of results and effectiveness of our measurement system.

Volvo GTT: The product was an embedded software system serving as one of the main computer nodes for a product line of trucks. It consisted of a few hundred thousand lines of code and several thousand C functions. The version control system is ClearCase. The software product had tight releases every 6-8 weeks. The analyses that were conducted were replications of the case study at Ericsson under the same conditions and using the same tools. The results were communicated with designers of the software product after the data was analyzed.

At both companies we developed measurement systems for monitoring the files and functions that can be risk driving when merging new code into the main branch. We defined the risk of merging a newly developed or a maintained function to main code base as a chance that the merged code would introduce new faults or would be noticeably more difficult to understand and maintain.

3.5 Measures in the Study

Table 1 presents the measures which we used in our study and their definitions:

Table 1. Metrics and their definitions

Name of measure	**Abbre-viation**	**Definition**
Number of non-commented lines of code	NCLOC	The lines of non-blank, non-comment source code in a function

McCabe's cyclomatic complexity of a function	M	The number of linearly independent paths in the control flow graph of a function, measured by calculating the number of 'if', 'while', 'for', 'switch', 'break', '&&', '||' tokens
McCabe's cyclomatic complexity of a file	File M	The sum of all functions' M in a file
McCabe's cyclomatic complexity delta of a function	ΔM	The increase or decrease of M of a function during a specified time interval. We register the file name, class name (if available) and function name in order to identify the same function and calculate its complexity change in different releases.
McCabe's cyclomatic complexity delta of a file	File ΔM	The increase or decrease of File M during a specified time interval
Number of revisions of a file	NR	The number of check-ins of files in a specified ClearCase branch and its all sub-branches in a specified time interval
Number of designers of a file	ND	The number of developers that do check-in of a file on a specified ClearCase branch and all of its sub-branches during a specified time interval
Complexity of the most complex function in a file	Max M f	The complexity number M of the most complex function in a file

3.6 Focus Group

During this study we had the opportunity to work with a reference group at Ericsson and a designer at Volvo GTT. The aim of the reference group was to support the research team with expertise in the product domain and to validate the intermediate findings as prescribed by the principles of Action research. The group interacted with researchers on a bi-weekly meeting basis for over 8 months. At Ericsson the reference group consisted of:

- One product manager with over 10 years of experience and over 5 years of experience with Agile/Lean software development
- One measurement program/team leader with over 10 years of experience with software development and over 5 years of experience with Agile/Lean at Ericsson
- Two designers with over 6 years of experience in telecom product development.
- One operational architect with over 6 years of experience
- One research engineer with over 20 years of experience in telecom product development

At Volvo GTT we worked with one designer who had the knowledge about the product and over 10 years of experience with software development at the company.

4 Results and analysis

4.1 Evolution of the Studied Measures Over Time

We measured M for 4 main and 5 service releases at Ericsson and for 4 releases for the product at Volvo GTT. The results showed there are many new complex functions introduced as part of service releases. We observed that a large number of functions change the argument list during development. Many functions had long list of arguments which meant that the designers need to add or remove an argument or change the argument name to resolve a specific task. Thus the majority of the functions that has been included as "new" in the statistics were actually old functions, which have changed argument's list. The designers agreed that these functions may introduce risks but with considerably less exposure than if these functions were indeed newly developed. Hence we disregarded the argument's list of functions in our measurement. Figure 1 shows the complexity evolution of functions in 5 service releases of the telecom product. Each line on the figure represents a C/C++ function.

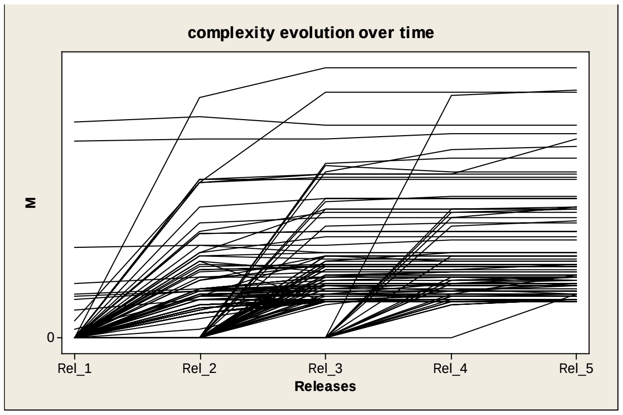


Figure 1. Evolution of complexity for functions with large complexity delta for one release and subsequent service releases in Telecom product

Measuring the evolution of McCabe's complexity M through releases in this manner resulted in:

- Observation that it is the newly developed functions which drive complexity increase between two major releases, as shows in Table 2.
- Observation that the majority of functions that are created complex keep the complexity at the same level over many releases – e.g. see Figure 1.

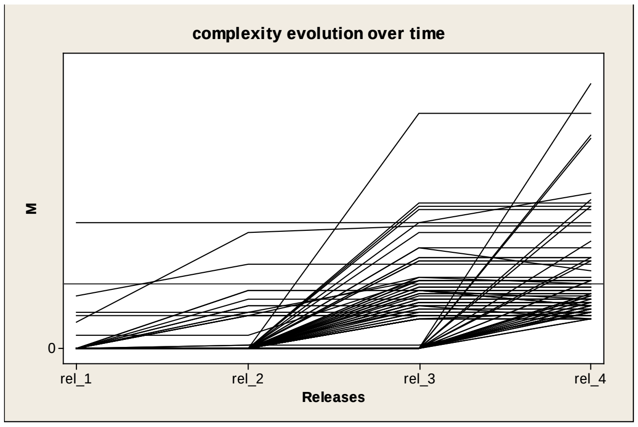


Figure 2. Evolution of complexity for functions with large complexity delta for four releases in product ECU of trucks

Figure 2 shows the complexity development of ECU of trucks for 4 releases.

The trends presented in Figure 2 are similar to the trends in Figure 1 and the number of functions in the diagram reflects the difference in size of the products.

Table 2 presents the results of complexity change between two service releases. The dashes in the table, under **old M** column indicate that the functions did not exist in the previous measurement point. The table shows that there are few functions that are new and already complex. In this particular measurement interval there are also 5 functions that were removed from the release. These functions are indicated by dashes under **new M** column (not shown in Figure 1). The results were consistent for all service releases for the telecom product, irrespective if there was a new functionality development or correction caused by customer's feedback. As opposed to the telecom product the number of newly introduced complex functions was dependent on whether a new end-to-end feature is implemented for truck. In Figure 2 we can see that for ECU software after the first release the number of functions with increased complexity is 5, whereas from second and third release there are many of them.

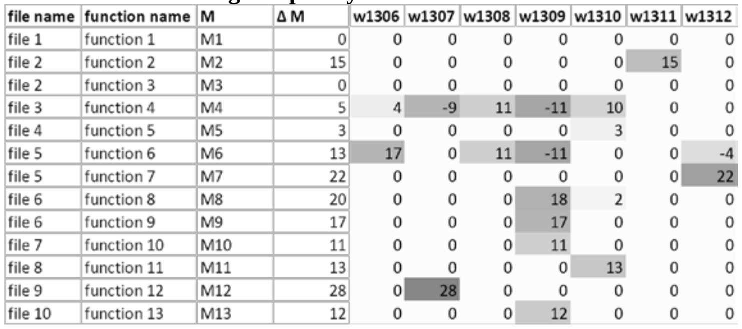
Table 2. Top functions of telecom product with highest complexity change between two service releases

file name	function name	old M	new M	Δ M
file 1	function 1	25	-	-25
file 2	function 2	83	-	-83
file 2	function 3	26	-	-26
file 3	function 4	57	90	33
file 4	function 5	27	-	-27
file 5	function 6	22	-	-22
file 5	function 7	-	25	25
file 6	function 8	-	30	30
file 6	function 9	-	51	51
file 7	function 10	-	23	23
file 8	function 11	-	26	26
file 9	function 12	-	26	26
file 10	function 13	-	22	22
file 11	function 14	-	27	27

In both products new complex functions appeared over time regardless the development time period. We investigated the reasons for high complexity of newly introduced functions in each release (both service and main) and unchanged complexity of existing functions. We observed that both companies assure that the most complex functions are maintained by the most skilled engineers to reduce the risks of faultiness. One of these functions was function 4 in Table 2, which between two releases increased the complexity significantly from an already high level. We observed the change of complexity for both long time intervals (between main releases) and for short time intervals (one week). Table 3 shows how the complexity of functions changes over weeks. The initial complexity of functions is provided under column **M** in the table (the real numbers are not provided for confidentiality reasons).We can see the week numbers on the top of the columns, and every column shows the complexity growth of functions in that particular week. Under **ΔM** column we can see the overall delta complexity per function that is the sum of weekly deltas per function.

The fact that the complexity of these functions fluctuates irregularly was interesting for the designers, as the fluctuations indicate active modifications of functions, which might be due to new feature development or represent defect removals with multiple test-modify-test cycles. Functions 4 and 6 are such instances illustrated in Table 3.

Table 3. Visualizing complexity evolution of functions over weeks



file name	function name	M	Δ M	w1306	w1307	w1308	w1309	w1310	w1311	w1312
file 1	function 1	M1	0	0	0	0	0	0	0	0
file 2	function 2	M2	15	0	0	0	0	0	15	0
file 2	function 3	M3	0	0	0	0	0	0	0	0
file 3	function 4	M4	5	4	-9	11	-11	10	0	0
file 4	function 5	M5	3	0	0	0	0	3	0	0
file 5	function 6	M6	13	17	0	11	-11	0	0	-4
file 5	function 7	M7	22	0	0	0	0	0	0	22
file 6	function 8	M8	20	0	0	0	18	2	0	0
file 6	function 9	M9	17	0	0	0	17	0	0	0
file 7	function 10	M10	11	0	0	0	11	0	0	0
file 8	function 11	M11	13	0	0	0	0	13	0	0
file 9	function 12	M12	28	0	28	0	0	0	0	0
file 10	function 13	M13	12	0	0	0	12	0	0	0

4.2 Correlation Analyses

When adding new measures to our analyses we needed to evaluate how the measures relate to each other by performing correlation analyses. However, in order to correlate the measures we need to define all the measures for the same entity (e.g. for a file or for a function, see Table 1). The correlation analysis for the telecom product is presented in Table 4.

Table 4. Correlation of measures for telecom product

	File M	File Δ M	Max ΔM	NR	ND
NCLOC	**0.9**	0.27	0.33	0.56	0.47
File M		0.28	0.32	0.48	0.41
File Δ M			**0.77**	0.24	0.25
Max Δ M f				0.35	0.37
NR					**0.92**

The correlations which are over 0.7 are in boldface, since it means that the correlated variables characterize the same aspect of the code. Table 5 presents the Pearson correlation coefficients between measures for the ECU for a truck. The correlations are visualized using correlograms in Figure 3 and Figure 4.

Table 5. Correlation of measures for ECU of truck

	File M	File Δ M	Max ΔM	NR	ND
NCLOC	**0.9**	0.43	0.48	0.61	0.38
File M		0.48	0.5	0.68	0.4
File Δ M			**0.84**	0.13	0.19
Max ΔM f				0.3	0.23
NR					0.46

The tables show that the M change is weakly correlated with NRs for both products. This was expected by the designers as the files with the most complex functions are usually maintained by certain designers and do not need many changes. The files with smaller complexity are not risky since they are easy to be modified. The designers noted that the really risky files are those which contain multiple complex functions that change often.

The strong correlation visible in the tables and diagrams above of NCLOC and M has been manifested by a number of other researchers previously [32], [33], [8].

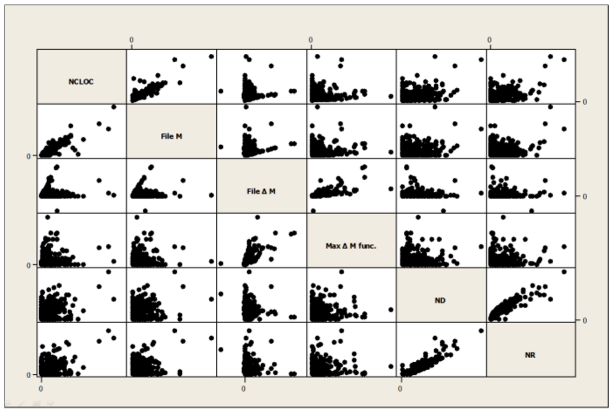


Figure 3. Correlogram of measures for telecom software

The original complexity definition is for a function as a measurement unit, thus we did correlation analyses on function's level. The results were:

- Correl. (M; NCLOC) = 0.76 telecom product
- Correl. (M; NCLOC) = 0.77 truck's software product

The correlation coefficient was weaker compared to correlation between the complexity of a file, which was caused by the fact that we measure the complexity of each file as a sum of complexities of all of its functions. This means that larger files with functions of small complexity will result in higher correlation. Designers claimed that there are many files having moderately complex functions that are solving independent tasks, which did not mean that the file is risky. This resulted in that we used the measure of complexity delta of functions rather than files in our measurement system as a complementary base measure.

Another important observation was the strong correlation between the number of designers and the number of revisions for telecom product Figure 3. Although at the beginning of this study the designers in the reference group believed that a developer of a file might check-in and check-out the file several times which probably is not a problem.

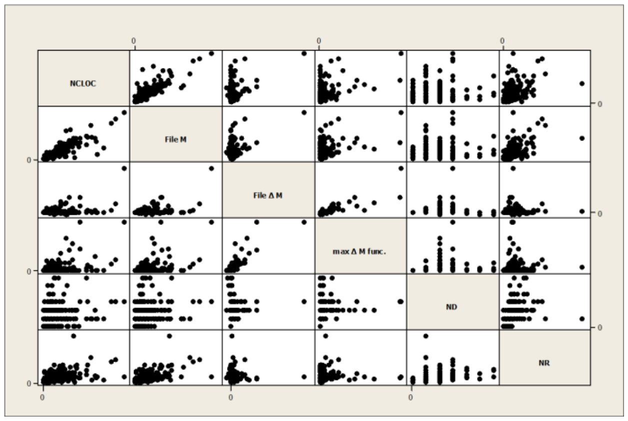


Figure 4. Correlogram of measures for ECU software

They assumed that large number of revisions itself is not as large problem as when many different designers change the file in parallel. This parallel development most likely increase the risk of being uninformed of one another's activities between different developers. The high correlation between **File ΔM** and **max ΔM** shows that the complexity change of the file is mainly due to complexity change of the most complex function in that file. A later observation showed that most of the files contain only one or two complex functions along with many other simple ones.

4.3 Design of the Measurement System

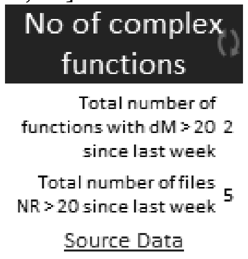
Based on the results that we obtained from investigation of complexity evolution and correlation analyses, we designed two indicators based on M and NR measures. These indicators capture the evolution of complexity and highlight potentially problematic files over time. These indicators were designed according to ISO/IEC 15959. An example definition of one indicator is presented in Table 6.

Table 6. ISO/IEC 15939 definition of the complexity growth indicator

Information Need	Monitor cyclomatic complexity evolution over development time
Measurable Concept	Complexity development of delivered source code
Relevant Entities	Source code
Attributes	McCabe's cyclomatic complexity of C/C++ functions
Base Measures	Cyclomatic complexity number of C/C++ functions – M
Measurement Method	Count cyclomatic number per C/C++ function according to the algorithm in CCCC tool
Type of measurement method	Objective
Scale	Positive integers
Unit of measurement	Execution paths over the C/C++ function
Derived Measure	The difference of cyclomatic number of a C/C++ function in one week development time period
Measurement Function	Subtract old cyclomatic number of a function from new one: ΔM = M(week) – M(week-1)
Indicator	Complexity growth: *The number of functions that exceeded McCabe complexity of 20 during the last week*
Model	Calculate the number of functions that exceeded cyclomatic number 20 during last week development period
Decision Criteria	If the number of functions that have exceeded cyclomatic number 20 is different than 0 then it indicates that there are functions that have exceeded established complexity threshold. This suggests the need of reviewing those functions, finding out the reasons of complexity increase and refactoring if necessary

The other indicator is defined in the same way: the number of files that had NR > 20 during last week development time period.

The measurement system was provided as a gadget with the necessary information updated on a weekly basis (Figure 5). The measurement system relies on two previous studies carried out at Ericsson [34, 35].


No of complex functions

Total number of functions with dM > 20 since last week 2

Total number of files NR > 20 since last week 5

Source Data

Figure 5. Information product for monitoring ΔM and NR metrics over time

For instance the total number of files with more than 20 revisions since last week is 5 (Figure 5). The gadget provides the link to the source file where the designers can find the list of files or functions and the color-coded tables with details.

We visualized the NR and ΔM measures using tables as depicted in Table 3. Presenting the ΔM and NR measures in this manner enabled the designers to monitor those few most relevant files and functions at a time out of several thousands. As in Streamline development the development team merged builds to the main code branch in every week it was important for the team to be notified about functions with drastically increased complexity (over 20). This table drew the attention of designers to the potentially problematic functions on a weekly basis – e.g. together with a team meeting.

5 Threats to Validity

In this paper we evaluate the validity of our results based on the framework described by Wohlin et al. [36]. The framework is recommended for empirical studies in software engineering.

The main external validity threat is the fact that our results come for an action research. However, since two companies from different domains (telecom and automotive) were involved, we believe that the results can be generalized to more contexts than just one company.

The main internal validity threat is related to the construct of the study and the products. In order to minimize the risk of making mistakes in data collection we communicated with reference groups at both companies to validate the results.

The limit 20 for cyclomatic number established as a threshold in this study does not have any firm empirical or theoretical support. It is rather an agreement of skilled developers of large software systems. We suggest that this threshold can vary dependent on other parameters of functions (block depth, cohesion, etc.). The number 20 is a preliminary established number taking into account the number of functions that can be handled on weekly basis by developers.

The main construct validity threats are related to how we match the names of functions for comparison over time. The measurement has been in the following way: We measured the M complexity number of all functions for two consequent releases, registering in a table function name and file name that the function belongs to. We register the class name of the functions also if it is a C++ function. Then we compare the function's, file's and class' names of registered functions for two releases. If there is a function that has the same registered names in both releases we consider that they are the same functions and calculate the complexity number variance for them.

Finally the main threat to conclusion validity is the fact that we do not use inferential statistics to monitor relation between the code characteristics and project properties, e.g. number of defects. This was attempted during the study but the data in defect reports could not be mapped to individual files, this jeopardizing the reliability of such an analysis. Therefore we chose to rely on the most skilled designers' perception of how fault-prone and unmaintainable code is delivered.

6 Conclusions

In Agile and Lean software development quick feedbacks on developed code and its complexity is crucial. With small software increments there is a risk that the complexity of units of code or their size can grow to unmanageable extensions through small increments.

In this paper we explored how complexity changes by studying two software products – one telecom product at Ericsson and one software for electronic control unit at Volvo GTT. We identified that in short periods of time a few out of tens of thousands functions have significant complexity increase. In large products software development teams need automated tools to identify these potentially problematic functions. We also identified that the self-organized teams should be able to make the final assessment whether the "potentially" problematic is indeed problematic.

By analyzing correlations we found that it is enough to use two measures – McCabe complexity and number of revisions – to draw attention of the teams and to designate files as "potentially" problematic.

The automated support for the teams was provided in form of a MS Sidebar gadget with the indicators and links to statistics and trends with detailed complexity development. The method was validated on a set of historical releases.

In our further work we intend to extend our validation to products under development and evaluate which decisions are triggered by the measurement systems. We also intend to study how the teams formulate the decisions and monitor their implementation.

Acknowledgment

The authors thank the companies for their support in the study. This research has been carried out in the Software Centre, Chalmers, University of Gothenburg and Ericsson AB, Volvo Group Truck Technology.

References

[1] B. Boehm, "A view of 20th and 21st century software engineering," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 12-29.

[2] T. Little, "Context-adaptive agility: managing complexity and uncertainty," *Software, IEEE,* vol. 22, pp. 28-35, 2005.

[3] J. Bosch and P. Bosch-Sijtsema, "From integration to composition: On the impact of software product lines, global development and ecosystems," *Journal of Systems and Software,* vol. 83, pp. 67-76, 1// 2010.

[4] S. Henry and D. Kafura, "Software structure metrics based on information flow," *Software Engineering, IEEE Transactions on,* pp. 510-518, 1981.

[5] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on,* pp. 308-320, 1976.

[6] B. Curtis, "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics," *IEEE Transactions on Software Engineering,* vol. SE-5, p. 96.

[7] M. H. Halstead, *Elements of software science* vol. 19: Elsevier New York, 1977.

[8] G. K. Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *Software Engineering, IEEE Transactions on,* vol. 17, pp. 1284-1288, 1991.

[9] R. P. L. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 121-130.

[10] Y. Wang, "On the Cognitive Complexity of Software and its Quantification and Formal Measurement," *International Journal of Software Science and Computational Intelligence (IJSSCI),* vol. 1, pp. 31-53, 2009.

[11] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering,* 2006, pp. 452-461.

[12] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel, "Early quality prediction: A case study in telecommunications," *Software, IEEE,* vol. 13, pp. 65-71, 1996.

[13] B. Ramamurthy and A. Melton, "A synthesis of software science measures and the cyclomatic number," *Software Engineering, IEEE Transactions on,* vol. 14, pp. 1116-1121, 1988.

[14] M. Shepperd and D. C. Ince, "A critique of three metrics," *Journal of Systems and Software,* vol. 26, pp. 197-210, 9// 1994.

[15] T. Fitz. (2009). *Continuous Deployment at IMVU: Doing the impossible fifty times a day*. Available: http://timothyfitz.wordpress.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/

[16] T. Chow and D.-B. Cao, "A survey study of critical success factors in agile software projects," *Journal of Systems and Software,* vol. 81, pp. 961-971, 2008.

[17] G. I. U. S. Perera and M. S. D. Fernando, "Enhanced agile software development - hybrid paradigm with LEAN practice," in *International Conference on Industrial and Information Systems (ICIIS)*, 2007, pp. 239-244.

[18] H. Zhang, X. Zhang, and M. Gu, "Predicting defective software components from code complexity measures," in *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, 2007, pp. 93-96.

[19] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, 2002, pp. 87-94.

[20] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, 2008, pp. 181-190.

[21] D. Wisell, P. Stenvard, A. Hansebacke, and N. Keskitalo, "Considerations when Designing and Using Virtual Instruments as Building Blocks in Flexible Measurement System

Solutions," in *IEEE Instrumentation and Measurement Technology Conference*, 2007, pp. 1-5.

[22] International Bureau of Weights and Measures., *International vocabulary of basic and general terms in metrology = Vocabulaire international des termes fondamentaux et généraux de métrologie*, 2nd ed. Genève, Switzerland: International Organization for Standardization, 1993.

[23] J. Lawler and B. Kitchenham, "Measurement modeling technology," *IEEE Software,* vol. 20, pp. 68-75, 2003.

[24] Predicate Logic. (2007, 2008-06-30). *TychoMetrics*. Available: http://www.predicatelogic.com

[25] F. Garcia, M. Serrano, J. Cruz-Lemus, F. Ruiz, M. Pattini, and ALARACOS Research Group, "Managing Software Process Measurement: A Meta-model Based Approach," *Information Sciences,* vol. 177, pp. 2570-2586, 2007.

[26] Harvard Business School, *Harvard business review on measuring corporate performance*. Boston, MA: Harvard Business School Press, 1998.

[27] D. Parmenter, *Key performance indicators : developing, implementing, and using winning KPIs*. Hoboken, N.J.: John Wiley & Sons, 2007.

[28] A. Sandberg, L. Pareto, and T. Arts, "Agile Collaborative Research: Action Principles for Industry-Academia Collaboration," *IEEE Software,* vol. 28, pp. 74-83, Jun-Aug 2011 2011.

[29] R. L. Baskerville and A. T. Wood-Harper, "A Critical Perspective on Action Research as a Method for Information Systems Research," *Journal of Information Technology,* vol. 1996, pp. 235-246, 1996.

[30] G. I. Susman and R. D. Evered, "An Assessment of the Scientific Merits of Action Research," *Administrative Science Quarterly,* vol. 1978, pp. 582-603, 1978.

[31] P. Tomaszewski, P. Berander, and L.-O. Damm, "From Traditional to Streamline Development - Opportunities and Challenges," *Software Process Improvement and Practice,* vol. 2007, pp. 1-20, 2007.

[32] G. Jay, J. E. Hale, R. K. Smith, D. Hale, N. A. Kraft, and C. Ward, "Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship," *Journal of Software Engineering and Applications (JSEA),* 2009.

[33]M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal,* vol. 3, pp. 30-36, 1988.

[34] M. Staron, W. Meding, G. Karlsson, and C. Nilsson, "Developing measurement systems: an industrial case study," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 23, pp. 89-107, 2011.

[35] M. Staron and W. Meding, "Ensuring reliability of information provided by measurement systems," in *Software Process and Product Measurement*, ed: Springer, 2009, pp. 1-16.

[36] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslèn, *Experimentation in Software Engineering: An Introduction*. Boston MA: Kluwer Academic Publisher, 2000.

Configuring Software for Reuse with VCL

Dan Daniel[1,*], Stan Jarzabek[2], and Rudolf Ferenc[1]

[1]Department of Software Engineering
University of Szeged, Hungary
{danield,ferenc}@inf.u-szeged.hu
[2]School of Computing
National University of Singapore, Singapore
dcssj@nus.edu.sg

Abstract. Preprocessors such as cpp are often used to manage families of programs from a common code base. The approach is simple, but code instrumented with preprocessing commands may become unreadable and difficult to work with. We describe a system called VCL (variant configuration language) that enhances cpp to provide a better solution to the same problem. The main extensions have to do with propagation of parameters across source files during VCL processing, the ability to adapt source files for reuse depending on the reuse context, and the ability to form general templates to represent any group of similar program structures (methods, functions, classes, files, directories) in generic, adaptable form. In the paper, we describe salient features of VCL, explain how they alleviate some of the problems of cpp, and illustrate reuse capabilities of VCL with an example.

1 Introduction

Preprocessors are often used to manage families of programs from a common code base. In the paper, we focus on cpp which is a most commonly used preprocessor. Variant code relevant to some but not all family members appears in the code base under commands such as `#ifdef` for selective inclusion into family members that need that code. preprocessor parameters (`#define`) control the process of configuring the code base to build a specific family member.

There are well-known problems involved in managing large number of configuration options in the code base with cpp [10,16,12,11]. As the number of configuration options grows, programs instrumented with cpp macros become difficult to understand, test, maintain and reuse. It is difficult to figure out which code is related to which options, and to understand or change program in general. Managing configuration options with `#ifdef`s is technically feasible, but is error-prone and does not scale. Karhinen et al. observed that management of configuration options at the implementation level only is bound to be

[*] This work was done during author's internship at National University of Singapore.

complex [10]. They described problems from Nokia projects in which preprocessing and file-level configuration management were used to manage configuration options. They proposed to address variability at the higher level of program design to overcome these problems. Similar problems with preprocessing were also reported in a research project FAME-DBMS [12,6].

Today's mainstream approach to reuse is motivated by the above experiences. Much emphasis is placed on architectural design as means to manage product variants in reuse-based way [11,5]. Still, mappings between features, reusable components and specific variation points in components affected by features are often complex. Problems magnify in the presence of feature dependencies, when the presence or absence of one feature affects the way other features are implemented [4]. Feature dependencies lead to overly complex conditions under `#if`, or many nesting levels under `#ifdef` macros.

Despite many benefits of architecture- and component-based approaches to reuse, managing features that have fine-grained impact on many reusable components requires extensive manual, error-prone customizations during product derivation [13]. Therefore, it is common to use variation mechanisms such as preprocessing, configuration files or wizards, in addition to component/architecture design, to manage features at the level of the component code.

We describe a system called VCL (variant configuration language) that enhances cpp to provide a better solution to configuring a code base for reuse. The main extensions have to do with propagation of parameters across source files during VCL processing, the ability to adapt code for reuse depending in the reuse context, and the ability to represent any group of similar program structures (methods, functions, classes, files, directories) in generic, adaptable form.

This paper describes how VCL works. In Section 2, we describe salient features of VCL and comment on how our extensions alleviate some of the problems of cpp. In Section 3 we describe the most commonly used VCL commands, and how the VCL processor works. In Section 4, we lead the reader through an example that illustrates reuse capabilities of VCL. Concluding remarks close the paper.

2 An Overview of VCL

VCL is an improved and enhanced version of XVCL [17]. Like XVCL, VCL is based on Bassett's Frame Technology [3]. XVCL is a dialect of XML and uses XML trees and parser for processing. VCL parts with XML syntax and processing, and offers a flexible, user-defined syntax. VCL syntax is based on cpp, just because cpp is so widely used and we see many good reasons and benefits for cpp users to try VCL.

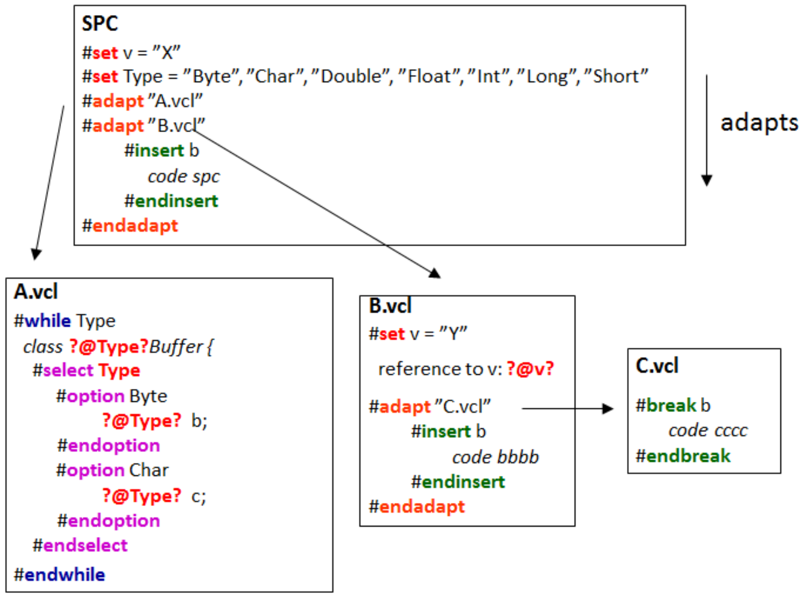
The overall scheme of VCL operation is similar to that of cpp: The goal is to manage a family of program variants from a common code base. Program variants are similar, but also differ one from another in variant features. VCL organizes and instruments the source files for ease of configuring variant features into the base. VCL commands appear at distinct variation points in the code base at which configuring occurs.

As compared to cpp, VCL leads to more generic, more reusable code base, giving programmers better control over the process of configuring variant features into the code. VCL's ability to organize code base in a way that replaces any significant pattern of repetition with a generic, adaptable VCL representation, leads to much smaller code base and simpler to work with. The main differences between VCL and cpp are the following:

- **VCL #adapt *file* command** is like cpp `#include`, except that with `#adapt` the same source *file* can be customized differently in different contexts in which it is reused (i.e., adapted). Any kind of differences among those custom versions of a *file* can be handled by VCL. There are no technical limits of when and how to reuse source files. However, for reuse to be cost-effective, it is wise to reuse only if specifications of *file* customizations are reasonably simple.
- **VCL variables** assigned values in `#set` commands are like cpp variables assigned values in `#define` commands, except that VCL variable values propagate to all adapted source files (along `#adapt` links). In addition, the variable propagation mechanism is subject to overriding rules that are supportive to effective reuse of source files in multiple contexts.
- **VCL #while command** allows us to define code generation loops. Suppose we have 20 similar source code structures f_i in our system (where f_i can be a function, class method, class, file, or directory). If the differences among f_i are not too extreme, it pays off to define a generic code structure F in VCL representation. Then we set up a `#while` loop to generate 20 instances f_i by adapting F. Generated code can be conveniently placed in the directories and files of our choice.

Each XVCL command has a direct counterpart in VCL with the same meaning. Based on XVCL usage experience, besides simplified and more readable syntax we introduced the following enhancements:

- **Expanding the customization options under #adapt command:** In XVCL, the only command that can be placed under `#adapt` is `#insert`. In VCL, it is possible to use any other VCL command here. Using `#set`, `#while` and `#select` commands under `#adapt` proved to be particularly useful.
- **Specification of output files:** Rather than specifying output file per `#adapt` or per file as it was the case in XVCL, we introduced a separate command to control where VCL Processor is to emit output. Details about `#output` command can be found in section 3.4
- **Robust structure instead of unreadable loops:** while loops using many multi-value variables can be quite confusing. We introduced a structure called set-loop which gives us the possibility to store and use more multi-value variables together as one loop descriptor data structure.
- **Flexible syntax:** It is possible that VCL command words conflict with reserved words in the target language. For this case, we introduced the ability



```
SPC
#set v = "X"
#set Type = "Byte","Char","Double","Float","Int","Long","Short"
#adapt "A.vcl"
#adapt "B.vcl"
        #insert b
            code spc
        #endinsert
#endadapt
```

```
A.vcl
#while Type
  class ?@Type?Buffer{
    #select Type
        #option Byte
                ?@Type?  b;
        #endoption
        #option Char
                ?@Type?  c;
        #endoption
    #endselect
#endwhile
```

```
B.vcl
#set v = "Y"
  reference to v: ?@v?
#adapt "C.vcl"
        #insert b
            code bbbb
        #endinsert
#endadapt
```

```
C.vcl
#break b
      code cccc
#endbreak
```

Fig. 1. Salient VCL commands

to easily change any VCL command's syntax. This way the users can define their own syntax.

VCL Processor starts processing with the specification (SPC) file. VCL commands in the SPC, and in each subsequently adapted file, are processed in the order in which they appear. Whenever the processor encounters an `#adapt "A.vcl"` command, processing of the current file is suspended and the Processor starts processing the file A.vcl. Once processing of the file A.vcl is completed, the Processor resumes processing of the current file from the location just after the `#adapt "A.vcl"` command. In that way processing ends when VCL Processor reaches the end of the SPC file.

In the example in Figure 1, `#set` command declares variables. VCL variables parametrize code and also control the flow of processing. Loop command `#while` is controlled by a multi-value variable (in the above example 'Type'). Any reference to variable 'Type' in the i^{th} iteration of the loop fetches the i^{th} value of the variable. Variable 'Type' also controls selection of one of the options under `#select` command in file A.vcl, namely the `#option` whose value matches the current value of variable 'Type' is selected for processing. VCL `#insert` command inserts its contents into any matching `#break`. `#insert` plays a similar role to weaving aspect code in Aspect-Oriented Programming [1]. The reader will find a more detailed explanation of VCL commands in the next section.

3 VCL Commands

3.1 #adapt Command

Figure 2 shows how `#adapt` commands control the processing flow of the source files instrumented with VCL. VCL Processor starts at the first line of the SPC.

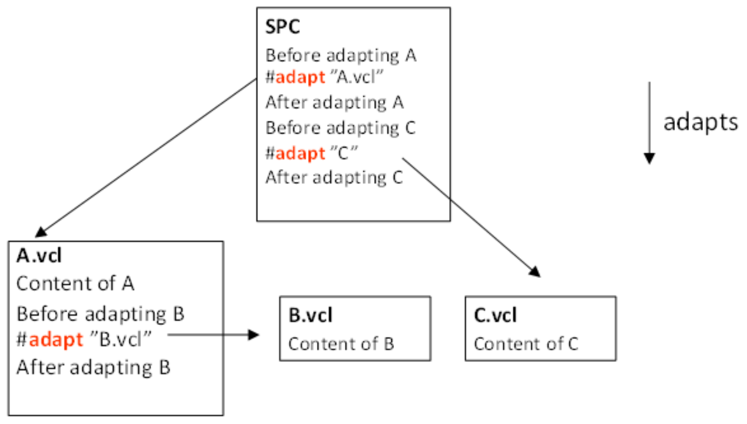


Fig. 2. Processing the adapt commands

In Figure 2, this is text "Before adapting A". The Processor emits the text to the output file and then executes command `#adapt "A.vcl"`. This suspends processing of SPC and transfers processing to the file A. VCL Processor emits the text "Content of A" and continues processing this way. At the end of processing we get the following output:

```
Before adapting A
Content of A
Before adapting B
Content of B
After adapting B
After adapting A
Before adapting C
Content of C
After adapting C
```

`#adapt` command may specify customizations that should be applied to the adapted file.

```
#adapt: file
    <customizations>
#endadapt
```

Customizations may include any VCL commands. VCL applies customizations to a designated file and proceeds to processing it.

3.2 #set Command

`#set` command declares a VCL variable and sets its value. `#set` command is similar to cpp's `#define` except that VCL variable values propagate across the files along `#adapt` links. With the `#set` command, we can declare single and multi-value variables. A variable value can be an integer, string or expression. For example:

```
#set x = 5        %assign integer 5 to x
#set y = x        %assign value of x to y
#set z = y + 2    %assign 7 to z
#set a = “text”   %string must be enclosed in double-quotes
```

The value of a multi-value variable is a list of values, for example:

```
#set X = 1,2,2+1
#set Y = ”one”, ”two”, ”three”
```

In the `#set` command, a direct reference to variable x can be written `?@x?` or simply `x`. There are three types of expressions in VCL, namely name, string and arithmetic expressions. Expressions can be used in `#set` commands to assign a value to a new variable, and they may also appear anywhere in the source files.

Name Expression

A name expression can contain variable references (like ?@x?), and combinations of variable references (like ?@x@y@z? or ?@@x?). The value of a name expression is computed by applying the '@' operator from right to left. At each step, the result of application of '@' is concatenated with the rest of the expression. Example 1:

```
#set a = “b”
#set b = 20
?@a?    %value of a
?@@a?   %value of (value of a)
```

Output of the example:

```
b
20
```

Example 2:

```
#set x = “y”
#set y = “z”
#set z = “w”
#set yw = “u”
```

```
#set xu = “q”
?@x@y@z?

%Evaluation steps:
%1: replace @z with its value "w"
%2: replace @yw with its value "u"
%3: replace @xu with its value "q"
```

Output of the example:

```
q
```

String Expression

A string expression can contain any number of name expressions intermixed with character strings. To evaluate a string expression, we evaluate its component name expressions from the left to the right replacing them with their respective values and concatenating with character strings. Example:

```
#set x = “y”
#set y = “z”
#set z = “w”
#set yw = “u”
#set xu = “q”
?@x@y@z?”String”?@xu?

%Evaluation steps:
%1: eval ?@x@y@z? -> “q”
%2: concat ”String”
%3: eval ?@xu? -> “q”
```

Output of the example:

```
qStringq
```

Arithmetic Expression

If an expression is a well-formed arithmetic expression, VCL Processor recognizes it as such and evaluates its value. An arithmetic expression can contain ‘+’, ‘-’, ‘*’, ‘/’ operators and nested parenthesis can be used. An arithmetic expression used in `#set` command must yield to integer. In arithmetic expressions variables can be used by simple reference i.e.:

```
#set b = a * (c + 2)
```

3.3 Propagation of VLC Variable Values

Having executed `#set x = 10`, VCL Processor propagates value of x to all files reached along `#adapt` links. The first executed `#set x` overrides any subsequently found `#set x` commands in adapted files. An exception from the above rule is the situation where two `#set` commands assign values to the same variable in the same file. Example:

```
SPC:
#set x = 1
#adapt "A.vcl"
#set x = 2           %overriding in the same file
#adapt "A.vcl"

File A.vcl:
#set x = 3           %this command will be ignored
Value of x is: ?@x?
```

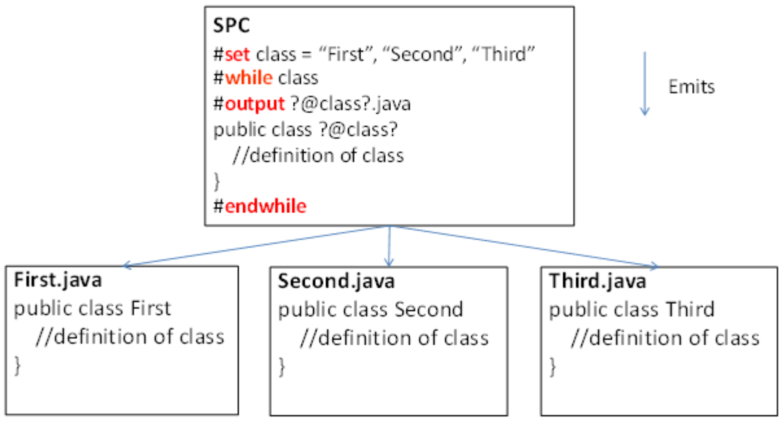
Output of the example:

```
Value of x is: 1
Value of x is: 2
```

3.4 #output Command

VCL Processor interprets VCL commands and emits any source code found in the visited files. VCL `#output <path>` command specifies the output file where the source code should be placed. The `<path>` can be absolute or relative path. If the output file is not specified, then VCL Processor emits code to an automatically generated default file named *defaultOutput* in the main folder of the installed VCL Processor. It is recommended to use the `#output` command.

We can put `#output` command in many files, so that VCL Processor organizes the emitted output as we like. Once `#output f` has been executed, all subsequently emitted text is placed in file `f`, until the next `#output` overrides `f` with another file name, as Figure 3 shows.



SPC
```
#set class = "First", "Second", "Third"
#while class
#output ?@class?.java
public class ?@class?
  //definition of class
}
#endwhile
```

First.java
```
public class First
  //definition of class
}
```

Second.java
```
public class Second
  //definition of class
}
```

Third.java
```
public class Third
  //definition of class
}
```

Fig. 3. Output example

When VCL Processor executes `#output <path>` and the path does not exist, VCL Processor creates relevant folders and file. The first `#output f` in a given processing sequence command deletes any existing `f` and creates a new one. Any subsequent `#output f` command in the same processing sequence appends the new content to the file.

3.5 #while Command

`#while` command is controlled by one or more multi-value variables. The ith value of each of the control variables is used in ith iteration of the loop. This means that all the control variables should have the same number of values, and the number of values determines the number of iterations of the loop. VCL Processor interprets the loop body in each iteration and emits custom text accordingly. Example:

```
#set x = 1,2,3
#set y = "a","b","c"
#while x, y
    Value of x is ?@x? and value of y is ?@y?
#endwhile
```

Output of the example:

```
Value of x is 1 and value of y is a
Value of x is 2 and value of y is b
Value of x is 3 and value of y is c
```

3.6 #select Command

Please refer to the example of Figure 1. `#select` control-variable command is used to select one or more of given options, depending on the value of a control-variable.

VCL Processor checks `#option <value>`-s in sequential order. If the value given in the option clause is the same as the value of the `#select`'s control-variable, the body of that `#option` will be processed. One `#option` clause can specify more values separated with '|' character. For example `#option 1|5` will be executed if the value of the control variable is 1 or 5. A `#select` command can include one `#option-undefined` and one `#otherwise` clause. `#option-undefined` is executed if the control-variable of the `#select` command is not defined, the `#otherwise` is executed if none of the `#option`s matches the value of the control-variable.

VCL Processor selects and processes in turn all the `#option`s whose values match the value of the control variable.

3.7 #insert Command

An `#insert <name>` command replaces all matching `#break` command's content in all files reached via adapt chain with its content. Matching is done by a name. Commands `#insert-before` and `#insert-after` add their content before or after matching `#breaks`, without deleting their content. Any `#break` may be simultaneously extended by any number of `#insert`, `#insert-before` and `#insert-after` commands.

In the following example we demonstrate how insert-break works in VCL. Example:

```
SPC:
#adapt: "A.vcl"
    #insert-before breakX
        inserting before the breakpoint
    #endinsert
    #insert breakX
        inserting into the breakpoint
    #endinsert
    #insert-after breakX
        inserting after the breakpoint
    #endinsert
#endadapt

File A.vcl:
#break: breakX
    default text
#endbreak
```

VCL Processor emits the following output for the above example:

```
inserting before the breakpoint
inserting into the breakpoint
inserting after the breakpoint
```

The content under `#break` is called a default content: If no `#insert` matches a `#break`, then the break's content is processed. The propagation and overriding rules for `#insert` (`#insert-before` and `#insert-after`) are the same as for VCL variables.

4 Java Buffer Library Example

Studies show that even in well-designed programs, we typically find 50%-90% of redundant code contained in program structures (functorial, classes, source files or directories) replicated many times in variant forms. , repeated many times. For example, the extent of the redundant code in the Java Buffer library is 68%

[9], in parts of STL (C++) - over 50% [2], in J2EE Web Portals – 61% [18], and in certain ASP Web portal modules – up to 90% [15].

Redundant code obstructs program understanding during software maintenance. The engineering benefits of non-redundancy become evident especially if we pay attention to large granularity clones. In this section we demonstrate VCL's potential to reduce program complexity by eliminating redundant codes.

4.1 An Overview of the Original Buffer Library

A buffer contains data in a linear sequence for reading and writing [14]. Buffer classes differ in features such as a buffer element type, memory allocation scheme, byte ordering and access mode, as described in [8]. Buffer classes can be found in *java.nio* package. Each legal combination of features yields a unique buffer class. That is why, even though all the buffer classes play essentially the same role, there are 74 classes in the Java Buffer library.

Buffer classes differ one from another in buffer element type (byte, char, int, float, double, long,short), memeory allocation scheme (direct, indirect), byte ordering(native, non-native, big endian, little endian) and access mode (writable, read-only). Classes that differ in certain features are similar one to another. Earlier studies showed that it is difficult to eliminate these redundancies with conventional techniques such as generics and refactorings.

4.2 Buffer Classes in VCL

Representing repeated code with a generic adaptable form is a good approach to make the code smaller and easier to understand. We start by creating groups of similar Buffer classes. For example, classes *ByteBufferR, IntBufferR, LognBufferR...* form a group of similar classes. Figure 4 highlights similarities and differences between classes *HeapByteBufferR* and *HeapIntBufferR*. 71 classes (all classes except *Buffer*, *MappedByteBuffer* and *StringCharBuffer*) can be categorized into seven similartity groups as follows:

– **[T]Buffer:** contains 7 buffer classes of type T (level 1). T denotes one of the buffer element types, namely, Byte, Char, Int, Double, Float, Long, Short
– **Heap[T]Buffer:** contains 7 Heap classes of type T (level 2)
– **Direct[T]Buffer[S|U]:** contains 13 Direct classes (level 2) U denotes native and S - non-native byte ordering.
– **Heap[T]BufferR:** contains 7 Heap read-only classes (level 3).
– **Direct[T]BufferR[S|U]:** contains 13 Direct read-only classes (level 3).
– **ByteBufferAs[T]Buffer[B|L]:** contains 12 classes (level 2) providing views T of a Byte buffer with different byte orderings (B or L). T here denotes buffer element type except Byte. B denotes big endian and L – little endian byte ordering.
– **ByteBufferAs[T]BufferR[B|L]:** contains 12 read-only classes (level 2) providing views T of a Byte buffer with different byte orderings (B or L). T here denotes buffer element type except Byte. B denotes big endian and L – little endian byte ordering.

```
class HeapByteBufferR
    extends HeapByteBuffer
{
    HeapByteBufferR(int cap, int lim) {             // package-private
       super(cap, lim);
       this.isReadOnly = true;
    }
    HeapByteBufferR(byte[] buf, int off, int len) { // package-private
       super(buf, off, len);
       this.isReadOnly = true;
    }
```

```
class HeapIntBufferR
    extends HeapIntBuffer
{
    HeapIntBufferR(int cap, int lim) {              // package-private
       super(cap, lim);
       this.isReadOnly = true;
    }
    HeapIntBufferR(int[] buf, int off, int len) {  // package-private
       super(buf, off, len);
       this.isReadOnly = true;
    }
```

Fig. 4. Similarities and differences between two Buffer classes

We can build a VCL generic representation for each group. This generic representation can then be adapted to form each of the individual classes.

For example, generation of classes in the group *Heap[T]BufferR* is done as follows:

1. We build a so-called meta-class which will lead the generation of all files from this group, in this case this meta-class will be named *Heap[T]BufferR*. In the meta-class we declare the type of the Buffer class (T) as a multi value variable using `#set` command.

   ```
   #set elmtType = "Byte", "Char", "Double", "Float",
   "Int", "Long", "Short"
   ```

2. In a loop command we iterate over variable *elmtType* adapting the common template for all classes using `#adapt` command.

   ```
   #while elmtType
       #adapt Heap[T]BufferR.tmp
   #endwhile
   ```

3. Customizing the adapt command, we insert the unique codes in the template using `#insert` commands. We decide about the insertions based on the value of the variable *elmtType* using `#select command` inside of the `#adapt` command with `#option` and `#otherwise` clauses.

   ```
   #while elmtType
      #adapt: Heap[T]BufferR.tmp
          #select elmtType
   ```

```
                    #option Byte
                         #insert-after moreMethods
                              #adapt byteMoreMethods
                         #endinsert
                     #endoption
                     #option Char
                         #insert-after moreMethods
                              #adapt charMoreMethods
                         #endinsert
                         #insert toString
                              #adapt chartoString
                         #endinsert
                     #endoption
                     #otherwise
                         #insert-after moreMethods
                              #adapt otherMethods
                         #endinsert
                     #endotherwise
                 #endselect
              #endadapt
          #endwhile
```

4. In the template file *Heap[T]BufferR.tmp* we control generation of files using `#output` command, and we give place to `#break` commands for customizing the content.

```
#output "output/Heap"?@elmtType?"BufferR.java"
... //Template content
#break moreMethods
... //Template content
#break: toString
    [Default toString]
#endbreak
```

With this approach we can generate all the classes in the seven groups mentioned earlier.

Bonding together the representation of the seven groups with a specification (SPC) file, we can define a structure that generates the whole Buffer library code. The groups of the similar classes are represented by the meta-classes marked in Figure 5. Meta-methods are representations of similar Java methods and meta-fragments are representations of smaller code fragments. In Figure 5 we indicate adaption of meta-components by a black arrow. Any meta-component can adapt other meta-components, and any meta-component can be easily reused with parametrization.

The original representation of the mentioned 71 classes consists of 16299 lines of code including comments. The representation with VCL consists of 3720 lines of code. With the VCL representation we could eliminate 77.2% of the code using the commonalities between files.

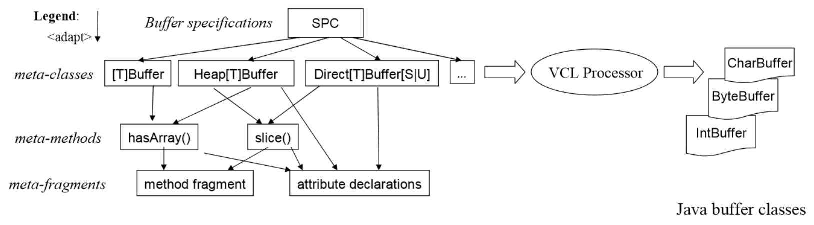


Fig. 5. An overview of the complete solution in VCL

The reader can find complete VCL representation of the Buffer library on our web site [7].

5 Conclusions

We presented a new, improved and enhanced implementation of a variability management technique first implemented in Frame Technology$^{\mathrm{TM}}$ [3] and then popularized as XVCL [17]. Like its predecessors, VCL builds on the tradition of preprocessors such as cpp, but extends them to provide better support for managing program variants from a common base of reusable code. These extensions include propagation of parameters across source files during VCL processing, the ability to adapt code for reuse depending in the reuse context, and the ability to form general templates that represent any group of similar program structures (methods, functions, classes, files, directories) in generic, adaptable form. VCL parts with XML syntax and processing, and offers a flexible, user-defined syntax. VCL offers new constructs that allow programmers to write simpler and clearer code. In the paper, we described salient features of VCL, explained how they alleviate some of the problems of cpp, and illustrated reuse capabilities of VCL with an example.

The power of VCL is mostly in its simplicity and scalability. It is easy to understand, to learn and there are strategies to take conventional programs under control of VCL.

In future work, we plan to conduct experiments on a bigger scale, further refine VCL mechanisms and formulate methodological guidelines for applying VCL.

References

1. Sven Apel, Christian Kästner, Thomas Leich, and Gunter Saake. Aspect refinement-unifying aop and stepwise refinement. *Journal of Object Technology*, 6(9):13–33, 2007.
2. Hamid Abdul Basit, Damith C. Rajapakse, and Stan Jarzabek. Beyond templates: a study of clones in the stl and some general implications. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 451–459, New York, NY, USA, 2005. ACM.

3. P. Bassett. Framing software reuse - lessons from real world, 1997.
4. P. Clements and D. Muthig. Proc. workshop on variability management – working with variation mechanisms, 2006.
5. Paul Clements and Linda Northrop. *Software product lines.* Addison-Wesley Boston, 2002.
6. S. Jarzabek. Effective software maintenance and evolution: Reuse-based approach, 2007.
7. S. Jarzabek and D. Daniel. Variant configuration language. `http://vcl.comp.nus.edu.sg`, 2013.
8. S. Jarzabek and S Li. Unifying clones with a generative programming technique: a case study. *J. Softw. Maint. Evol.: Res. Pract.*, pages 267—-292, 2006.
9. Stan Jarzabek and Li Shubiao. Eliminating redundancies with a "composition with adaptation" meta-programming technique. *SIGSOFT Softw. Eng. Notes*, 28(5):237–246, September 2003.
10. Anssi Karhinen, Alexander Ran, and Tapio Tallgren. Configuring designs for reuse. In *Proceedings of the 19th international conference on Software engineering*, ICSE '97, pages 701–710, New York, NY, USA, 1997. ACM.
11. C. Kastner, S. Apel, and D. Batory. A case study implementing features using aspectj. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 223–232, 2007.
12. Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 311–320, New York, NY, USA, 2008. ACM.
13. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming.* Springer, 1997.
14. Oracle. Buffer javadoc. `http://docs.oracle.com/javase/6/docs/api/java/nio/Buffer.html`, 2011.
15. Ulf Pettersson and Stan Jarzabek. An industrial application of a reuse technique to a web portal product line. *submitted for publication*, 2005.
16. Henry Spencer and Geoff Collyer. Ifdef considered harmful, or portability experience with c news, 1992.
17. National University of Singapore (XVCL) Team. Xml-based variant configuration language. `http://xvcl.comp.nus.edu.sg`, 2011.
18. J. Yang and S. Jarzabek. Applying a generative technique for enhanced reuse on jee platform. *Conf. on Generative Programming and Component Engineering*, (4):237–255, September 2005.

Identifying Code Clones with RefactorErl [⋆]

Viktória Fördős, Melinda Tóth

ELTE-Soft Ltd., Eötvös Loránd University, Budapest, Hungary
{f-viktoria,tothmelinda}@elte.hu

Abstract. Code clones, the results of "copy&paste programming", have a negative impact on software maintenance. Therefore several tools and techniques have been developed to identify them in the source code. However most of them concentrate on imperative, well known languages. In this paper we give an AST/metric based clone detection algorithm for the functional programming language Erlang and evaluate it on an open source project.

1 Introduction

Duplicated code detectors are software [2, 5, 13, 16], which help in the identification of duplicates. Various approaches have been proposed, including the analysis of code tokens [11], the syntax tree built up using the tokens [7], and using different metrics [14]. The majority of these methods and algorithms have been constructed specifically for the most common programming paradigm today, that is imperative programming, and for the leading imperative programming languages.

Imperative programming languages have several duplicates detector algorithms and software, whilst in functional programming only a few exist, such as [9] developed for the Haskell language, and [12] for the Erlang [1] language.

Most duplicated code detection software do not work directly on the source code, but rather on a transformed representation. Such representations include the series of tokens and the abstract syntax tree, from which crucial information for the analysis can be retrieved faster and more efficiently. RefactorErl [4, 8, 18] is a static source code analyser and transformer tool for Erlang, that provides a representation that contains more information about the source beyond that of the abstract syntax tree.

In this paper we show an AST/metric based algorithm for duplicated code detection in Erlang programs. The implementation of this work uses the RefactorErl framework. A short test run is also presented to show the results of the algorithm on open source projects.

[⋆] Supported by Ericsson–ELTE-Soft–ELTE Software Technology Lab

2 Erlang & RefactorErl

2.1 Erlang

Erlang is a declarative, dynamically typed, functional, concurrent programming language, which was designed to develop soft real-time, distributed applications.

The compilation unit of Erlang programs is called a *module*, which is built up from attributes and function definitions. The encapsulating module, the name of the function, and the arity of the function can identify a function uniquely in Erlang. Pattern matching features are a prominent way to define functions by case. The cases of a function definition are called *function clauses*, and they are separated from each other by `;` token. A one-arity function which consists of two function clauses is shown in Erlang source 1. This function will be our running example through out the paper.

```
clone_fun(L) when is_list(L)->
    ShortVar = L,
    A = 1,
    B = lists:max([I || I<-lists:seq(1, 10)]),
   (A == 1) andalso throw(badarg),
   self ! B;

clone_fun(_)->
   V = f(g(42)),
   LongVariableName = V,
   B = lists:max([J || J<-lists:seq(V, V*2)]),
   X = fun(E) -> E + B end,
   self ! X.
```

Erlang source 1: clone_fun/1 function definition form

A function clause is built up from either one expression, called the *top-level expression*, or a sequence of top-level expressions as defined in the Erlang grammar. There are no statements in Erlang, only expressions. Contrary to statements, every expression has a value, which is the value of its last top-level expression.

Other branching expressions, such as `case`, `if`, `receive` and the `try` expression, are also built up from clauses, for which the previous statements, analogously hold, too.

2.2 RefactorErl

The main aim of RefactorErl is to support the daily work of Erlang programmers with both code comprehension and refactoring tools. It provides the ability to retrieve semantic information and metric values about the source code, to perform dependency analysis and to visualise the results of the analysis. It facilitates code reorganisation with clustering algorithms and several refactoring methods. The incremental and asynchronous analyser architecture allows the programmer to track source code changes. The tool has multiple user interfaces to choose from. These include a web-based interface, an interactive console or one can use Emacs or Vim with RefactorErl plugins.

The source code has to be loaded into RefactorErl in order to be analysed. The code is first transformed into a series of tokens using whitespace- and layout-preserving lexical analysis, and is then passed on to the RefactorErl preprocessor. After preprocessing, the abstract syntax tree is constructed from the token series based on Erlang syntactical rules. Next, semantic analysers decorate the AST with attributes and links, resulting in a graph, called *Semantic Program Graph*, which is the internal data model of RefactorErl.

The labelled vertices of the Semantic Program Graph are the lexical, syntactic and semantic units of the source code, while the directed edges between them represent the lexical, syntactic and semantic relations between the entities. Information from the Semantic Program Graph is gathered by the evaluation of path expressions and traversal of the graph. For this purpose, RefactorErl provides a complete, high-level API.

The algorithm presented in this paper uses information from the Semantic Program Graph and metrics of RefactorErl.

3 Clone IdentifiErl

In this section, we present a new algorithm for clone detection. Our algorithm combines a number of existing techniques, but introduces a novel filtering component, as described in Section 3.4. As to our current knowledge these techniques have never been used specifically in Erlang.

What does clone detection mean intuitively? One may try to compare every code fragment to every other. The original representation of a code is too particular, thus generalisation is needed and the generalised form of code needs to be used. The similarity of each pair of code fragments can be represented by a matrix. The first component of our algorithm produces this matrix, which is detailed in Section 3.2. From this matrix, the *initial clones* can be extracted along diagonals. This is what the second component of our algorithm does, which is described in Section 3.3. Irrelevant clones can be found among these clones, which are removed by evaluating filters which are described in Section 3.4. The ideas behind each filter have been based on case studies using Mnesia[3].

3.1 Unit

The generalisation part of the first component of the algorithm is described in this section.

Choosing the unit The unit of a clone instance has to be chosen as cautiously as possible. One of our goals was to design and construct an algorithm that can be successfully used on legacy code, so the source code of several Erlang programs were studied.

The abstraction level of Erlang is extremely high. Due to this abstraction, an application written in Erlang is so brief that 1 line of Erlang code can be expressed with 8 to 10 lines of C code generally. It follows that block-based algorithms cannot be used. It also follows that the size of the chosen unit should be small. Tokens and sub-expressions are small enough to be selected. However, they are too small to be used efficiently. A function clause is not small enough, therefore the top-level expression becomes the unit of the algorithm.

Transforming the unit The program text of a top-level expression is considered too particular, thus generalisation is needed. A good idea is to use a formal alphabet over a formal language which can cover unneeded specialisations of the tokens. Algorithm 1 shall be used for generalising.

```
function TransformWithAlphabet(TopLevelExpression)
      # $\epsilon$ is the empty word
    $Word \leftarrow \epsilon$
    for all $Token \in$ Tokenizer(TopLevelExpression) do
          # '$\cdot$' operator expresses the concatenation between words
        $Word \leftarrow Word \cdot$ WordOverAlphabet(Token)
    end for
    return $Word$
end function
```

Algorithm 1: Algorithm of the alphabet

A generalised top-level expression is a sentence over the fixed alphabet that is made of the concatenation of words. Every word is produced by the function `WordOverAlphabet` based on the type of the token. Tokens are produced by tokenizing expressions in the same order as given by the lexical analyser. It is necessary to preserve this order to keep the characteristics of the original expression. The alphabet of the language is not injective, in order to cover unneeded differences, for example, the difference between a variable and a constant (either a number or an atom).

Example After generalisation, our running example will be the same as shown in Figure 1. What we can see there, that every top-level expression got indexed and generalised.

3.2 Matrix

How do code clones occur? Usually, they are the result of "copy&paste programming". For example, let us assume, that one has copied a three-unit long

Index	Top-level expression	Generalised top-level expr.
	`clone_fun(L) when is_list(L)->`	
i-1	`ShortVar = L,`	`A=A`
i	`A = 1,`	`A=A`
i+1	`B = lists:max([I || I<-lists:seq(1, 10)]),`	`A=A:A([AlAvA:A(A,A)])`
i+2	`(A == 1) andalso throw(badarg),`	`(AfA)FA(A)`
i+3	`self ! B;`	`A!A`
	`clone_fun(_)->`	
j-1	`V = f(g(42)),`	`A=A(A(A))`
j	`LongVariableName = V,`	`A=A`
j+1	`B = lists:max([J || J<-lists:seq(V, V*2)]),`	`A=A:A([AlAvA:A(A,A*A)])`
j+2	`X = fun(E) -> E + B end,`	`A=x(A)zA+Ae`
j+3	`self ! X.`	`A!A`

Fig. 1. Demonstrating the transformation part of the first component

sequence and has modified the second unit of the sequence, but the order of the sequence has been kept.

Usually larger clones are preferred, so we want to collect the three-unit long sequence as one clone instead of collecting two one-unit long clones. To be able to do it, modifications should be handled flexibly. Our algorithm works primarily on a matrix, which is a view of the problem, with which the flexibility criteria can be satisfied. Each element of the matrix expresses the similarity between two expressions and while a clone is made by preserving its original, correct order of its elements, the diagonals of a matrix are enough to be focused on. In other words, the fragments of diagonals are completely isomorphic to the fragments of code sequences found in the code directly. We put this idea in perspective in the following subsections.

Introducing the matrix Let us assume that every top-level expression is numbered (indexed) sequentially, as shown in Figure 1. By taking the cardinality of the indexes as the size (denoted by n), a square matrix can be constructed, whose elements express similarity between the defining rows and columns, which are the top-level expressions identified by their indexes.

The relation, denoted by *Similarity*, between two top-level expressions, satisfies the following properties:

- *Similarity* is a binary relation.
- *Similarity* is reflexive, namely all values are related to themselves.
- *Similarity* is symmetric.
- *Similarity* expresses the equivalence of two top-level expressions in a significant manner.

If the symmetric property holds, then only the elements of the lower triangular matrix need to be computed. If the reflexive property also holds, it follows that the elements of the main diagonal do not need to be computed. With these two properties the volume of computation is slightly reduced to the following cardinality:

$$\frac{1}{2}n^2 - n$$

Clone IdentifiErl uses Dice-Sørensen metric[10, 17] for determining similarity, which does satisfy the properties of *Similarity* relation, too. The authors see no reason why the metric should not be replaced with other string similarity metric. Let Dice-Sørensen metric be portrayed by the m function

$$m : \mathit{String} \times \mathit{String} \to [0 \ldots 1] \subset \mathbb{R}$$

Let n be the cardinality of the top-level expressions, A be the n- sized, square matrix. Let *selecttle* be a selector function which returns the top-level expression indexed by the given index. Now, the matrix can be exactly defined:

$$A(i,j) ::= \begin{cases} m(\mathit{selecttle}(i), \mathit{selecttle}(j)) & \text{if } i,j \in [1 \ldots n], i < j; \\ 0 & \text{otherwise.} \end{cases}$$

Example Let us consider the following code fragments that are shown in Figure 1 with indexes. By using Dice-Sørensen metric the matrix can be constructed, whose relevant part is shown in Figure 2.

$$\begin{array}{c|ccccccc}
 & 1 & i-1 & i & i+1 & i+2 & i+3 & n \\
\hline
1 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
j-1 & \vdots & 0.5 & 0.5 & 0.43 & 0.46 & 0 & \vdots \\
j & \vdots & \mathbf{1.0} & \mathbf{1.0} & 0.21 & 0 & 0 & \vdots \\
j+1 & \vdots & 0.19 & 0.19 & \mathbf{0.94} & 0.23 & 0 & \vdots \\
j+2 & \vdots & 0.17 & 0.17 & 0.22 & 0.24 & 0 & \vdots \\
j+3 & \vdots & 0 & 0 & 0 & 0 & \mathbf{1.0} & \vdots \\
n & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots
\end{array}$$

Fig. 2. Similarities are represented by a matrix

Patterns in the matrix What we expect, that the clauses are clones of each other, except that the `(i-1)`-th line differs from the `(j-1)`-th line and the `(i+2)`-th line also differs greatly from `(j+2)`-th line. Therefore, it can be said, that 3 clones are present: the first one is a one-unit long pair, namely `([i-1], [j])`, the second one is also a one-unit long pair, namely `([i+3], [j+3])` and the third one is a two-unit long pair, namely `([i, i+1], [j,j+1])`.

What we expect, that the following pairs are related to each other according to relation *isClone*:

$$\{\ldots, (i-1,j), (i,j), (i+1,j+1), (i+3,j+3), \ldots\} = \mathit{isClone}$$

In practice, the one-unit long clone pairs are not interesting and multi-unit long clone pairs should be focused on.

Let us assume that the starting units of a k-unit long clone pair can be found on the a-th, and b-th indexes (k is a positive, fixed integer). Then

$$\{(a+i, b+i) \mid i \in [0 \ldots k-1] \subset \mathbb{Z}\} \subseteq isClone$$

As observed by Baxter [6], every pair in the defined set is an element of the matrix, and based on a k-unit long clone pair one of the diagonals of the matrix can be partially formed.

What we cannot find in inter-diagonals (inside a diagonal) is the following. Let us assume that the first clause of `clone_fun/1` is the same as shown in Figure 1, but its second clause contains one newly inserted top-level expression. The new definition of `clone_fun/1` is shown in Figure 3.

Index	**Top-level expression**	**Generalised top-level expr.**
	`clone_fun(L) when is_list(L)->`	
i-1	`ShortVar = L,`	`A=A`
i	`A = 1,`	`A=A`
i+1	`B = lists:max([I || I<-lists:seq(1, 10)]),`	`A=A:A([AlAvA:A(A,A)])`
i+2	`(A == 1) andalso throw(badarg),`	`(AfA)FA(A)`
i+3	`self ! B;`	`A!A`
	`clone_fun(_)->`	
j-1	`V = f(g(42)),`	`A=A(A(A))`
j	`LongVariableName = V,`	`A=A`
j+1	`B = lists:max([J || J<-lists:seq(V, V*2)]),`	`A=A:A([AlAvA:A(A,A*A)])`
j+2	`X = fun(E) -> E + B end,`	`A=x(A)zA+Ae`
j+3	`Y = lists:zip([1,2,3],[3,21]),`	`A=A:A([A,A,A],[A,A,A])`
j+4	`self ! X.`	`A!A`

Fig. 3. The new definition of `clone_fun/1`

The `([i+3],[j+4])` clone pair and the `([i, i+1], [j,j+1])` clone pair are in different diagonals. If the instances of a clone differ from each other in that way, then the full clone cannot be collected from the same diagonal, for instance, when the cardinality of inserted, deleted or rewritten top-level expressions differ from each other.

To summarise, instead of finding any pattern in the matrix, it is enough to search in diagonals. Although a full clone cannot be collected from the same diagonal in every case, its parts can be collected from different diagonals.

3.3 Determining initial clones

In this section, we describe a parallel, efficient algorithm for determining initial clones.

As demonstrated in a previous example with Figure 3, a clone may be divided into sub clones due to insertions, deletions or other kinds of modifications. It would be practical if a full clone could be gathered somehow, therefore we need to add a new parameter, called the *invalid sequence length*. An invalid sequence length is the maximum length of a sequence whose middle elements can differ too much from each other. This limitation to the elements is naturally needed because of the beginnings and the endings of the clones should be similar to each other. By introducing invalid sequence length, one can customise the allowable maximum deviation of a clone.

If the chosen metric is exactly a distance, its values should be normalised to the 0 to 1 interval to be able to handle the threshold correctly.

Now, we are able to define exactly the *isClone* relation which expresses whether two units are considered to be clones of each other. The Dice-Sørensen metric is portrayed by the m function, and $Threshold$ contains a previously defined non-negative real number less than one. Let *isClone* be a general Boolean function operating on string pairs.

$$isClone : String \times String \to \mathbb{L}$$

The truth set of this function is:

$$\lceil isClone \rceil ::= \{(a, b) | a \in String, b \in String, m(a, b) > Threshold\}$$

As shown in Section 3.2, it is enough to focus only on the diagonals, thus, if the set of diagonals is constructed first, the elements of the set can be computed in parallel, because every element of the matrix is affected by only one complete diagonal.

Let n be the cardinality of the top-level expressions, then the set of diagonals is the following:

$$Diagonals ::= \{\langle (i, 1), (i + 1, 2) \ldots, (n, (n - i + 1)) \rangle \mid i \in [2 \ldots n]\}$$

Working with diagonals has a deficiency: the gathered instances of a clone can overlap the natural boundaries of the clone. The overlap should be avoided if possible, so a boundary needs to be defined as a trimming rule of the production of initial clones, as follows: every top-level expression of a clone must belong to the same function clause per instance. This rule works, because function clauses act like natural boundaries.

Algorithm 2 for calculating the initial clones is detailed below. The inputs of the algorithm are the followings:

- N is the cardinality of the top-level expressions,
- T is the value of the threshold,
- $InvSeqLength$ is the maximum length of a sequence which is built-up with invalid items.

```
function $\textsc{InitialClonesBasedOnDiagonals}(N, T, InvSeqLength)$
    $Diagonals \leftarrow \{\langle(i,1),(i+1,2)\ldots,(N,(N-i+1))\rangle \mid i \in [2\ldots N]\}$
    parallel for all $Diagonal \in Diagonals$ do
        $InitialClones \leftarrow \emptyset$
         # $\langle\rangle$ is the empty sequence
        $InitialClone \leftarrow \langle\rangle$
        $InvSeqCount \leftarrow 0$
        for all $Index \in Diagonal$ do
            $TlePair \leftarrow \textsc{SelectTlePairs}(Index)$
            if $\textsc{isClone}(TlePair, T)$ then
                if $\textsc{isSameClone}(InitialClone, TlePair)$ then
                     # $\oplus$ operator express the concatenation between sequences
                    $InitialClone \leftarrow InitialClone \oplus \langle TlePair\rangle$
                    $InvSeqCount \leftarrow 0$
                else
                    $InitialClones \leftarrow InitialClones \cup \textsc{Trim}(InitialClone)$
                    $InitialClone \leftarrow \langle TlePair\rangle$
                    $InvSeqCount \leftarrow 0$
                end if
            else
                if $InvSeqCount < InvSeqLength\ \wedge$
                    $\textsc{isSameClone}(InitialClone, TlePair)$ then
                    $InitialClone \leftarrow InitialClone \oplus \langle TlePair\rangle$
                    $InvSeqCount \leftarrow InvSeqCount + 1$
                else
                    $InitialClones \leftarrow InitialClones \cup \textsc{Trim}(InitialClone)$
                    $InitialClone \leftarrow \langle\rangle$
                    $InvSeqCount \leftarrow 0$
                end if
            end if
        end for
        return $InitialClones$
    end parallel for
end function
```

Algorithm 2: Parallel algorithm of the initial clones detector

The output of the algorithm is a set of the initial clones which are produced in parallel, so a union is needed to be constructed from them by "adding" them together.

Every diagonal is calculated in parallel in Algorithm 2, where the local variables are independent from each other, so no interference can happen between the parallel processes.

The algorithm never enters an infinite loop; it always terminates due to the iterations which are based on items, which are pre-calculated, cannot be expanded and also the cardinality of the items decreases at each iteration.

– `SelectTlePairs` is a function returning a pair of indexed top-level expressions, whose indexes are given as input.

- `isClone` is a function which determines if the given pair can form a clone by examining whether the calculated metric is greater then the given threshold.
- `isSameClone` is a function which determines if the given pair can be appended to the actual clone (`InitialClone`). A pair can be appended to the given clone only if the top-level expressions of the resulting clone belong to the same function clause per instance. This limitation is needed, because as mentioned above, overlapping must be avoided.
- `Trim` is a function which trims the beginnings and endings of the given pair. It is needed because invalid items may occur in the forming sequence of a clone. Invalid items are only allowed in the middle of the sequence, as described above, so the beginnings and endings of a pair must be cut out.

Example The three initial clones which are detected by the described algorithm with using 1 for invalid sequence length are shown below:

1. `LongVariableName = Var` and `ShortVar = L`
2. `A = 1,`
   `B = lists:max([I || I<-lists:seq(1, 10)]),`
   `(A == 1) andalso throw(badarg),`
   `self ! B` and
   `LongVariableName = Var,`
   `B = lists:max([J || J<-lists:seq(Var, Var*2)]),`
   `X = fun(E) -> E + B end,`
   `self ! X`
3. `A = 1` and `ShortVar = L`

Alternative method A more efficient way might be to exploit transitivity in calculating the elements of the matrix. In order to do so, we need to replace the string similarity metric with a transitive relation.

Let us assume that *isClone* is a binary relation between duplicates. Two items are duplicates of each other if the value of the Dice-Sørensen metric (DC) computed for them is greater then 0.3. The relation *isClone* is not transitive because of the intransitivity of the string similarity metric, consider the following example:

$$\begin{aligned}
DC("aabb","bbcc") &= 0.33 > 0.3 &\Longrightarrow\quad& ("aabb","bbcc") \in isClone,\\
DC("bbcc","ccdd") &= 0.33 > 0.3 &\Longrightarrow\quad& ("bbcc","ccdd") \in isClone,\\
DC("aabb","ccdd") &= 0.0 < 0.3 &\Longrightarrow\quad& ("bbcc","ccdd") \notin isClone.
\end{aligned}$$

3.4 Filtering and trimming unit

A parallel algorithm for a new filtering system is detailed in this section.

In practice, the set of initial clones is too large and contains many false positive or irrelevant clones, therefore further operations are needed to narrow down the result set. An example for an irrelevant clone can be `A = 1` and `X = 5`.

First of all, note the difference between one-unit long, and multi-unit long clones. Due to the high abstraction level of the alphabet and the usage of the similarity metrics, lots of false positive clones appear in the result set of the production of initial clones, if only the one-unit long clones are taken into consideration. It follows that the filters on one-unit long clones need to be stronger than the filters on the multi-unit long clones.

In Section 3.3, invalid sequence length is used as a new parameter of the algorithm. This parameter is also used in the filtering unit to process the multi-unit long clones. During the filtering, it can happen that a multi-unit long clone is split into a one-unit long clone and the rest of a multi-unit long clone. In this case, the one-unit long clone has to be also processed by the filters that are relevant for one-unit long clones.

As mentioned in Section 2.2, RefactorErl provides ready-to-use source code metrics and a Semantic Program Graph which is rich in information and easy to query. Thus every filter concentrates only on one characteristic of the code, computed by using the libraries of RefactorErl.

A clone appears in the result set of the algorithm only if it meets all the requirements which are stated in the corresponding filters. For all *clone* in *InitialClones*, we have:

$$\bigwedge_{Filter \in Filters} Filter(clone) \implies clone \in ResultClones$$

If only one clone is examined in each iteration, a parallel algorithm, called `FilteringAndTrimmingUnit`, can be constructed along initial clones. If the currently examined clone is a one-unit long clone, then the `FiltersForOneLongs` function is responsible for dealing with it, otherwise the `FiltersForMultiLongs` function is the one in charge.

The `FiltersForOneLongs` function forms the conjunction of the results of evaluated filters, which are dedicated for one-unit long clones. If the conjunction is true, then the examined clone is returned, otherwise an empty set is returned.

The `FiltersForMultiLongs` function, which focuses only on the multi-unit long clones, is detailed in Algorithm 3.

The input of the `FilteringAndTrimmingUnit` algorithm is the set of the initial clones and the invalid sequence length.

The algorithm always terminates, because the cardinality of the unprocessed items decreases at each iteration.

The output of the algorithm is a set of clones which are produced in parallel, so a union is needed to be constructed from them.

A bit more explanation is needed for the `FurtherTrim` function. This function is responsible for trimming invalid items from the beginnings and endings of the given clone. The result of a trimming is a set, whose one-unit long elements are further filtered by the `FiltersForOneLongs` to check whether the examined clone fulfils the stronger filters.

```
function FILTERSFORMULTILONGS($Clone, InvSeqLength$)
    $Clones \leftarrow \emptyset$
    $AClone \leftarrow \langle\rangle$
    $InvSeqCount \leftarrow 0$
    for all $UnitPair \in Clone$ do
        if $\wedge_{FilterFun \in FilterFuns*}$ FILTERFUN($UnitPair$) then
            $AClone \leftarrow AClone \oplus \langle UnitPair \rangle$
            $InvSeqCount \leftarrow 0$
        else
            if $InvSeqCount < InvSeqLength$ then
                $AClone \leftarrow AClone \oplus \langle UnitPair \rangle$
                $InvSeqCount \leftarrow InvSeqCount + 1$
            else
                $Clones \leftarrow Clones \cup$ FURTHERTRIM($AClone$)
                $AClone \leftarrow \langle\rangle$
                $InvSeqCount \leftarrow 0$
            end if
        end if
    end for
    $Clones \leftarrow Clones \cup$ FURTHERTRIM($AClone$)
    return $Clones$
end function
```

Algorithm 3: Filtering and trimming unit of the multi-unit long clones

Used filters The ideas behind filters were based on separate case studies on the results of the algorithm on a real life application, called Mnesia. Mnesia is a database management system and belongs to the standard Erlang/OTP library. It is written in Erlang. There are three types of filters in Clone IdentifiErl:

- **Filters for one-unit long clones.**
  These filters try to eliminate such pairs, which are basic expressions, or which are match or send expressions having basic right sides. Basic expressions are atoms, integers, floats, chars, variables, lists, tuples, record operations or function applications. It may seem to be too strict, but nobody takes care of these clones.
- **Filters for multi-unit long clones.**
  These filters work similarily to the ones in the previous group, but they are not so strict.
- **Filters for any clones.**
  These filters focus on different branching expressions and list comprehensions. If the cardinalities of clauses, the function applications or the head expressions of the list comprehensions differ from each other then the examined clone is not needed.

Example From the three initial clones, only the four-unit long clone is the result of the algorithm, the two one-unit long clones are filtered out. These clones are object-lessons for irrelevant clones.

3.5 Short test run on Mnesia

Clone IdentifiErl has been implemented to the best of our knowledge, it is also extremely specialised on Erlang.

Clone IdentifiErl was tried out on Mnesia, which has 22594 effective lines of code. (The number of empty lines is not included in the sum.) It consists of 31 modules, 1687 functions, 5393 top-level expressions.

Clone IdentifiErl detected 801 clone pairs in Mnesia around 120 seconds. Neither irrelevant nor false positive clones were found. The main types of the clones are duplicated configurations, handler branchings, debugging sequences, constructions of validator functions and message processings. A non-trivial example is shown below.

```
Left one (found in mnesia_loader):
    case ?catch_val(send_compressed) of
        {'EXIT', _} ->
            mnesia_lib:set(send_compressed, NoCompression),
            NoCompression;
        Val -> Val
    end
Right one (found in mnesia_controller):
    case ?catch_val(no_table_loaders) of
        {'EXIT', _} ->
            mnesia_lib:set(no_table_loaders,1),
            1;
        Val -> Val
    end
```

3.6 Comparison with Wrangler

Wrangler is a refactoring tool for Erlang which introduces a duplicated code detection algorithm [12]. They build a suffix tree, and calculate the code clones based on this representation.

It is hard to compare our result to the result of Wrangler, because the tools work on different granularity of the source code. Wrangler can identify code clones inside top-level expressions, while the smallest unit of the analysis of RefactorErl is a top-level expression. Thus Wrangler can identify smaller clones. However Clone IdentifiErl can identify larger code clones because of the $i$nvalid sequence length and filter out the non-relevant clones.

4 Related work

Usually, duplicated code detectors consist of two phases: the first phase is responsible for making the internal representation and the second phase collects clones from this representation. The techniques used by duplicated code detection software are the concrete realisation of these components. The techniques for each component can be chosen independently as long as their composition is well typed.

The simplest approach to the first phase is *line-based detection*. It occurs when the selected unit is the line of the source code. This method is infrequent in practice, therefore further discussion is disregarded.

The most commonly used techniques are the token and AST based methods. *Token-based detection* [11] uses lexical units of the source code as base units. Tokens are transformed according to their characteristics over an abstract alphabet over a formal language. Clone detection algorithms can be performed on this representation, or an extended suffix tree can be constructed from the transformed token resulting in the suffix tree becoming the set of clone candidates, where all the occurrences of every duplicate can be gathered as a sub-suffix tree. This technique is used by Wrangler [12] and the previous, unstable, unfinished prototype within RefactorErl.

Syntax/metric based detection [7] usually comes in two variants. Block-based methods use program blocks as unit of the instance, whilst function-based methods use function bodies. The source code is partitioned and transformed according to the chosen unit. Usually the transformation results in a hash value or fingerprint. Even a sequence database can be constructed from the transformed units. The algorithm described in [15] operates on the blocks created from the sequence of statements, and uses fingerprints and sequence database for detection of clones.

Another possibility is the transformation of the sub-trees into simple values, thereby flattening the syntax tree, and using composition to produce the candidates. In the case, that the syntactical structure chosen for the units is too large, it is suggested to pair every unit with every other to form the clone candidates. The constructed abstract syntax tree should also be preserved, as the information inside may be used by the detection algorithm to provide more precise results. This technique is used by [14] by having functions as base units.

5 Conclusions

Duplicated code detection is a special static analysis, where code clones (either identical, or similar) are identified in the source code. Code clones can result several bugs and inconsistency during software maintenance.

In this paper we have described and evaluated an own duplicated code detection algorithm to identify code clones in Erlang programs. We have shown the three main parts of the algorithm: candidate production, initial clone detection, trimming and filtering possibilities. We use the representation of Erlang programs defined by RefactorErl (a static analyser and transformer tool) to build the internal representation and to calculate metric values. We have also evaluated our technique on open source projects.

Acknowledgement

The authors would like to thank Julia Lawall and Christian Rinderknecht for their useful advices.

References

1. Erlang Programming Language. http://erlang.org.
2. Finding Duplicate Code by using Code Clone Detection. http://msdn.microsoft.com/en-us/library/hh205279.aspx.
3. Mnesia Reference Manual. http://www.erlang.org/doc/apps/mnesia/.
4. RefactorErl Homepage. http://plc.inf.elte.hu/erlang.
5. Simian - Similarity Analyser. http://www.harukizaemon.com/simian/.
6. B. S. Baker. A program for identifying duplicated code. In *Computer Science and Statistics: Proc. Symp. on the Interface*, pages 49–57, March 1992.
7. I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377, 1998.
8. I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Kőszegi, T. M., and M. Tóth. RefactorErl - Source Code Analysis and Refactoring in Erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*, pages 138–148, Tallin, Estonia, October 2011.
9. C. Brown and S. Thompson. Clone Detection and Elimination for Haskell. In J. Gallagher and J. Voigtlander, editors, *PEPM'10: Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 111–120. ACM Press, January 2010.
10. L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, July 1945.
11. T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002.
12. H. Li and S. Thompson. Clone detection and removal for erlang/otp within a refactoring environment. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '09, pages 169–178, New York, NY, USA, 2009. ACM.
13. Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.
14. J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 244–253, 1996.
15. S. H. Randy Smith. Detecting and Measuring Similarity in Code Clones. *IWSC*, 2009.
16. S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. ACM.
17. T. Sørensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons. *Biol. Skr.*, 5:1–34, 1948.
18. M. Tóth and I. Bozó. Static analysis of complex software systems implemented in Erlang. In *Proceedings of the 4th Summer School conference on Central European Functional Programming School*, CEFP'11, pages 440–498, Berlin, Heidelberg, 2012. Springer-Verlag.

Code Coverage Measurement Framework for Android Devices

Szabolcs Bognár[1], Tamás Gergely[1], Róbert Rácz[1], Árpád Beszédes[1], and Vladimir Marinkovic[2]

[1] University of Szeged, Department of Software Engineering
{bszabi,gertom,rrobi,beszedes}@inf.u-szeged.hu
[2] University of Novi Sad, Faculty of Technical Sciences vladam@uns.ac.rs

Abstract. Software testing is a very important activity in the software development life cycle. Numerous general black- and white-box techniques exist to achieve different goals and there are a lot of practices for different kinds of software. The testing of embedded systems, however, raises some very special constraints and requirements in software testing. Special solutions exist in this field, but there is no general testing methodology for embedded systems. One of the goals of the CIRENE project was to fill this gap and define a general testing methodology for embedded systems that could be specialized to different environments. The project included a pilot implementation of this methodology in a specific environment: on an Android-based Digital TV receiver (Set-Top-Box).
In this pilot, we implemented method level code coverage measurement of Android applications. This was done by instrumenting the applications and creating a framework for the Android device that collected basic information from the instrumented applications and communicated it through the network towards a server where the data was finally processed. The resulting code coverage information was used for many purposes according to the methodology: test case selection and prioritization, traceability computation, dead code detection, etc.
In this paper, we introduce this pilot implementation and, as a proof-of-concept, present how the coverage results were used for different purposes.

1 Introduction

Software testing is a very important quality assurance activity of the software development life cycle. With testing, the risk of a residing bug in the software can be reduced, and by reacting to the revealed defects, the quality of the software can be improved. Testing can be performed in various ways. Static testing – for example – can be performed on any workproducts of the project; it includes the manual checking of documents and the automatic analysis of the source code without executing the software. During dynamic testing the software or a specific part of the software is executed. Many dynamic test design techniques exist, the two most well known groups among them are black-box and white-box techniques.

Black-box test design techniques concentrate on testing functionalities and requirements by systematically checking whether the software works as intended and produces the expected output for a specific input. The techniques take the software as a black box, examine "what" the program does without having any knowledge on the structure of the program, and they are not intrerested in the question "how?".

On the other hand, white-box testing examines the question "How does the program do that?", and tries to exhaustively examine the code from several aspects. This exhaustive examination is given by a so-called coverage criterion which defines the conditions to be fulfilled by the set of instruction sequences executed during the tests. (E.g. 100% instruction coverage criterion is fulfilled if all instructions of the program are executed during the tests.) Coverage measures give a feedback on the quality of the tests themselves.

The reliability of the test can be improved, by combining black-box and white-box techniques. During the execution of test cases generated from the specifications using black-box techniques, white-box techniques can be used to measure how completely the actual implementation is checked. If necessary, reliability of the tests can be improved by generating new test cases for the not verified code fragments.

1.1 Specific problems with embedded system testing

Testing in embedded environments has special attributes and characteristics. Embedded systems are neither uniform nor general-purpose. Each embedded system has its own hardware and software configuration typically designed and optimized for a specific task, which affects the development activities on the specific system. Development, debugging, and testing are more difficult since different tools are required for different platforms.

However, high product quality and testing that ensures this high quality is very important as the correction of residual bugs can be very difficult for these systems. For example, the software of a digital TV with play-from-USB capabilities fails to recover after opening a specific media file format and this bug can only be repaired by replacing the ROM of the TV. Once the TVs are produced and sold, it might be impossible to correct this bug without spending a huge amount of money on logistic issues. Although there are some solutions aiming at the uniformisation of the software layers of embedded systems (e.g. the Android platform [1]), there has not been a uniform methodology for embedded systems testing.

1.2 The CIRENE project

One of the goals of the CIRENE project [2] is to fill this gap and define a general testing methodology for embedded systems that copes with the above mentioned specialities and whose parts can be implemented on specific systems. The methodology combines black-box tests responsible for the quality assesment of the system under test and white-box tests responsible for the quality assesment of the tests themselves. Using this methodology the reliability of the test results and the quality of the embedded system can be improved. As a proof-of-concept, the CIRENE project included a pilot implementation of the methodology for a specific, Android-based digital Set-Top-Box system. Although the proposed solution was developed for a specific embedded environment, it can be used for any Android-based embedded devices such as smart phones or more general-purpose tablets.

The methodology specialized to the Set-Top-Box in the pilot implementation can be seen on Figure 1. The coverage measurement toolchain plays an important role in the methodology. Many coverage measurement tools (e.g. EMMA [3]) exist that are not specific but can be used on Android applications. However, these are applicable only during the early development phases as they are able to measure code coverage on the development platform side. This kind of testing ommits to test the real environment, misses the hardware-software co-existance issues which can be essential in embedded systems. We are not aware of any common toolchain that measures code coverage directly on Android devices.

Our coverage measurement toolchain starts with the instrumentation of the application we want to test, which allows us to the measure code coverage of the given application during test execution. As the device of the pilot project runs the Java-based Android operation system, Java instrumentation techniques can be used. Then, the test cases are executed and the coverage information is collected. In the pilot implementation, the collection is split between the Android device and the used testing tool RT-Executor [4]: the service collects the information from the individual applications of the device, while the testing tool processes the information (through its plug-ins).

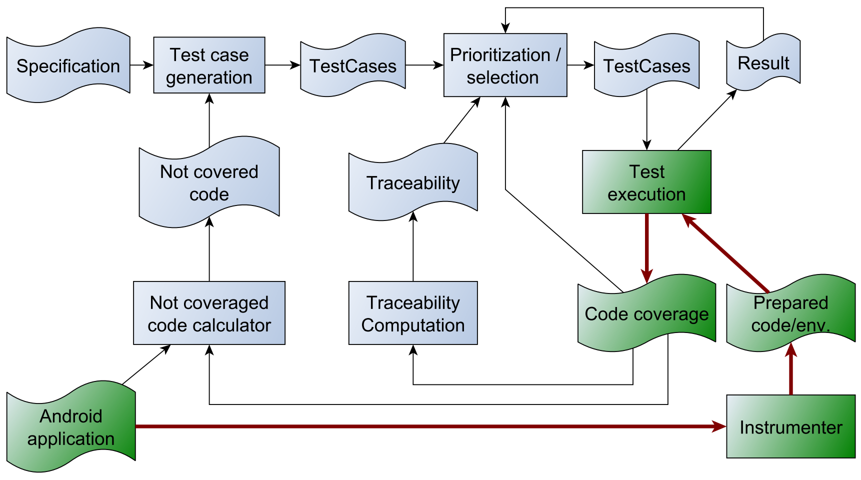


Fig. 1. Coverage collection methodology on the Set-Top-Box

The coverage information gathered with the help of the coverage framework can be utilized by many applications in the testing methodology. They can be used for selecting and prioritizing test cases for further test executions, or for helping to generate additional test cases if the coverage is not sufficient. It is also useful for dead code detection or traceability links computation.

The rest of the paper is organized as follows. In Section 2, we give an overview on the related work. In Section 3, the implementation of the coverage measurement framework is presented. In Section 4, some use cases are presented to demonstrate the usefulness of coverage information. In Section 5, we summarize our achievements and elaborate on some possible future works.

2 Related Work

Software testing is a very important activity during the software development process. It helps reducing the risk of residual bugs and so contributes to the quality of the released software. Different testing techniques can be categorized by many criteria. One of these categories contain the dynamic testing methods where testing includes the execution of the program under test. There are two well known groups of dynamic testing techniques: black-box and white-box testing techniques. While black-box techniques help to assess the quality of the software under test, white-box techniques rather assess the quality of the executed test sets. A good test includes a wide range of testing techniques, combines them to lessen the weaknesses of the individual methods, and utilizes the advantages of the combination. For example, tests prepared using black-box techniques are usually measured for code coverage (a white-box technique), which helps to estimate the remaining risks more accurately.

In the CIRENE project, one of our first tasks was to assess the state-of-the-art in embedded systems testing techniques with special attention to the combined use of black and white-box techniques. We prepared a technical report on it [5]. In this paper, we report only a few number of combined testing techniques that have been specialized and implemented in the embedded environment.

Gotlieb and Petit presented a path-based test case generation method [6]. They used symbolic program execution and did not execute the software on the embedded device prior to the test case definitions. We use code coverage measurement of real executions to determine information that can be used in test case generation.

José et *al.* defined a new coverage metric for embedded systems to indicate instructions that had no effect on the output of the program [7]. Their implementation used source code instrumentation and worked for C programs at instruction level, and had a great influence on the performance of the program. Biswas et *al.* also utilized C code instrumentation in embedded environment to gather profiling information for model-based test case prioritization [8]. We use binary code instrumentation at method level, use traditional metric that indicates whether the method is executed during the test case or not, and our solution has a minimal overhead on execution time. The resulting coverage information can also be used for test case selection and prioritization.

Hazelwood and Klauser worked on binary code instrumentation for ARM-based embedded systems [9]. They reported the design, implementation and applications of the ARM port of the Pin, a dynamic binary rewriting framework. However, we are working with Android systems that hides the concrete hardware architecture but provides a Java-based one.

There are many solutions for Java code coverage measurement. For example, EMMA [3] provides a complete solution for tracing and reporting code coverage of Java applications. However, it is, as well as others are general solutions not concerning the specialities of Android or any embedded systems.

Most of the coverage measurement tools utilize code instrumentation. In Java-based systems, byte code instrumentation is more popular than source code instrumentation. There are many frameworks providing instrumenting functionalities (e.g. DiSL [10], InsECT [11,12], jCello [13], BCEL [14], etc.) for Java. These are very similar to each other regarding their provided functionalities. We chose Javassist [15] to be our instrumentation framework in the pilot project.

3 Coverage Measurement Toolchain

The implemented coverage measurement toolchain consists of several parts. First, the applications selected for measurement have to be prepared. The preparation process includes program instrumentation that inserts extra code in the application so that the application can produce the information necessary for tracing its execution path during the test executions. The modified applications and the environment that helps collect the results must be installed on the device under test.

Next, tests are executed using this measurement environment and the prepared applications, and coverage information is produced. In general, test execution can be either manual or automated. In the current implementation, we use the *RT-Executor* [4] for test automation. The RT-Executor is a black-box test automation tool developed for testing multimedia devices by RT-RK corporation in Novi Sad [16]. During the execution of the test cases, the instrumented applications produce their traces which are collected, and coverage information is sent back to the automation tool.

Third, the coverage information resulted from the previous test executions is processed and used for different purposes e.g. for test selection and prioritization, additional test case generation, traceability computation, and dead code detection.

In the rest of this section, we describe the technical details of the coverage measurement toolchain.

3.1 Preparation

In order to measure code coverage, we have to prepare the environment and/or the programs under test to produce the necessary information on the executed items of the program. In our case, the Android system uses the Dalvik virtual machine to execute the applications. Although modifying this virtual machine to produce the necessary information would result in a more extensive solution that would not require the individual preparation of the measured applications, we decided not to do so, as we assumed that modifying the VM itself had higher risks than modifying the individual applications. With individual preparation it is much easier to decide what to measure and at what level of details. So, we decided to individually prepare the applications to be measured. As we were interested in method level granularity, the methods of the applications were instrumented before test execution, and this instrumented version of the application was installed on the device. In addition, a service application serving as a communication interface between the tested applications and the network was also necessary to be present on the device.

Instrumentation During the instrumentation process, extra instructions are inserted in the code of the application. These extra instructions should not modify the original functionality of the application except that they are logging the necessary information and slowing down the execution. Instrumentation can be done on the source code or on the binary code.

In our pilot implementation, we are interested in method level code coverage measurement. It requires the instrumentation of each method inserting a code that logs the fact that the method is called. As our targets are Android applications usually available in binary form, we have chosen binary instrumentation.

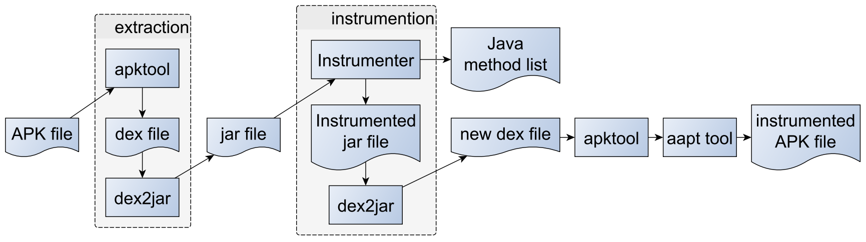


Fig. 2. Instrumentation toolchain

Android is a Java-based system which in our case means that the applications are written in Java language and compiled to Java Bytecode before a further step creates the final Dalvik binary form of the Android application. The transformation from Java to Dalvik is reversible, so we can use Java tools to manipulate the program and instrument the necessary instructions. We used the `Javassist` [15] library for Java bytecode instrumentation, `apktool` [17] for unpacking and repacking the Android applications, the `dex2jar` [18] tool for converting between the Dalvik and the Java program representations, and `aapt` [19] tool for sign the application. The *Instrumentation toolchain* (see Figure 2) is the following:

- The Android binary form of the program needs to be instrumented. It is an `.apk` file (a special Java package, similar to the `.jar` files, but extended with other data to become executable).
- Using the `apktool` the `.apk` file is unpacked and `.dex` file is extracted. This `.dex` file is the main source package of the application, it contains its code in a special binary format. [19,20]
- For all `.dex` files the `dex2jar` is used to convert them to `.jar` format.
- On the `.jar` files we can use the `JInstrumenter`. The `JInstrumenter` is our Java instrumentation tool based on the `Javassist` library [15].
  `JInstrumenter` first adds a new collector class with two responsibilities to the application. On the one hand, it contains a coverage array that holds the numbers indicating how many times the methods (or any other items that is to be measured) were executed. On the other hand, this class is responsible for the communication with the service layer of the measurement framework. Next, the `JInstrumenter` assigns a unique number as ID to each of the methods. This number indicates the method's place in the coverage array of the collector class. Then a single instruction is inserted in the beginning of all methods which updates the corresponding element of the coverage array on all executions of the method.
  The result of the instrumentation is a new `.jar` file with instrumented methods and another file with all the methods' names and IDs.
- The instrumented `.jar` files are converted to `.dex` files using the `dex2jar` tool again.
- Finally, the `.apk` file instrumented application is created by repacking the `.dex` files with the `apktool` and signing it with the `aapt` tool.

During the instrumentation, we give a name to each application. This name will uniquely identify the application in the measurement toolchain, so the service application can identify and separate the coverage information of different applications.

After the instrumentation, the application is ready for installation on the target device.

Service application In our coverage measurement framework implementation it is necessary to have an application that is continuously running on the Android device in parallel with the program under test. During the test execution, this application is serving as a communication interface between the tested applications and the external tool collecting and processing the coverage data. On the one hand this is necessary because of the rights management of the Android systems. Using the network requires special rights from the application and it is much simplier and more controllable to give these rights to only a single application than to all of the tested applications. On the other hand, this solution provides a single interface to query the coverage data even if there are more applications tested and measured simultaneously.

In Android systems, there are two types of applications: “normal” and “service”. Normal applications start, do something while they are visible on the screen, and are destroyed on closing. Services are running in the background continuously and are not destroyed on closing. So, we had to implement this interface application as a service. It serves as a bridge between the Android applications under test and the “external world” as it can be seen on Figure 3. The tested applications are measuring their own coverage and the service queries these data on-demand. As the communication is usually initiated before the start and after the end of the test cases, this means no regular communication overhead in the system during the test case executions.

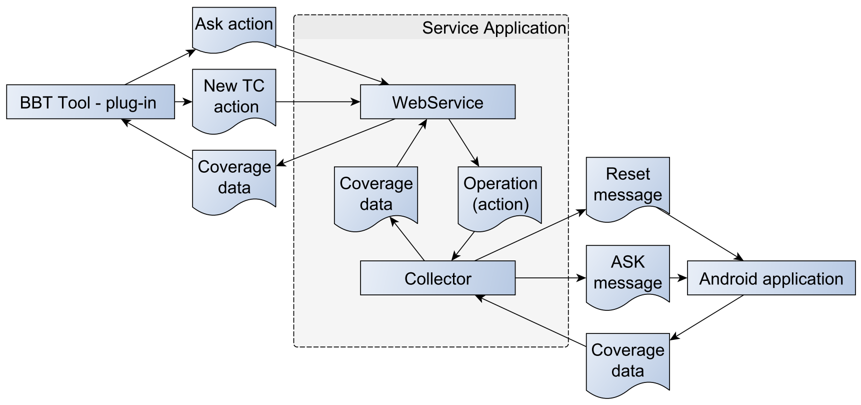


Fig. 3. Service Layer

Messages are accepted from and sent to the external coverage measurement tools. The communication uses JSON [21] objects (type-value pairs) over the TCP/IP protocol. Implemented messages are:

NEWTC The testing tool sends this message to the service to sign that there is a new test case to be executed and asks it to perform the required actions.
ASK The testing tool sends this message to query the actual coverage information.
COVERAGE DATA The service sends this message to the testing tool in response to the **ASK** message. The message contains coverage information.

Internally, the service also uses JSON objects to communicate with the instrumented applications. Implemented messages are:

reset The service sends this message to the application to reset the stored coverage values.

ask The service sends this message to query the actual coverage information.
coverage data The application sends this message to the service in response to the **ask** message. The message contains coverage information.

Installation To measure coverage on the Android system, two things need to be installed: the particular application we want to test and the common service application that collects coverage information from any instrumented application and provides a communication interface for querying the data from the device.

The service application needs to be installed on a device only once; this single entity can handle the communication of all tested applications.

The instrumented version of each application that is going to be measured must be installed on the Android device. The original version of such an application (if there was one) must be removed before the instrumented version can be installed. It is necessary because Android idetifies the applications by their special android-name and package, and our instrtumentation process does not change these attributes of the applications; it only inserts the appropriate instructions into the code. Our toolchain uses the `adb` tool (can be found in Android Development Kit) to remove and install packages.

3.2 Execution

During test execution, the Android device executes the program under test and the service application simultaneously. The program under test counts its own coverage information and sends this information when the service layer application asks for it. The coverage information can be queried from this service layer application through network connection. We implemented a simple query interface in Java for manual testing and a plugin for the RT-Executor [4] (a black-box test automation tool we used in this project) for automated testing.

In our pilot project, we used two possible modes of test execution: manual and automatized. Either mode is used, the service layer application must be started prior to the beginning of the execution of the test cases. It is done automatically by the instrumented applications if the service is not running already.

In the case of automated testing, the RT-Executor reads the test case scripts and executes the test cases. The client side of the measurement framework is contained in a plug-in of the automation tool, and this plug-in must be controlled from the test case itself. Thus, the test case scripts must be prepared in order to measure the code coverage of the executed applications.

The plug-in can indicate the beginning and the end of the particular test cases to the service, so the service can distinguish the test cases and can separate the collected information. In order to measure the test case coverages individually, one instruction must be inserted in the beginning of the test script to reset the coverage values and one instruction must be inserted in the end instructing the plug-in to collect and store coverage information belonging to the test case.

During test execution the following steps are taken:

– Start the program under test.

– The start of the program triggers the start of the measurement service if necessary. Then the program under test connects to the service and registers itself by its unique name given to it in instrumetnation process.
– The test automation system starts a test case. The test case forces the automation system plug-in to send a **NEWTC** message to the service. The service sends the **reset** message to the program under test. The PUT resets the coverage array in its collector class. The service returns the actual time to the plug-in.
– The test automation system performs the test steps. The PUT collects the coverage data.
– The test case ends. The automation tool plug-in sends the **ASK** signal to the service. The service sends the **ask** signal to the PUT. The PUT sends back the coverage data to the service. The service sends back the coverage data and the actual time to the automation tool plug-in.
– The plug-in calculates the necessary information from the coverage data and stores it in the local files. The stored data are: execution time, trace length, coverage value, lists of covered and not covered methods. Another plug-in decides if the test case was passed or failed and stores this information in other local files.

These steps are repeated during the whole test suite execution. At the end, the coverage information of all the executed test cases are stored in local files and are ready to be processed by different stages of the testing methodology.

3.3 Processing the Data

As we mentioned above, the client side of the coverage measurement system is realized as a plug-in of the RT-Executor tool.

The plug-in is controlled from the test cases. It indicates the beginning and the end of a test cases to the service layer application. The service replies to these signals by sending the valuable data back. When the measurement client indicates the start of a test case (by sending the **NEWTC** message to the service), the service replies with the current time which is stored by the client. At the end of a test case (when the **ASK** signal is sent by the client), the service replies with the current time and the collected coverage information of the methods.

When the coverage data is received, the measurement client computes the execution time, trace length (the number of method calls), and the list of covered and not covered methods' IDs. Then, the client stores these data in a *result* file for further use. The client makes other files, the *trace* files, separately for each test case. Such a trace file stores the identifiers of the methods covered during the execution of the test case.

As an alternative client, we implemented a simple standalone java application that is able to connect to the measurement service (and this way it replaces the RT-Executor plug-in). This client is able to visualize the code coverage information online, and is useful during the manual testing activities (e.g. during exploratory tests).

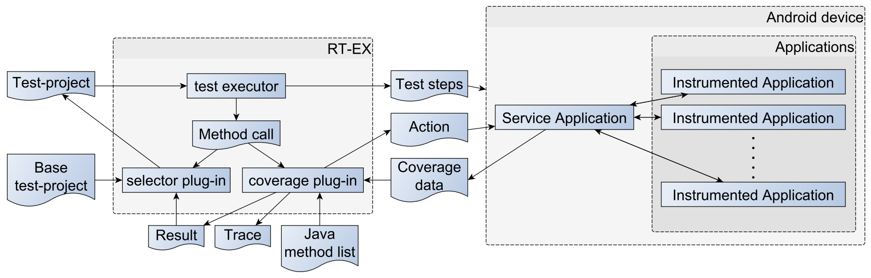


Fig. 4. Test execution framework with coverage measurement

3.4 Applications on the Measurement Framework Results

The code coverage and other information collected during the test execution can be used in various ways. In the pilot project, we implemented some of the possible applications. These implementations process the data files locally stored by the client plug-in.

Test Case Selection and Prioritization Test case selection is a process that defines a subset of a test suite based on some properties of the test cases. Test case prioritization is a process that sorts the test suite elements according to their properties [22]. A prioritized list of test cases can be cut at some points resulting in a kind of selection.

Code coverage data can be used for test case selection and prioritization. We implemented some selection and prioritization algorithms as a plug-in of the RT-Executor, which utilizes the code coverage information collected by the measurement framework:

– A change-based selection algorithm was implemented that used the list of changed methods and the code coverage information to select the test cases that covered some of the changed methods. Executing the selected test cases can only reduce the time required for regression test execution while the failure detection capability of the suite is not reduced.
– We implemented two well-known coverage-based prioritization algorithms: one that prefers test cases covering more methods; and another that aims at higher overall method coverage with less test cases.
– We also implemented a simple prioritization that used the trace length of the test cases. It can prioritize the tests either in the descending or the ascending order of the length of their traces.

Not Covered Code Not covered code plays an important role in program verification. There are two possible reasons for a code part not being covered by any test case executions. The test suite can simply omit its test case, in which case we have to define some new test cases executing the missed code. It can also happen that the not covered code cannot be executed by any test cases, which means that it is a dead code. In the latter case, the code can be dropped from the codebase.

In our pilot implementation, automatic test case generation is not implemented. We simply calculate the lists of methods covered and not covered during the tests. These lists can be used by the testers and the developers to examine the methods in question and generate new test cases to cover the methods, or to simply eliminate the methods from the code.

Traceability Calculation Traceability links between different software development artifacts play a very important role in the change management processes. For example, traceability information can be used to estimate the required resources to perform a specific change or to select the test cases related to the change of the specification. Relationship exists between different types of development artifacts. Some of them can simply be recorded when the artifact is created, some of them must be determined later.

We implemented a very simple traceability calculator that computes the correlation between the requirements and the methods, based on the pre-defined relationships between the requirements and the test cases and between the test cases and the methods (code coverage). If a requirement-method pair is assigned with high correlation, we can assume that the required functionality is implemented in the method. This information can be used to asses the number of methods to be changed if the particular requirement changes.

4 Usage and Evaluation

In this section, we present and evaluate some use cases to demonstrate the usability of the measurement toolchain.

4.1 Additional Test Case Generation

In the pilot project our target embedded hardware was an Android-based Set-Top-Box. We had this device with different pre-installed applications and test cases for some of these apps. A media-settings application was selected for testing our methodology and implementation. After executing the tests of this application with coverage measurement, we found that the pre-defined tests covered only 54% of the methods. We examined the methods and defined new test cases. Although the source code of this applications was not available, based on the not covered method names and the GUI, we were able to define new test cases that raised the number of covered methods to 69%. This is still less than the required 100% method level coverage, but shows that the feedback on code coverage can be used to improve the quality of the test suite.

4.2 Traceability Calculation

In the pilot project a simple implementation that is able to determine the correlation between the code segments and the requirements was made. We did not conduct detailed experimentation in this topic, but we did test the tool. Instead of the requirements, we defined 12 functionalities performed by three media applications (players) on our target Set-Top-Box device. Then, we assigned these functionalities to 15 complex black-box test cases of the media applications and executed the test cases with coverage measurement. The traceability tool computed correlations between the 12 functionalities and 608 methods, and was able to separate the methods relevant in implementing a functionality from the not relevant methods.

5 Conclusions and Future Work

In this paper, we presented a methodology for method level code coverage measurement on Android-based embedded systems. Although there were more solutions allowing the measure of the code coverage of Android applications on the developers' computers, no common methods were known to us that performed coverage measurement on the devices. We also reported the implementation of this methodology on a digital Set-Top-Box running Android. The coverage measurement was integrated in the test automation process of this device allowing the use of the collected coverage data in different applications like test case selection and prioritization of the automated tests, or additional test case generation.

There are many improvement possibilities of this work. Regarding the implementation of code coverage measurement on Android devices, we wish to examine if the granularity of tracing could be fined to sub-method level (e.g. to basic block or instruction levels) without significantly affecting the runtime behaviour of the applications. This would allow us to extract instruction and branch level coverages that would result in more reliable tests. We are also thinking of improving the instrumentation in order to build dynamic call trees for further use. The current implementation (simple coverage measurement) does not need to deal with timing, threads and exception handling, both of which are necessary for building the more detailed call trees. It would also be interesting to help the integration of this coverage measurement in commonly used continuous integration and test execution tools.

We are also examining the utilization possibilities of the resulting coverage data. For example, traceability information between code and the visible graphical elements could be established, and this information might help to partially automate collecting data for usability tests and to establish usability models. The implemented code coverage measurement and the testing process that utilizes this information are a good base for measuring the effect of using coverage measurement data on the efficiency and reliability of testing. We are planning to conduct researches in these topics.

Acknowledgement

This work was done in the Cross-border ICT Research Network (CIRENE) project (project number is `HUSRB1002/214/044`) supported by the **Hungary-Serbia IPA Cross-border Co-operation Programme**, co-financed by the European Union.

References

1. Google: Android homepage. `https://www.android.com/` (June 2013)
2. Kukolj, S., Marinković, V., Popović, M., Bognár, Sz.: Selection and prioritization of test cases by combining white-box and black-box testing methods. In: Proceedings of the 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC 2013). (2013)
3. Vlad Roubtsov: EMMA: a free java code coverage tool. `http://emma.sourceforge.net/` (June 2013)
4. RT-RK Institute: RT-Executor. `http://bbt.rt-rk.com/software/rt-executor/` (May 2013)
5. Beszédes, Á., Gergely, T., Papp, I., Marinković, V., Zlokolica, V.: Survey on testing embedded systems. Technical report, Department of Software Engineering, University of Szeged and Faculty of Technical Sciences, University of Novi Sad (2012)
6. Gotlieb, A., Petit, M.: Path-oriented random testing. In: Proceedings of the 1st international workshop on Random testing. RT '06, New York, NY, USA, ACM (2006) 28–35
7. Costa, J.C., Devadas, S., Monteiro, J.C.: Observability analysis of embedded software for coverage-directed validation. In: In Proceedings of the International Conference on Computer Aided Design. (2000) 27–32
8. Biswas, S., Mall, R., Satpathy, M., Sukumaran, S.: A model-based regression test selection approach for embedded applications. SIGSOFT Softw. Eng. Notes **34**(4) (July 2009) 1–9
9. Hazelwood, K., Klauser, A.: A dynamic binary instrumentation engine for the arm architecture. In: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems. CASES '06, New York, NY, USA, ACM (2006) 261–270
10. Marek, L., Zheng, Y., Ansaloni, D., Sarimbekov, A., Binder, W., Tůma, P., Qi, Z.: Java bytecode instrumentation made easy: The disl framework for dynamic program analysis. In Jhala, R., Igarashi, A., eds.: Programming Languages and Systems. Volume 7705 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 256–263
11. Chawla, A., Orso, A.: A generic instrumentation framework for collecting dynamic information. In: Online Proceedings of the ISSTA Workshop on Empirical Research in Software Testing (WERST 2004), Boston, MA, USA (july 2004)
12. Seesing, A., Orso, A.: InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse. In: Proceedings of the eclipse Technology eXchange (eTX) Workshop at OOPSLA 2005, San Diego, CA, USA (october 2005) 49–53
13. Slife, D., Chesney, M.: jCello. `http://jcello.sourceforge.net/` (June 2013)

14. Apache Commons: BCEL homepage.
`http://commons.apache.org/proper/commons-bcel/` (June 2013)
15. Chiba, Shigeru: Javassist homepage.
`http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/` (May 2013)
16. RT-RK Institute: Homepage.
`http://rt-rk.com/corporate-profile/` (May 2013)
17. Google: apktool homepage.
`https://code.google.com/p/android-apktool/` (May 2013)
18. Google: dex2jar.
`https://code.google.com/p/dex2jar/` (May 2013)
19. Google Android Developers: Building and running an android application.
`http://developer.android.com/tools/building/index.html` (May 2013)
20. Bornstein, D.: Presentation of Dalvik VM internals (2008)
21. Developers: JSON.
`http://www.json.org/` (June 2013)
22. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability **22**(2) (2012) 67–120

The Role of Dependency Propagation in the Accumulation of Technical Debt for Software Implementations

Johannes Holvitie, Mikko-Jussi Laakso, Teemu Rajala, Erkki Kaila, and Ville Leppänen

TUCS - Turku Centre for Computer Science, Turku, Finland
&
University of Turku, Department of Information Technology, Turku, Finland
{jjholv, milaak, temira, ertaka, ville.leppanen}@utu.fi

Abstract. Technical debt management requires means to identify, track, and resolve technical debt in the various software project artifacts. There are several approaches for identifying technical debt from the software implementation but they all have their shortcomings in maintaining this information. This paper presents a case study that explores the role of dependency propagation in the accumulation of technical debt for a software implementation. A clear relation between the two is identified in addition to some differentiating characteristics. We conclude that formalization of this relation can lead to solutions for the maintenance problem. As such, we use this case study to improve the propagation method implemented in our DebtFlag tool.

Keywords: technical debt, technical debt propagation modeling, software implementation assessment, refactoring

1 Introduction

Technical debt is a metaphor that describes how various trade-offs in design decisions affect the future development of the software project. Trade-offs are made between development driving aspects - for example meeting a delivery date by relaxing some quality requirements - and they incur the project's technical debt while providing the organization with a short-term gain. Similarly to its financial counterpart, technical debt - for example through reuse in software implementations - accumulates interest over a principal until it has been paid back in full. Inability to manage the projects technical debt results to increased interest payments in the form of additional resources being consumed when implementing new requirements and ultimately to exceeding development resources and premature ending of the project. [1]

Technical debt management is a software development component and an actively researched area of software engineering [2]. It is interested in providing projects with means to identify, track, and payback technical debt in order to provide similar control to technical debt as there exists for other project components. There are various software project artifacts, such as process, testing, architecture, implementation, and documentation, that are prone to the aforementioned decisions and thus to hosting technical debt. As these fields differ from each other to a large degree, techniques for managing technical debt are separate for each of them.

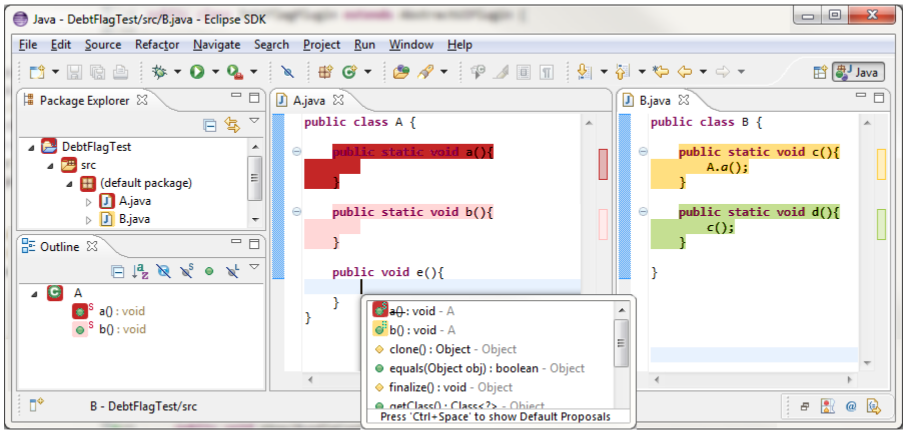
For the software implementation artifact, we can divide the technical debt identification techniques into automated [3] and manual approaches [4]. What is problematic is that the information produced by either of these approaches is only applicable to the assessed implementation version: automated approaches can produce results for all implementation versions, but they only highlight modules that are in violation when compared against a static model, leaving out information regarding module relations and links to previous implementation versions. Manual approaches on the other hand do provide some information regarding the history of a certain technical debt occurrence, but update frequencies to this information make these approaches only capable for tracking and managing technical debt on higher levels. These observations have lead us to conclude that if the relation between software implementation updates and increases in technical debt could be made explicit, we could extend the applicability of technical debt information, produced for a certain implementation version, to future versions. This would greatly increase the efficiency of technical debt information production for software implementations.

In this paper we present a case study that explores the aforementioned opportunity. Basing onto related research we make an assumption that dependency propagation is largely responsible for the accumulation of technical debt in the software implementation and that by better understanding this relationship we can increase the efficiency of technical debt information production and maintenance for this area. We focus on exploring this relationship by deriving two objectives for this case study: to identify technical debt and its structure in the studied system as well as to establish the role of dependency propagation in the formation of this structure.

The presented study is part of a research into establishing if a tool-assisted approach can be introduced for software projects in order to efficiently identify, track, and resolve technical debt in developed implementations. The results of this case study will be used to further develop the DebtFlag-tool [5] (see Figure 1) and its propagation model for technical debt. The DebtFlag-tool is a plug-in for the Eclipse IDE and it implements the DebtFlag-mechanism described in [5]. The tool is used to identify technical debt instances from the implementation and to merge them into entities allowing management at both the implementation and project levels.

2 Technical Debt

The term technical debt was first introduced by Ward Cunningham in his technical report to OOPSLA'92 [1]. Complementary definitions have been provided, in



```
public class A {

    public static void a(){

    }

    public static void b(){

    }

    public void e(){

    }
}
```

```
public class B {

    public static void c(){
        A.a();
    }

    public static void d(){
        c();
    }

}
```

```
~~a()~~ : void - A
b() : void - A
clone() : Object - Object
equals(Object obj) : boolean - Object
finalize() : void - Object
getClass() : Class<?> - Object
Press 'Ctrl+Space' to show Default Proposals
```

Fig. 1. *DebtFlag* code highlighting and content-assist cues in the Eclipse IDE [5]

amongst others, in the works of Brown et al. [6] and Seaman et al. [7]. A general consensus between these definitions is that technical debt bases on a principal on top of which some interest is paid. The principal corresponds to the size and amount of unfinished tasks that emerge as design decisions make trade-offs between development driving aspects. Principal is paid back by correctly finishing these tasks. Interest is increased by making more solutions depend onto areas where there are unfinished tasks. When creating these solutions, if additional work is required due to nonoptimality of these areas, this constitutes as paying interest. Seaman et al. formalize this further by defining interest as an occurrence probability coupled with a value [4]. The occurrence probability takes into account that not all technical debt affects the project: for example if a part of the software implementation is never re-used, the probability of this part hindering further implementation updates is zero.

Management of technical debt can be either implicit - like in many agile software practices, where reviews are made during and in between iterations to ensure that the sub-products meet the organizations definition of done - or explicit - like employing a variation of the Technical Debt Management Framework [4], [8]. In either case, the success of technical debt management is largely, if not solely, dependent onto the availability of technical debt information [7].

2.1 Technical Debt in Software Implementations

Following the definition of technical debt (see Section 2) we can see that for software implementations the unfinished tasks are components that, in their current state, are unable to fulfill their requirements. The size of these tasks corresponds to how difficult it is to finish each component and together they form the principal of the software implementation's technical debt. Similarly, we can see how the interest of technical debt forms in software implementations: dependency onto unfinished components indicates that the dependent may have had to accommodate this in some manner. This accommodation accounts as increased interest for the depended upon component's principal and if the amount of work required to implement the dependent is increased then this corresponds to paying interest.

To clarify, in the previous paragraph, a software implementation component refers to an entity that is defined by the used programming paradigm and technique and is capable of forming dependencies. The target system of this case study is implemented using the Java programming language. Here, like in many object oriented languages, direct references and inheritances create dependencies to public interfaces formed out of variables and methods [9].

In order to maintain the technical debt information produced either by means of automatic or manual identification, there needs to exist a model explaining how technical debt propagates in the software implementation. A theory on the propagation of technical debt in ecosystems by McGregor et al. [10] acknowledges some of the issues relating to this, which will be discussed in the next section. Additionally, certain implementation technique and paradigm specific characteristics need to be taken into account when identifying possible propagation routes for technical debt. Especially interfaces which can hide partitions of technical debt or decouple dependents from refactorizations.

Software implementation technical debt is paid back through refactoring the software product. Fowler et al. [11] define refactoring as "*changes made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*". In the following, we use this definition to identify which software components where affected by technical debt.

2.2 Related Work on Debt Propagation

Previously referenced work by McGregor et al. [10] is an important motivator for our research. In this, they hypothesize that technical debt has the ability to aggregate within elements of the software implementation and provide two concurrent mechanisms for it. In the first one "*technical debt for a newly created asset is the sum of the technical debt incurred by the decisions during development of the asset and some amount based on the quality of the assets integrated into its implementation*". In respect of this, they note that technical debt may diminish as a result of increased implementation layer nesting. The second mechanism providing another possibility in that "*the technical debt of an asset is not directly incurred by integrating an asset in object code form, but there is an indirect effect on the user of the asset*". For a software implementation this can mean for example that the implementation of a new element does not necessarily increase the technical debt quota but deficiencies in the documentation still result into more consumed resources.

Research is scarce in relating technical debt accumulation with the mechanics of software dependency propagation. Thus, we refer to research on software evolution and change impact analysis to gain insight into dependency propagation and its characteristics. Avellis discusses the implementation of a change impact function in [12] and notes that for domain-specific areas the information encoded into the domain models can be used to parameterize the change propagation rules while monitoring the ripple-effect of a change requires deep knowledge about the modification's implications. It is also concluded that the use of more specialized information in the definition of the propagation paths, results into a more specific and accurate impact set.

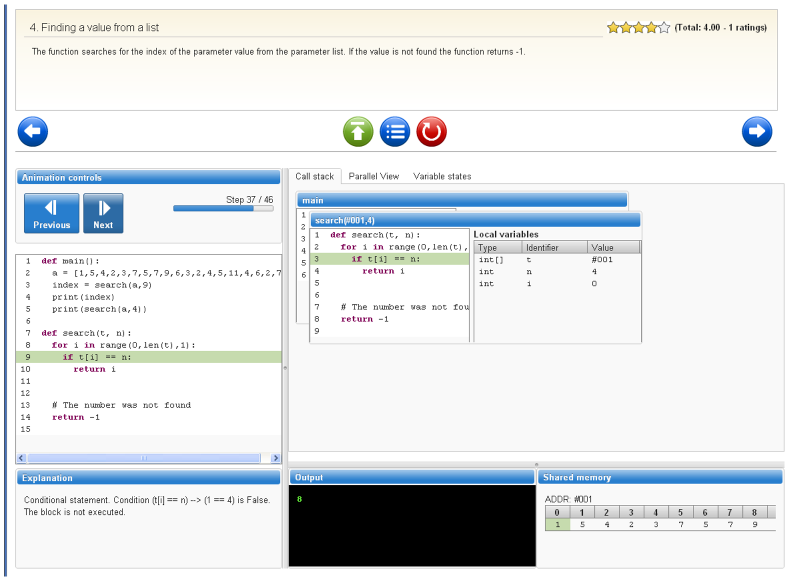
In Bianchi et al. [13] the authors note that the number of outgoing dependencies from a component is related to the number of paths through which the effects of a change may propagate. Robillard [14] presents an algorithm for providing an interest ranking for directly dependent change candidates. The ranking of elements is based onto *specificity* and *reinforcement*, where the former rules that *structural neighbors that have few structural dependencies are more likely to be interesting because their relation to an element of interest is more unique* and the latter that *structural neighbors that are part of a cluster that contains many elements already in the set of interest are more likely to be interesting because . . . [they] probably share some structural property that associates them to the code of interest.*

3 ViLLE

The system on which we will conduct this case study is called ViLLE (see Figure 2). It is a collaborative education platform that is being developed and researched at the University of Turku [15,16]. The system specializes in enabling the creation and to being host to various exercises with education enhancing features such as rich visualizations and immediate feedback [17, 18]. To date, the system has foregone 8 years of development, comprises circa 150k physical lines of code, serves over 1.5M immediate feedbacks annually, has circa 300 registered teachers and 6500 students, is being employed in over 20 countries and in its current state of robustness, trialled to become the selected system for providing electric matriculation examinations for the Finnish education ministry.

During its eight years of development ViLLE has gone over several smaller and two larger revamps. The first major revamp unified the platform into a single Java Applet and introduced automatically assessable exercises. Conversion to a Java Applet allowed the system to be run from the TRAKLA server which made the system accessible through the Internet and enabled its integration into distance teaching. The second major revamp enhanced this further: in order to reduce requirements to the end user to a bare minimum the system was converted into a SaaS (Software as a Service) by way of utilizing the Vaadin framework [19]. [16]

As the SaaS conversion made the system available to a larger audience, the research and development team simultaneously wanted to serve a broader spectrum of education subjects through extending the set of available exercise types. The old legacy exercise system was found to be too rigid for this purpose and it was decided that this part of the system was to be refactored. The authors have taken part in this process and it has also been the focus of a thesis [20].



The function searches for the index of the parameter value from the parameter list. If the value is not found the function returns -1.

```
def main():
  a = [1,5,4,2,3,7,5,7,9,6,3,2,4,5,11,4,6,2,7
  index = search(a,9)
  print(index)
  print(search(a,4))

def search(t, n):
  for i in range(0,len(t),1):
    if t[i] == n:
      return i


  # The number was not found
  return -1
```

```
def search(t, n):
  for i in range(0,len(t),
    if t[i] == n:
      return i


  # The number was not fou
  return -1
```

Local variables

Type	Identifier	Value
int[]	t	#001
int	n	4
int	i	0

Conditional statement. Condition (t[i] == n) --> (1 == 4) is False. The block is not executed.

```
8
```

ADDR: #001

0	1	2	3	4	5	6	7	8
1	5	4	2	3	7	5	7	9

Fig. 2. The interactive student view of a ViLLE-coding-exercise [16]

The thesis has documented the entire refactorization project that is used in the case study presented in this paper.

4 Case Study

4.1 Research Problem

The case study examines the role of dependency propagation in the accumulation of technical debt for a software implementation. Approaching the research problem we have divided it into two objectives. The first objective is to identify and produce a structured documentation for technical debt in the target implementation. The second objective is to understand the role of dependency propagation in the formation of this structure.

Fulfilling the first objective requires that we are first able to distinguish between modifications made to develop the implementation and modifications made to refactor the implementation. After identifying modifications that belong to the latter - and count as paying of technical debt, further information is required to identify relations between the modifications. Revealing these relations allows us to arrange the individual modifications to form a structure that indicates how technical debt has accumulated in the implementation.

The second objective is to understand the role of dependency propagation in the formation of this structure. Dependencies are formed between elements of the implementation. These elements and the rules for dependency formation between them are defined by the programming paradigm as well as the programming language. As each identified modification operates on a set of implementation elements we can utilize the dependency formation rules to identify all elements that are dependent onto this set. Comparing the revealed dependencies to the connections in the technical debt accumulation structure is used in this case study to examine the role of dependency propagation in the accumulation of technical debt for the software implementation.

4.2 Case Selection

The case study is conducted on the results of the ViLLE refactorization project (see Section 3). This case selection is made to expand on earlier research described in [20]. We consult this research to establish what parts of the system were targeted in the refactorization, what are the tools and practices used for the refactorization, what are the motivations as well as the requirements for the refactorization and finally access to the version control system which is queried for information regarding the conduction of this refactorization.

The ViLLE system is a web-application that is implemented using the Vaadin web-application framework. The used development language is Java. At the time of the refactoring the running configuration of the ViLLE system was comprised out of 122k physical lines of code organized into a hierarchy of 26 Java packages encompassing a total of 460 Java classes.

The thesis [20] documented that the motivation for the refactorization was that the development team perceived the exercise system to be too rigid to accommodate efficient development in the future. Further analysis in [20] pinpointed this problem to four Java classes. These core system classes were responsible for the execution, modification, storing and retrieving, as well as modeling of interactive exercises in ViLLE. For each of these [20] documented a set of problems as well as a set of reparative actions, which were used as the starting point for the refactorization.

The refactorization used a well defined refactorization process - adapted from *The Rhythm of Refactoring* by Fowler et al. [11] and *The Legacy Code Change Algorithm* by Feathers et al. [21] - as well as a library of best practices - compiled from the *Design Patterns* by Martin et al. [22], *Refactorizations* by Fowler et al. [11], and *Dependency-Breaking-Techniques* by Feathers et al. [21] - to implement the suggested reparations. [20]

An example of the refactorization process and the resulting refactorizations is the abstraction of the exercise execution class via decoupling it from exercise type specific information. Applying this five step process first called for identifying change points. In this case, all references to specific exercises. The next step of finding test points consisted from identifying change routes and understanding how the system could be shielded from unintended changes by way of constraining these routes with tests. The third step called for breaking dependencies in order to get the tests in place. The end result of this was a set of unit tests adhering to the JUnit framework. The last, fifth, step was to make changes and refactor. An example of a singular refactoring here was the removal of specific exercise information from the constructor of the exercise executor.

The *Replace Constructor with Factory Method* [11] refactorization was used to relocate a switch case from the constructor to a separate method, making the use of the constructor possible without first modifying its implementation.

Development towards refactoring the system was done independent from the main development line. In practice, a separate version control branch was used. Further, due to the nature of this project the branch in question could only contain commits that corresponded to meeting the requirements of the refactorization. From the point-of-view of this case study, we interpreted this as all modifications observable from this version control branch as constituting to paying of technical debt and thus relevant data to the study in question.

4.3 Data Collection and Analysis

The data provider in this case study was the version control system for ViLLE's implementation. We constrained this data set to the branch in the version control system identified in Section 4.2. As this constriction limited the data set to only containing modifications that corresponded to refactorizations, we proceeded to building the structured representation for technical debt accumulation for this implementation (see Section 4.1).

In Section 2.1 we discussed how technical debt manifests in software implementations: reliance onto technically incomplete objects may call for adaptation in dependents. Successfully paying off technical debt for the implementation implies that individual refactorizations are able to nullify the adaptations as well as to remove the root cause. In this case the root cause was confined within four Java classes (Section 4.2). Each of these classes were responsible for implementing an independent and distinctive functionality in the system. As the structured representation for technical debt accumulation was to reflect how inabilities in implementing system functionalities had affected the system, four root nodes were chosen. Each root node consisted out of a set of modifications corresponding to all refactorizations made to repair the functionality of - and to remove the root cause from - one aforementioned class.

Having identified the root nodes and their modification sets, we continued to study the remaining modifications. Links between modifications were determined as cause-effect-relations: a link existed between modifications if successful completion of the cause-one required a successful completion of the effect-one. The chronological order - of cause-modifications taking place before effect-modifications - was ensured by observing that the effect-ones could only exist in revisions that were the same or superseded that of the cause-ones'. The two step process was repeated until all modifications were associated with the structure for technical debt accumulation.

To facilitate the fulfillment of the second objective, we related information about the propagation of dependencies to the structured representation for technical debt accumulation. As the system in question is implemented using the Java language the object-oriented paradigm as well as the Java technology can be consulted for information about the propagation of dependencies in the implementation. Exploiting this, for each modification, the set of implementation elements dependent onto its target implementation element were identified. This set was then queried to find out if it contained elements being targets of modifications linked with the modification used to spawn the set. The results were then associated with the structure for technical debt accumulation in order to clearly indicate the role of dependency propagation in its formation. Analysis of the resulting structures is done to fulfill the second objective.

5 Results

This case study was conducted in order to examine the role of dependency propagation in the accumulation of technical debt for a software implementation. The research problem was divided into two objectives: determining and providing a structured representation for the accumulation of technical debt in the implementation as well as relating dependency propagation information to this structure in order to understand its role in the formation of the structure. The data used in the analysis of this case study is an interval of version control revisions encompassing an entire refactorization undertaking for a software system.

Analyzing revisions of the ViLLE system, we found that the refactorization consisted out of 140 individual modifications or refactorizations which affected a total of 71 Java classes. Amongst these were the four Java classes encompassing what [20] had identified as the root cause. Observing which modifications realized the removal of the root cause in these four classes lead to the formation of four modification sets that served as the root nodes for our structured representation for technical debt accumulation. According to the case study design (see Section 4.3) an iterative process of identifying cause-effect-relations lead to populating the four substructures with rest of the modifications. Identification of cause-effect-relations for all modifications also indicated that a modification could only be associated with a single substructure.

The resulting technical debt accumulation structure was then associated with information regarding the propagation of dependencies. This corresponded to identifying the target elements for all modifications, identifying sets of elements that were dependent on the target elements, searching for possible relations between element dependencies and modification links and finally relating this information to the technical debt accumulation structure. The resulting structure is presented in the following as four Technical Debt Propagation Trees (TDPT).

5.1 Technical Debt Propagation Trees

Figures 3 through 6 depict the resulting Technical Debt Propagation Trees when modifications made to Java classes responsible for execution, modification, storing and retrieving, as well as data modeling the exercises in the ViLLE system are respectively used as root nodes for the analysis presented in Section 4.

The same visual aids apply for all presented TDPTs. Nodes represent modifications (Section 5.2 discusses a common modification and its implementation). Arrows indicate cause-effect-relations between modifications. The root node - the used modification set - is modeled as a triangle. If a dependency exists between the target elements of modifications of a cause-effect-relationship, then the node for the effect-modification is modeled as an ellipse. If not, the node is modeled as a rectangle. If the modification type is addition of new implementation elements, then the node is colored green (light shade). Else, if the modification type is removal of implementation elements, then the node is colored red (dark shade). Finally, the number inside each node is the sum of dependencies to target elements of the modifications.

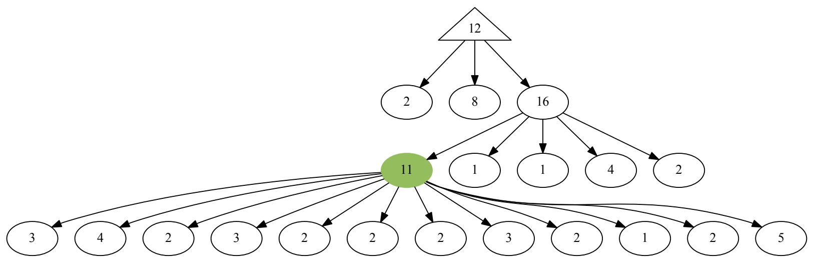


Fig. 3. The Technical Debt Propagation Tree having the modifications made to the exercise execution implementation as its root node

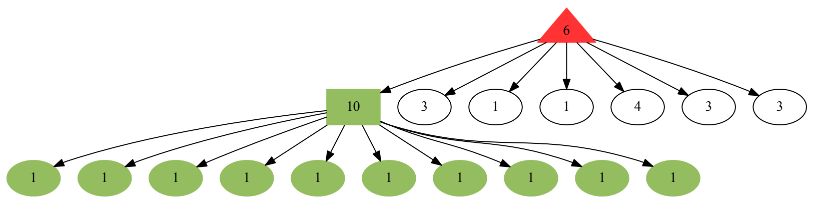


Fig. 4. The Technical Debt Propagation Tree having the modifications made to the exercise storing and retrieval implementation as its root node

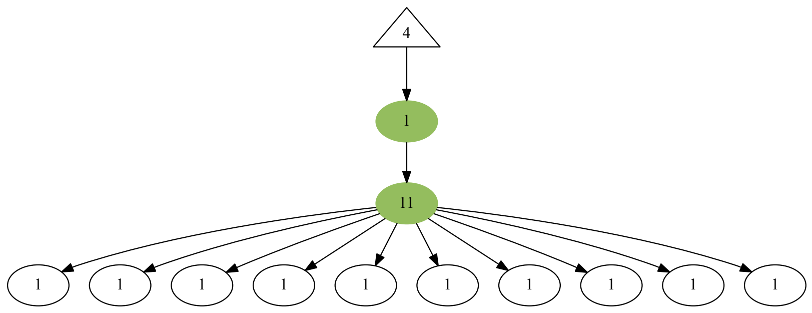


Fig. 5. The Technical Debt Propagation Tree having the modifications made to the exercise modification implementation as its root node

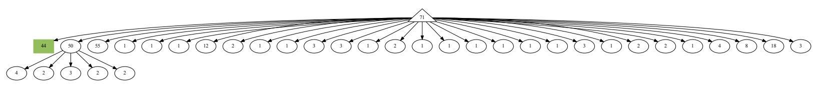


Fig. 6. The Technical Debt Propagation Tree having the modifications made to the exercise data modeling implementation as its root node

5.2 Analysis of the Technical Debt Propagation Trees

In analyzing the TDPTs (see Figures 3 through 6) we have observed the following. First, modifications to implementation elements with a large number of incoming dependencies seem to invoke an increased number of further modifications. This however is not consistent as the number of incoming dependencies deviates from the number of invoked modifications, which is evident for example by observing the TDPT for data modeling (Figure 6): at the second tier of the tree where the number of incoming dependencies greatly exceeds the number of invoked modifications in five occasions - more than ten incoming dependencies while number of invoked modifications is five for one case and zero for others.

Second, examining the cause-effect-relations forming the edges of our TDPTs, in all but two cases there exists a dependency between underlying implementation elements for an observed cause-effect-relationship between modifications. Close examination of the first non-dependency case (between the root and a second tier node in TDPT for storing and retrieving in Figure 4) revealed that refactoring here separated functionality from the original area and the newly formed element hierarchy was thus made completely independent from its original element leading to non-dependency between modifications' target elements. In the second non-dependency case (between the root and a second tier node in TDPT for data modeling in Figure 6) similar motivation could be observed. Exercise type declarations were separated here from the generic exercise data model and placed into their own containing class. Hence, it seemed that in almost all cases dependency propagation was the evident cause for technical debt accumulation.

Third, examining the depths of the TDPTs we can observe the following. In the case of TDPTs for storing and retrieval as well as data modeling the tree depth is three, while for TDPTs for execution and modification the tree depth is four (see Figures 4, 6, 3, and 5 respectively). Further, for all leaf modifications the number of dependencies incoming to their target elements is rather low - under ten. Except for the few cases mentioned in the previous paragraph.

Fourth, an observation made from the evident differences in the tree structures. In the case studied system modifying a component that is responsible for providing a data model in the implementation (see TDPT for data modeling in Figure 6) seemed to invoke a series of modifications that could be described as shallow but wide. While, modifications responsible for implementing specific features of the system seemed to invoke a series of modifications that were more narrow and focused than the former (see TDPTs for execution, modifying, and storing and retrieval in Figures 3, 5, and 4 respectively). This seems to indicate that for refactored-to-be elements of the implementation, their role in the system could be used to postulate the course of the refactorization undertaking in this part of the system.

6 Conclusions and Validity

This case study has examined the role of dependency propagation in the accumulation of technical debt for a software implementation. The research problem was divided into two objectives and an approach was derived to fulfill them. Applying this approach to case study data resulted into the successful formation of four Technical Debt Propagation Trees. Analysis of these trees lead to the following observations.

The number of incoming dependencies correlates with the number of propagation paths for technical debt with the exception of a small number of events which does not adhere to this. Secondly, dependency propagation can be seen to drive the accumulation of technical debt in this software implementation, except for two cases where this can not be observed. Thirdly, examination of the TDPTs supports what has been earlier hypothesized about technical debt diminishing due to dependency propagation. Finally, as an additional observation, the role of a system component could be used to explain how technical debt had propagated in the system.

Concluding onto these observations: it is evident that dependency propagation plays a significant role in the accumulation of technical debt for a software implementation. The propagation of dependencies, which are possible to explicitly indicate for a software implementation, can be used to predict the size and distribution of technical debt. If differences between the propagation paths for technical debt and implementation dependencies can be taken into account, this information could be automatically generated for indicated sources of technical debt providing a mean to forecast the state of the software implementation as well as a tool to estimating the size and urgency of reparative efforts. Finally, these conclusions indicate that the approach derived for this case study is viable for examining the role of dependency propagation in the accumulation of technical debt.

6.1 Validity

As this case study examines a unique phenomenon in a specific context, applicability of the results requires that certain threats to validity are discussed. A matter affecting the validity of the case study's construct is the definition used for an acceptable modification. In this case study all observed modifications were accepted as paying off technical debt. This acceptance criteria was based firstly on to the provided definition of a refactorization in Section 2.1 as well as the limitation of the data set discussed in Section 4. It can be argued that the used acceptance criteria was too loose, and the resulting TDPTs were over populated. However, it can also be counterclaimed that as the case study specifically targeted a refactorization project with the foremost intend of not altering the system's behavior this bias will be small in size.

The results of this case study required that we identified a causal relation between the propagation of dependencies and the accumulation of technical debt. Matters distorting the identification affect the case study's internal validity. Section 4 explained the processes used for determining both cause-effect-relations between modifications as well as the propagation of dependencies between implementation elements. Here, the latter is determined based on static rules and confirmed in the ability for the program to function. However, determining of cause-effect-relations was based on the researchers' ability to distinguish if two modifications shared a context. While most information in the contexts - for example close chronological ordering and linkage between affected implementation areas - lead to a strong conclusion, the possibility of making a wrong decision can not be excluded. However, issue-free and successful association of all modifications indicates that uncertainty played a small role in this step.

7 Future Work

Research following this case study will build on the conclusions in Section 6. Firstly, we intend to employ the approach derived and used in this case study for additional data sets. We expect this to provide more details on the intrinsics of technical debt accumulation in software implementations in addition to further examining the role of dependency propagation in this process. We are especially interested in identifying if certain dependency types accumulate technical debt differently, if the role of system components can be used to further explain the size and distribution of technical debt, and if other mechanisms can be established for the non-dependency driven accumulation of technical debt.

Further, the results of this and following analyses will be used to build and assess the propagation model used by the DebtFlag-tool [5]. As the tool relies on the ability to maintain technical debt notions through this model, explicitly presenting the differences in the propagation paths of technical debt and dependencies between implementation elements will allow for further enhancements. As such, our ongoing research is focused on assessing and evaluating possible models to identify viable solutions. A strong candidate is the link structure algorithm *PageRank* by Page et al. [23]. Initial analyses with the data provided in this paper has yielded promising results especially in accommodating the diminishment characteristic of technical debt.

References

1. Cunningham, W.: The WyCash portfolio management system. In: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (OOPSLA). Volume 18. (1992) 29–30
2. Ozkaya, I., Kruchten, P., Nord, R.L., Brown, N.: Managing technical debt in software development: report on the 2nd international workshop on managing technical debt, held at icse 2011. SIGSOFT Softw. Eng. Notes **36**(5) (September 2011) 33–35
3. Izurieta, C., Vetrò, A., Zazworka, N., Cai, Y., Seaman, C., Shull, F.: Organizing the technical debt landscape. In: Managing Technical Debt (MTD), 2012 Third International Workshop on, IEEE (2012) 23–26
4. Seaman, C., Guo, Y.: Measuring and monitoring technical debt. Advances in Computers **82** (2011) 25–46
5. Holvitie, J., Leppänen, V.: DebtFlag: Technical Debt Management with a Development Environment Integrated Tool. In: Managing Technical Debt (MTD), 2013 Fourth International Workshop on, IEEE (2013)
6. Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., et al.: Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP workshop on Future of software engineering research, ACM (2010) 47–52
7. Seaman, C., Guo, Y., Izurieta, C., Cai, Y., Zazworka, N., Shull, F., Vetrò, A.: Using technical debt data in decision making: Potential decision approaches. In: Managing Technical Debt (MTD), 2012 Third International Workshop on, IEEE (2012) 45–48
8. Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Da Silva, F., Santos, A., Siebra, C.: Tracking technical debtan exploratory case study. In: Software Maintenance (ICSM), 2011 27th IEEE International Conference on, IEEE (2011) 528–531
9. Barowski, L.A., Cross, J., et al.: Extraction and use of class dependency information for java. In: Reverse Engineering, 2002. Proceedings. Ninth Working Conference on, IEEE (2002) 309–315
10. McGregor, J., Monteith, J., Zhang, J.: Technical debt aggregation in ecosystems. In: Managing Technical Debt (MTD), 2012 Third International Workshop on, IEEE (2012) 27–30

11. Fowler, M., Beck, K.: Refactoring: improving the design of existing code. Addison-Wesley Professional (1999)
12. Avellis, G.: Case support for software evolution: A dependency approach to control the change process. In: Computer-Aided Software Engineering, 1992. Proceedings., Fifth International Workshop on, IEEE (1992) 62–73
13. Bianchi, A., Caivano, D., Lanubile, F., Visaggio, G.: Evaluating software degradation through entropy. In: Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International, IEEE (2001) 210–219
14. Robillard, M.P.: Topology analysis of software dependencies. ACM Transactions on Software Engineering and Methodology (TOSEM) **17**(4) (2008) 18
15. Rajala, T., Laakso, M.J., Kaila, E., Salakoski, T.: Ville: a language-independent program visualization tool. In: Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88, Australian Computer Society, Inc. (2007) 151–159
16. Rajala, T., Kaila, E., Laakso, M.J.: ViLLE. http://ville.cs.utu.fi/ (2013)
17. Laakso, M.J.: Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations. PhD thesis, Turku Centre for Computer Science (2010)
18. Kaila, E., Rajala, T., Laakso, M., Salakoski, T.: Important features in program visualization. In: Appeared in ICEE: An International Conference on Engineering Education. (2011) 21–26
19. Grönroos, M., et al.: Book of Vaadin. Vaadin Limited (2011)
20. Holvitie, J.: Code level agility and future development of software products. Master's thesis, Department of Information Technology, University of Turku (2012)
21. Feathers, M.: Working effectively with legacy code. Prentice Hall (2004)
22. Martin, R.C.: Agile software development: principles, patterns, and practices. Prentice Hall PTR (2003)
23. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: bringing order to the web. (1999)

A Regression Test Selection Technique for Magic Systems

Gábor Novák, Csaba Nagy, Rudolf Ferenc

Department of Software Engineering
University of Szeged, Hungary
{novakg|ncsaba|ferenc}@inf.u-szeged.hu

Abstract. Regression testing is an important step to make sure that after committing a change to our software we do not make unwanted changes to other, untouched features. For larger and faster evolving software, however, executing all the test cases of a regression test can easily become a tremendous process which takes too much time to thoroughly test each changes separately. In our paper, we present a method to support regression testing with impact analysis based test selection. As a result, we can show a limited set of test cases that must be re-executed after a change, to test the changed part of the code and its related code elements. Our technique is implemented for a special 4th generation language, the Magic xpa development environment. The technique was implemented in cooperation with our industrial partner, SZEGED Software Inc, who has been developing Magic applications for more than a decade.

1 Introduction

While the evolution of programming languages make the development process faster and faster, the new generations of programming languages allow us to create larger programs in less time than before. This faster program development needs new, faster testing methods, which allow us to test the new code as fast as the code has been created. A cost-effective practice for testing one part of the code or a function is to create a test case to check the functionality of the concerned code. Later if we want to check again the functionality somewhere because of some source code changes, we just rerun that test case, which verifies the concerned part, and we compare the result of the test with the expected, good result. There are well defined standards about testing too which define main steps of testing software and its documentation, e.g. standards created in 1998 [3,1,4]. In the standard, they define the software testing as a process centered around the goal of finding defects in system. The IEEE standard [4] defines test case as the basic element of software testing: "the test case is a set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement." There are a lot of different IDEs which allow the developer to trace the run of a test case with a dynamic monitoring or tracking mechanisms. That is, the IDE tracks all the events performed by the user during the test run of the test case, and with this event log we know every little detail about the test run.

While the source code is continuously changing, developers need to re-run these test cases time to time to make sure the changed source code still gives the appropriate result on the test cases. To make sure that we did not introduce a new bug, or unwanted changes of existing features, one must always re-run all the existing test cases. However, in most cases one change does not affect all the test cases, but only those one, which are closely related to the changed source code elements. These changes does not require to re-run the whole test case set, but it is hard to find the smallest set of related test cases. Test selection [4] tackles solutions to this problem. Its main purpose is to select the affected test cases. This problem usually emerges in regression test selection [4]. In regression tests the goal is to chose those test cases which are the most affected by modifications in the source code, and re-run these cases. In most cases this works statically, and requires only the source code of the system. Furthermore we are able to measure the percentage of the covered code (covered by the current test case), this method is the test coverage [4].

There are a number of tools implemented for regression tests or measuring test coverage for popular 3rd generation programming languages (E.g.: Squish [10], Appperfect [8]), but in the context of 4th generation languages (4GLs), there are no universal solutions for these problems.

Impact analysis [7,13] can be a great support in regression test selection. The aim of impact analysis is to get the transitive affected part of a certain change in the source code. A simple change (e.g. renaming a method) might impact a larger part of the code, even though we wanted to change only small part of it. This impacted code set is called the impact set. In this paper we represent a conceptual background of a regression test selection mechanism, that is based on impact analysis, and an implementation of this mechanism, to solve presented problems in Magic xpa, as a special 4 GL programming language.

Test selection and especially regression test selection is useful when the test rerun demands a lot of resources. We can save time, and money with the fast test rerun. Our presented technique is based on calculating the test coverage (for every single test case) in the same time, while executing the regression test and then, next time, use the previously collected coverage data for selecting a subset of test cases that must be rerun to test the modified code.

In Section 2 we introduce the reader to the world of the Magic programming language, and the main structures of a Magic application. Then, in Section 3 we show the conceptual background of the presented test selection technique, after that we present some results achieved in Section 4 and finally we conclude our paper in Section 6.

2 Structure of Magic Applications

Higher level, so called 4th generation programming languages (4GLs) do not use source code in the traditional way. The code is not directly written by the developer, but it is generated automatically by an application development environment, and in most cases the generated underlying code is hidden from the developer. The developer develops the application at a higher, conceptual level, and he uses ready solutions of the development environment. Usually the programmer define the expected mechanism in a well defined UI, and the development environment generates the code which represent that mechanism. In case of Magic the generated program runs on the Magic Runtime Engine. In the development environment, Magic xpa, we define the program, the expected mechanisms, and when we want to run it, the Magic xpa starts the generated code on the engine.

The "source code" of a Magic application is a set of XML files. In fact this source code of a Magic application is an XML snapshot of the actual state of the application loaded into the development environment. This XML format is appropriate for static analysis as it describes the whole application. Magic is based on a special way of development with special coding elements. The two most important elements for us in Magic are the Tasks (or Programs), and the Data Objects. A Task is constructed of Logic Units and one or more other Tasks, so-called Sub Tasks. None of them are necessary, but the most important function in a Task, is the Task call mechanism. There are 2 different kinds of Tasks in Magic, the Batch Task and the Online Task. Every single Task is able to call another Task, whether it is Sub Task or an other, independent Program (a top level Task). A Data Object is a persistent object, which allows the program to use different data sources with the same mechanism, hide the specification of the data source (The data source can be database, XML file ...).

Analysing a program is usually conducted by static analysis, since analysing the program during runtime can be really difficult task. The static analysis methods is usually based on Abstract Semantic Graph (ASG) [15]. The XML based source of a Magic application is not significantly different from this ASG format, this is due to the XML structure, which is also a graph representation of the system in a way.

Magix xpa allows us to create dynamic runtime traces, while the program under test is running on the application engine. This trace file is a text file where every line is an entry about one event in the program. The level of the recorded events can be also specified, but finer granularities have greater influence on the execution time of the program under test. The log level is enough for our test selection purposes, but it might not be enough to get information for deeper analysis. For example Magic does not log the entered text in a text field or a text area. There is one very important log entry in the trace file for the regression test selection, and this record is a task start entry.

```
<253693576794033600> 13:50:58.406 [Action ] - >>Starts load Batch
Task - ’Main Program (Calculator)’ in Query mode (Task Instance: 1)
```

3 Test Selection in Magic

In the system developing process creating the test cases is one of the most important steps. The developer needs to rethink once more the actual part of the program what she or he wants to test with that test case, because if the program will change, this test case ensures, that function works as it was specified before the modification. This idea leads us to the Test Driven Development (TDD) model [6]. In TDD, the main concept is to create the test case before writing the actual program, or function. This method encourages the developer to rethink the mechanism and create a better test case before the actual program or function is completed.

Regardless of the development model that we currently use in our development methods, over the time we need to re-run the completed test cases more often then creating new ones. From that point the regression test selection programs have many advantages. A regression test selection tool is responsible to choose the most affected test cases with a source code change, but this task is not easy at all. After the tool chooses the test cases, just rerun the chosen ones, or maybe the tool do this for us. If we have tons of test cases, we can save a lot of resource with this technique. We need to keep in mind, the best way is to always rerun every test case, but there are many situations when we are not able to rerun every single test case after every change of the code. We create a new method to chose the most concerned test cases for a Magic application, which of course depend on a source code change. Before we describe the technique we need to know which inputs are necessary for the test selection. Figure 1. shows the required inputs and the generated outputs of the test selection methods.

There are 3 essential inputs for the test selection. In the application which implement this method we use a test manager [5] tool and an SVN repository to give the input to the test selection tool. These are the required inputs for the method:

- **Test cases:** First of all, the program needs to identify every test cases, so the first important information about a test case is its unique identifier. In our case this information came from the test manager tool [5]. This test manager tool was created to manage test cases for Magic applications. The other important information, which came from this test manager tool is the trace files for every test case. This trace file, as mentioned earlier, is the test case run log. It contains every important information, about the user, and the program activities (For example: Task calls, mouse click events). Every test case has at least one trace file in the test manager, and we get these trace files for the test selection.
- **The source of the Magic project:** The tool needs to know the structure of the Magic program. That is, the tool performs a static analysis on the code to collect all the necessary information. In Magic the source code of a program is a set of XML files. In the implementation of the program we ask for an SVN repository, which contains a Magic program's source code. Basically the tool looking for 2 important information from the source code.

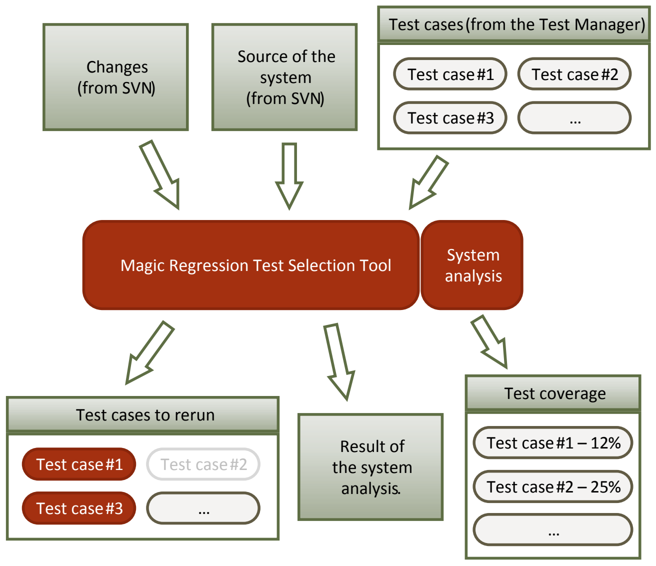


Fig. 1. The inputs and the outputs of the regression test selection method.

The first is the data object descriptions from the `DataSource.xml` file. This information important because because the task descriptions contains data object references and before processing tasks, the tool needs to know the data objects of the program. The second essential information is the task hierarchy. The tool creates a directed graph from the tasks and data objects which is required for the test selection. The task descriptions is in the `Progs.xml` file, this file contains the basic informations about the root tasks, the detailed information and the sub task descriptions are in the `Prog_{i}.xml` files. Where `i` is between 1 and the highest task id in the `Progs.xml`.

- **Source code changes:** Another required input for the algorithm is the changed source code parts. The method needs to know which task was modified, because if there were no changes in the code, there is nothing to rerun. In the implementation we give this information to the tool in the guise of two SVN revision number. The program in the preprocessing part analyzes the `diff` informations between this two revisions, and collects the modified task set. Because of the task's hierarchical structure the tool always chooses just the directly affected tasks into the modified task set. The algorithm needs a task list, which contains the modified tasks, nothing else. If we create a new preprocessor to analyze Magic application version controlled in GIT or Mercurial we can still use the method.

The operations in the algorithm:

- **Processing test cases:** As mentioned earlier every test case has a unique identifier, and at least one trace file. In this step we process the trace files for the test cases. We process every trace file's every line and look for special entries. The entries, what we look for are the "task successfully started" entries. There are two kinds of tasks in Magic, the batch and the online tasks. Every kind of task has a special unique entry in the trace file, which contains the unique identifier of the task. We collect every called task identifier from every trace file for every test case. When we finish this step we have a set of affected task identifiers for every test case. This is the pseudo code of this process:

```
processTestCases ():
  tasks = $\emptyset$
  For every test case
    For every test case's trace file
      For every line in the trace file
        If it's a successful task call entry
          tasks = tasks $\cup$ new called task
        End if
      End for
    End for
  End for
End
```

– **The system source code processing:** In this step we process the descriptor files of data objects and tasks from the SVN repository and compute a special graph from those informations. First of all, we need to process the data object information in the `DataSource.xml` file. The tasks refer to the data object with its unique identifier (a number), so we need to collect the available data object names and identifier numbers before we analyze the tasks. After this step we have this set:

$$D = \{d_1, \ldots, d_m\}, \quad d_i = \text{The i. data object} \tag{1}$$

In the next step we need to process every `Prg_{i}.xml` file, and extract the necessary informations about the tasks. While we process the files we create a special tree or if there are more than one `Prg_{i}.xml` files then we create a forest hierarchy from the tasks. Every `Prg_{i}.xml` represent a tree, where the root vertex is the main task, and the parent-child connections represented with the task-subtask connections. When we finish this step we have the following $F(V_f, E_f)$ forest:

$$F = \{f_1, \ldots, f_n\}, f_i = \text{A tree, represent the Prg\_\{i\}.xml } (1 \le i \le n) \tag{2}$$

After this step, we create a graph from the data object set and the forest of the tasks. The final $G(V_g, E_g)$ graph created with the following rules:

$$\begin{gathered} \forall v \in V_g, \quad v \in D \vee v \in V_f(f_i)(\exists f_i \in F) \\ \forall e(v_i, v_j) \in E_g, \quad (v_{i,j} \in V_f(f_k) \wedge e \in E_f(f_k)) \quad \vee \\ (v_i \in V_f(f_k) \wedge v_j \in D \wedge v_i \text{ task use } v_j \text{ data object }) \text{ , where } f_k \in F \end{gathered} \tag{3}$$

The algorithm which creates the G graph is the following:

```
processSystemSource ():
  dataObjects = ∅
  tasks = ∅
  V = E = ∅
  For all data object entry in dataSoruces.xml
    dataObjects = dataObjects ∪ current data object
    V = V ∪ current data object
  End for
  For all Prg_{i}.xml
    mainTask = Getting the main task
    getSubTask(mainTask, tasks, V, E)
  End for
  G = (V, E)
End

getSubTasks (task, tasks, V, E):
  tasks = tasks ∪ task
  V = V ∪ task
  For all data object, which used by the task
    //Currently in the graph, because we processed once before
    E = E ∪ (task - current data object)
  End for
  subTasks = All task's sub task
  For every task in the subTasks
    E = E ∪ (task - current sub task)
    getSubTasks(current sub task, tasks, V, E)
  End for
End
```

– **Get the changed tasks list:** In the implementation of the algorithm, we get two SVN revision numbers, which represent the changes in the source code. When we compare the difference between these two revisions we know the exact location of the changes. In this method, we pay attention only the task changes. We can use the SVN `diff` command to get the change information between the two revision. When we get the `diff` log from the SVN, we look for the `Prg_{i}.xml` file changes. If we know one `Prg_{i}.xml` file changed, we need to narrow it down for one task. We need to process the `diff` and get only the modified task or tasks from that file. As we mentioned before, every `Prg_{i}.xml` file may contain sub-tasks, and if the changes affect only one sub-task we can not mark "changed" the main task. We mark "changed" only the affected task, because if we start a task in runtime, that does not mean the sub-task start directly after that. The result of this step is a task set. In this set every task's one or more line of code changed between the two revision. The following pseudo code describe this method:

```
processChanges ():
  tasks = $\emptyset$
  Every diff entry in svn
    If Prg_{i}.xml modified
      task = Get the task which modified by the diff
      tasks = tasks $\cup$ task
    End if
  End for
End
```

– **Test coverage definition:** The test coverage definition or calculation in this case is a percentage value for every test case. This percentage is derived from the number of affected tasks by the test case divided by the number of tasks in the application. This rate shows us what percentage covered with the test case, of the whole system.
– **The system's dependencies:** In this step we calculate a percentage for every data object in the Magic system. This percentage describes a ratio between the current data object and the related tasks of the system. In this calculation we specify for every data object the number of tasks, which use this data object.
– **Test cases to rerun:** This is the most important part of the algorithm. In this step, we give the information, which test case is necessary to rerun. It depends on the previously extracted information: the changed tasks set and the $G(V_g, E_g)$ graph. The method uses 2 different steps to get this information. In the first step the algorithm computes the directly affected test cases, which means this test cases are the most important ones. This is a fast step, because we already have every necessary information. For each test case, the algorithm intersects the current test case's set of tasks and the set of changed tasks. If the intersection is not empty, then the affected test cases are those which belong to tasks in the intersection. These test cases are directly affected, so we select them for re-run. Figure 2. shows this step.

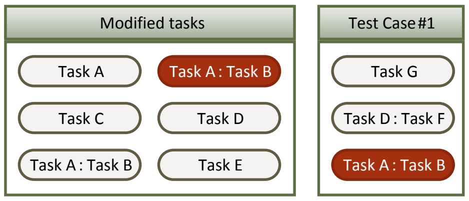


Fig. 2. The directly affected test cases has a not empty intersection with the changed tasks.

During the second step the method calculates the transitive dependencies, create the impact set of the affected codes, and chose the indirectly affected test cases. For this, we need to create a new graph from the existing $G(V_g, E_g)$

one. We mark "affected" every task in the graph, which are in the changed tasks set. After that we mark "affected" every vertex in the marked graph vertex's Markov blanket [2]. When we finished this step we do the first step with this extended affected tasks set and if the algorithm finds new affected test cases, that means those test cases are indirectly affected by the source code change. With the Markov blanket we make sure to chose all necessary test cases, but with some restrictions we can reduce the size of the result set (the test cases which need to re-run).
We tested the following mechanisms:

Full Markov blanket.

Chose only the connected data objects, and the tasks which use those data objects.

Chose only the parent and child tasks.

Figure 3. and 4. show this step.

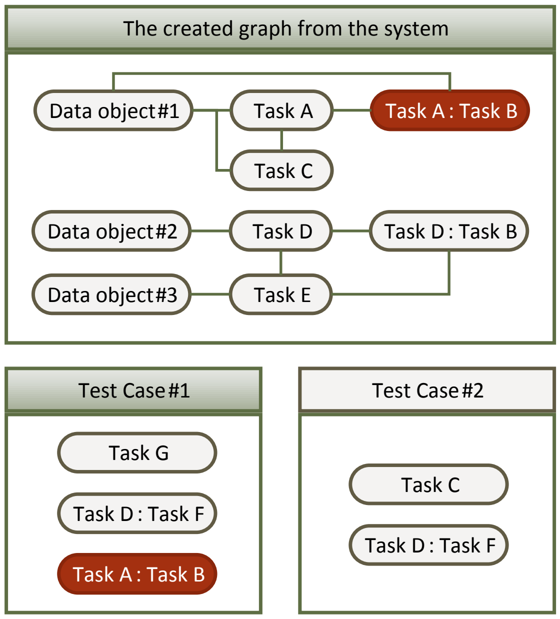


Fig. 3. The directly affected test cases immediately selected for rerun.

We can describe this method with the following pseudo code:

```
//At that point we have:
// - The G graph,
// - Every test case (testCases) and the set of the affected tasks
// - The modified tasks set from the svn diff (modifiedTasks)
```

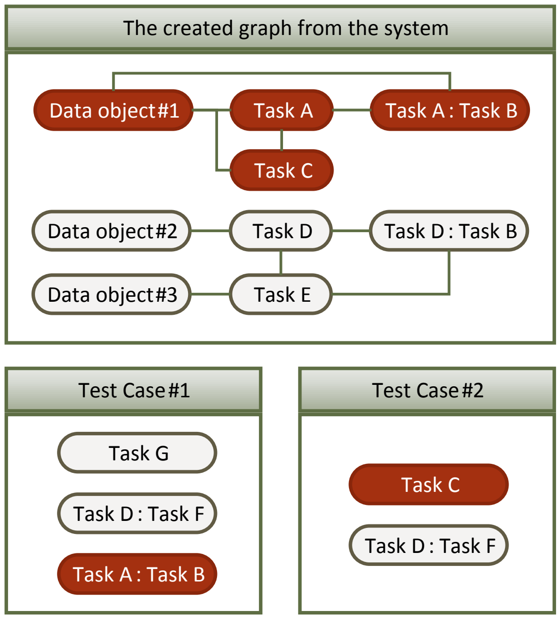


Fig. 4. When we expand the affected set with for example a Markov blanket we select inderectly affected test cases for rerun.

```
getRetesteredCases1():
  retesteredCases = $\emptyset$
  For every test case
    tasks = The affected tasks by the current test case
modifiedTasks
    retesteredCases = retesteredCases $\cup$ current test case
  End for
  changedTasks = modifiedTasks
  For every task in the modifiedTasks
    changedTasks = changedTasks $\cup$ The Markov blanket of
the current modified task in the G graph.
  End for
  For every test case
    tasks = The affected tasks by the current test case
changedTasks
    retesteredCases = retesteredCases $\cup$ current test case
  End for
End
```

As we see, we can migrate the two steps into one. In the implementation of the algorithm we migrate these two steps into one, but the result is the same, we just show separately to show the different reason's of the two step. The migrated algorithm is the following:

```
getRetesteredCases2():
  retesteredCases = ∅
  changedTasks = modifiedTasks
  For every task in the modifiedTasks
    changedTasks = changedTasks ∪ The Markov blanket of
the current modified task in the G graph.
  End for
  For every test case
    tasks = The affected tasks by the current test case
changedTasks
    retesteredCases = retesteredCases ∪ current test case
  End for
End
```

4 Evaluation

The test selection method and the implementation were mostly tested on self-constructed small projects and on Demo projects provided with the Magic xpa. Our test project has more than 300 tasks, 20 data objects and in the Magic Test Manager system we created 329 real test cases for it. Despite of the size of the project the result is very promising. With few changes, which mean 4-5 modified tasks, the tool successfully reduced the 329 test cases to 5. Although, changing 4-5 tasks is not a huge modification in the code. The result was 5 selected test cases, when the tool marked just the directly affected test cases. When we enabled the impact analysis, the result set was getting bigger and bigger, depending on the impact analysis method. We got the biggest (100 selected test cases) set if we used the Markov blanket to expand the impact set. Of course, that means that we chose the test cases, which affected in the slightest degree too, but the result was still better than the whole set of the test cases. If we chose the connected data objects, and the tasks which use those data objects, for the impact set, the result was 50 test cases. With this tool we can reduce the size of the set of "must rerun" test cases to 15-30%. If we need a lot of resources or time to rerun the test cases that means we save the 70-85% of the resource or the time. Note, the result is not that attractive if the change modify a lot of task, but in that case the only failsafe practice is to rerun all the test cases, because a lot of changes usually means changing the structure of the system.

5 Related Work

With the help of test selection the developers and testers are able to save a lot of time and resources, and if rerunning the test cases requires a lot of time or resource than it can be measured in money of course. This is the main reason why this topic is so popular. There are plenty of books and papers in this field, but especially for Magic language (and in general about the 4 GL languages) there are only a few available in the scope of regression testing. A comprehensive, mostly practical study on methods about software testing is collected in the The Art of Software Testing [14] book. In this book authors collect and describe the most useful methods, in the aspect of resource requirement. Emelie Engström's et al. [9] systematically collect information about the empirical evaluations of regression test selection techniques. They collect 28 different methods for regression test selection evaluations. They conclude that they can not make a final decision, about which method is the best, because every method depends on a different aspects and factors. Mary Jean Harrold et al. [11] wrote a paper about a new regression test selection technique for the Java programming language. This method is usable when the Java program's source code is not completely finished or the program uses 3rd party libraries. This RETEST technique successfully reduced the size of the test environment. This technique is similar to our method, but RETEST works on Java, a 3rd generation programming language. Gregg Rothermel et al. [16] describe a method, which uses the system's Control Flow Graph (CFG [17]) for regression test selection. The benefit of the CFG usage is that they are able to use this method in every 3 GL. This method has a lot of benefits, but it works on 3GLs.

In case of 4 GLs existing regression test selection techniques cannot be applied explicitly, they must be adopted to the specific language, because each language has a different structure. This different structure is the result of the different purpose of the 4GLs. A book, entitled Testing SAP Solutions [12] collect and describe the available methods for testing an SAP ABAP application, which are also popular 4GLs. For Magic applications there is no currently available solution for the regression test selection.

6 Conclusions and Future Work

The method which we presented in this paper, successfully reduces the resource and time requirement of the regression test selection method of Magic development processes. This method has clearly benefits in regression testing. Besides, the developers can easily use the system analysis results and the test case coverage informations to make better test cases or filter out the most useless test cases, or refactoring the system. With those extra functions this test selection method and the program implementing it, are very powerful tools, which can be used in the whole software development phase. Because the tool does not need a lot of input for the analysis and the test selection, and most of them automatically extracted from other tools (Magic Test Manager, SVN), this is a good choice for almost every time in a Magic application development. The prototype of the implementation is currently able to use only SVN to identify changes in the source code, but it is easy to expand it for other version control systems too. One drawback is that the developers need to frequently use the tool to have up-to-date coverage data for test cases. Also, when the change set is so large that it affects too many test cases the result might simply contain almost every test cases. In the future plans we want to test this method and the implementation on bigger Magic applications, and we want to generalize this method for more programming languages, not just Magic.

Acknowledgements

This research was supported by the Hungarian national grant GOP-1.1.1-11-2011-0039.

References

1. *Standard for Software Component Testing*. British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST).
2. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.
3. *IEEE Standard for Software Test Documentation IEEE Std 829-1998*. 1998.
4. *Standard glossary of terms used in Software Testing, ISTQB Glossary - version 2.1*. 'Glossary Working Party' International Software Testing Qualifications Board, 2010.
5. A layout independent gui test automation tool for applications developed in magic/unipaas. *In Proceedings of the 12th Symposium on Programming Languages and Software Tools*, 2011.
6. Kent Beck. *Test driven development: By example*. Addison-Wesley Professional, 2003.
7. Shawn A. Bohner. Software change impact analysis. *IEEE Computer Society Press*, 1996.
8. AppPerfect Corp. Appperfect software test tools. `http://www.appperfect.com/`.
9. Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14 – 30, 2010.
10. froglogic. Squish gui testing tool. `http://www.froglogic.com/squish/gui-testing/`.
11. Mary Jean Harrold, James A Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *ACM SIGPLAN Notices*, volume 36, pages 312–326. ACM, 2001.
12. Markus Helfen, Michael Lauer, and Hans Martin Trauthwein. *Testing SAP solutions*. Galileo Press, 2007.
13. MS Kilpinen. *The emergence of change at the systems engineering and software design interface: an investigation of impact analysis*. PhD thesis, 2008.
14. Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. Wiley, 2011.
15. Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases. *Software Maintenance, IEEE International Conference on*, 0:188–197, 2004.

16. Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM TRANSACTIONS ON SOFTWARE ENGINEERING AND METHODOLOGY*, 6:173–210, 1997.
17. O. Shivers. Control flow analysis in scheme. *SIGPLAN Not.*, 23(7):164–174, June 1988.

VOSD: A General-Purpose Virtual Observatory over Semantic Databases

Gergő Gombos, Tamás Matuszka, Balázs Pinczel, Gábor Rácz, and Attila Kiss

Eötvös Loránd University, Budapest, Hungary
{ggombos,tomintt,vic,gabee33,kiss}@inf.elte.hu

Abstract. E-Science relies heavily on manipulating massive amounts of data for research purposes. Researchers should be able to contribute their own data and methods, thus making their results accessible and reproducible by others worldwide. They need an environment which they can use anytime and anywhere to perform data-intensive computations. Virtual observatories serve this purpose. With the advance of the Semantic Web, more and more data is available in RDF databases. It is often desirable to have the ability to link local data sets to these public data sets. We present a prototype system, which satisfies the requirements of a virtual observatory over semantic databases, such as user roles, data import, query execution, visualization, exporting result, etc. The system has special features which facilitate working with semantic data: visual query editor, use of ontologies, knowledge inference, querying remote endpoints, linking remote data with local data, extracting data from web pages.

Keywords: Virtual Observatory, Semantic Web, e-Science, Data Sharing, Linked Data

1 Introduction

E-Science is based on the interconnection of enormous amounts of data collected from various scientific fields. These massive data sets can be used for conducting researches, during which it is often desirable that researchers can share their own data and methods, thus making the results of the research accessible and reproducible by anyone. The idea of virtual observatories coming from Jim Gray and Alex S. Szalay serves this purpose [1]. A system like this expands the possibilities of combining data coming from various different instruments. Virtual observatories can also be used to teach and demonstrate the basic research principles of various scientific fields (for example, astronomy or computer science). The researchers must have access to these constantly growing amounts of data, in order to be able to use them in various research projects. Another important requirement is to be able to publish the results. The Internet provides an excellent opportunity to satisfy the criteria mentioned above [1]. The primary motivation for creating virtual observatories is to facilitate making new discoveries, and to provide a solution for carrying out data-intensive computations remotely. To access remote data, web services can be used [2].

The basic principles of science have been extended with a fourth paradigm. A thousand years ago, experimental results and observations defined science. In the last few hundred years, it shifted towards a theoretical approach, focusing on creating and generalizing models. During the last few decades, simulating complex phenomena with computers were becoming more and more common. Nowadays, researchers have to deal with large amounts of data, usually coming from sensors, telescopes, particle accelerators, etc. The data is processed using software solutions, and the extracted knowledge is stored in databases. Analyzing or visualizing the results needs further software support [3, 4].

A possible way to manage the data available on the Internet is to use the Semantic Web [5]. The Semantic Web aims for creating a"web of data": a large distributed knowledge base, which contains the information of the World Wide Web in a format which is directly interpretable by computers. The goal of this web of linked data is to allow better, more sensible methods for information search, and knowledge inference. To achieve this, the Semantic Web provides a data model and its query language. The data model called the Resource Description Framework (RDF) [6] uses a simple conceptual description of the information: we represent our knowledge as statements in the form of subject-predicate-object (or entity-attribute-value). This way our data can be seen as a directed graph, where a statement is an edge labeled with the predicate, pointing from the subjects node to the objects node. The query language called SPARQL [7] formulates the queries as graph patterns, thus the query results can be calculated by matching the pattern against the data graph. Furthermore, there are numerous databases which contain theoretical and experimental results of various scientific experiments in the field of computer science, biology, chemistry, etc. There is a quite complex collection of these kinds of data maintained by the Linked Data Community [8]. This collection contains datasets and ontologies which are at least 1000 lines in length, and which contain links to each other.

In this paper, we present a prototype system, which fulfills the standard requirements of a virtual observatory, such as handling user roles, bulk loading data , answering queries, visualization, and storing results. In addition, we extended the system with special semantic technologies. We use the SPARQL language to formulate queries, aided by a visual SPARQL editor. Ontologies can be used to describe the hierarchy of complex conceptual systems, and to carry out knowledge inference. The system implements a tool, which helps its users to convert the data found on the web to the formats of the Semantic Web. We also provide a SPARQL endpoint to enable remote querying of the knowledge base. The query results can be exported to various common semantic data formats. We demonstrated the flexibility of the system by implementing two different database backends.

The structure of the paper is as follows. After the introductory Section 1, we present the high-level architecture of our virtual observatory in Section 2. Then, in Section 3, we describe the main functionality of the system. Section 4 writes about the functions supporting the collaboration of researchers. Then, we show some possible use cases of our system in Section 5, followed by the conclusion and our future plans in Section 6.

2 Architecture of the Virtual Observatory over Semantic Databases

The system is built using the Java EE platform. The user interface uses the Java Server Faces (JSF) technology, which is hosted using an Oracle WebLogic Application Server. Besides the JSF pages, the system is available via a REST webservice, to browse the models. This makes it possible to develop various applications even for mobile devices. Data storage can be realized with either of the two database backend solutions provided by us (Oracle, PostgreSQL). The Oracle database engine supports managing semantic models, and provides a Jena Adapter for these functions. Using the built-in semantic support, we can, for example, perform knowledge inference at the database level. We created a second database backend, which uses only the standard functionality of the relational databases. This can be used to connect to any standard relational database not natively supporting semantic technologies. The connection among the database backend and the other components is implemented using the JENA Framework. We tested this backend using the open-source PostgreSQL database. Figure 1 shows the main components of the system with the two different backend databases.

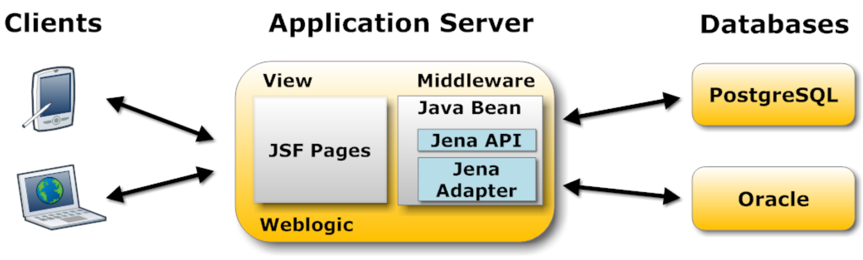


Fig. 1. The architecture of the Virtual Observatory over Semantic Databases

3 Functionality

3.1 Data Loading

There are two ways to load data into the system. One works by uploading a file containing the semantic data, the other requires a URL pointing to a resource on the Internet, which contains the data. There are various RDF serialization formats for RDF which can be used with the system, such as RDF/XML, N3, Turtle, and N-Triples. The most wide-spread is the RDF/XML, which represents the RDF graph as an XML document. This format is easier for computers to read, since there are numerous tools available for processing and transforming XML. The other formats store the data using a more human-readable serialization. The simplest one is the N-Triples [9], which is simply the enumeration of the RDF triples (the edges of the RDF graph) separated with a dot. The Turtle [10] serialization allows more structures to simplify the expressions. For example, we can use prefix abbreviations to eliminate long, repeating IRIs, thus reducing the file size significantly. Furthermore, we have the option to group triples sharing the same subject, without repeating the common subject for all triples. This works similarly, if both the subject and the predicates are the same, and only the objects vary. This, too, helps to reduce the file size. Literals in Turtle can have language tags, or data type information added to them. Notation 3 [11] (or N3) allows further simplifications to make the serialization of complex statements easier.

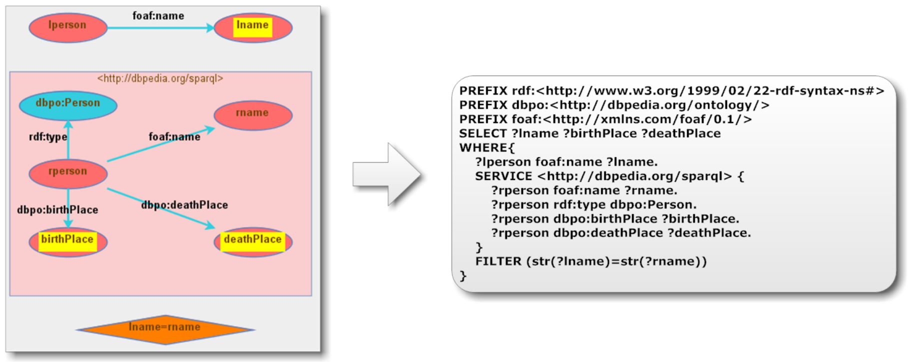
3.2 Querying and Saving Results

Another main function of the system is querying the already loaded data. The SPARQL [7] language is used to express queries over semantic data sets. The language is similar to the well-known SQL language. The SELECT clause defines a projection of the variables, the values for which we would like to see in the result set. The WHERE clause defines the criteria the data must satisfy in order to appear as a result. This is basically a graph pattern that has to match the data graph. The simplest queries contain only triples in the graph pattern. The FILTER clause lets us provide further filtering conditions for the nodes. For example, if we have numeric nodes, we can use arithmetic operators on them to restrict the values to a given range. If we have string nodes, we can filter for their values as well. IRIs, and string nodes can be filtered using regular expressions, too. By default, all edges in the graph pattern of the WHERE clause have to match the data. However, we have the option to define optional matching criteria with the OPTIONAL keyword. If parts of the graph pattern are optional, then we can have rows in the result set which satisfy only the non-optional parts, with null values for the variables appearing only in the optional parts. This is useful when some information is not given for all of our individuals. For example, if we have an address book with addresses for all contacts and phone numbers for some of them, we can ask the phone numbers in the optional part. Without the OPTIONAL keyword, we would only get the contacts with both an address and a phone number. The advantage of the Semantic Web is that we can link our data with knowledge from other sources. In queries, the SERVICE keyword allows querying remote data sets. The keyword requires a URL to a SPARQL endpoint, and a graph pattern that has to match the remote data. The most well-known data set is the DBpedia [12], which contains the knowledge of Wikipedia in semantic form. Data sets linked with DBpedia can be found in the LOD cloud [8].

Another useful feature of the semantic web is knowledge inference, which lets us extract new information based on what we already know. Computing inferred data may take long time, thats why our system offers two options regarding inference. One option is to run the query using only ground truth data (i.e. the data already available to us as facts), or we can enable inference meaning slower query execution. There are multiple ways to carry out inference. For example, we can use the relationship information given in ontologies, to generate new information. Another option is to use user-specified rules. A rule consists of a head (a new triple holding the new information) and a body (a condition that has to be satisfied in order for the rule to activate). The simplest example is the grandparent relationship (if x is parent of y, and y is parent of z, then x is grandparent of z). We can save the query results using the already mentioned formats: RDF/XML, N3, TURTLE, and also CSV.

3.3 Visual SPARQL Editor

With the spreading of the Semantic Web technologies, using SPARQL becomes more and more inevitable, since this declarative language is the standard tool to express queries over RDF data sets. VisualQuery is a visual query editor program, which allows us to build a SPARQL query using graphs and supplementary forms.



```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpo:<http://dbpedia.org/ontology/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
SELECT ?lname ?birthPlace ?deathPlace
WHERE{
   ?lperson foaf:name ?lname.
   SERVICE <http://dbpedia.org/sparql> {
      ?rperson foaf:name ?rname.
      ?rperson rdf:type dbpo:Person.
      ?rperson dbpo:birthPlace ?birthPlace.
      ?rperson dbpo:deathPlace ?deathPlace.
   }
   FILTER (str(?lname)=str(?rname))
}
```

Fig. 2. An example SPARQL query both in graphic and textual form which finds additional information on DBpedia about locally stored famous people

Graphic representation has various advantages. Firstly, using this approach, it is easier to see and understand the relationship of the individual elements, thus, the meaning of the query can clearly be seen as demonstrated in Figure 2 where the graphic and textual representation of the same query are shown. Secondly, we can quickly and easily modify the components and parameters defining the query. This way, we can improve or refine the query step-by-step. Thirdly, because the visual representation is language-independent, the co-operative work of researchers speaking different languages is supported. Another advantage of the program is that it performs various checks during editing, which helps preventing syntactical errors, for example:

– literal nodes can not have outgoing edges – they can not be subjects in a triple,
– only variables or IRI nodes can be edges – blank nodes and literals can not,
– variables in the head of a CONSTRUCT-type query must appear at least once in the WHERE clause.

What makes this solution different from similar programs – like iSparql [13] or LuposDate [14] – is the distinction of visual elements by type, and the built-in checks based on this distinction.

3.4 Visualizations

Visualizing semantic data helps us interpret them. We integrated into the system other, third-party visualizer tools. One of them is Cytoscape Web [**?**], which allows us to display the semantic graph of locally stored models using various built-in layouts, such as tree or circle. The application uses JavaScript, so rendering happens on the clients computer.

Another visualization tool integrated into the system is RelFinder [16], which searches connections among IRIs. To find connections, it runs SPARQL queries on an endpoint. The relations among the IRIs can be paths via common predicates. We can specify the depth of the search. The program uses ActionScript for the display that provides various tools to create animations.

3.5 Extracting Semantic Data from the Web

Nowadays, we can easily find all kinds of information using the web. There are numerous sites, which specialize in collecting and organizing knowledge about one specific topic. For example, we can find websites collecting information about hardware components, reviews about movies, historical weather data, recipe collection, etc. These websites usually operate using a database of their own, and the web pages displayed to us are generated based on the data from that database. However, the databases are usually not using semantic technologies, moreover, they are often not public, so the only way for us to access their data is to visit the web pages containing them. Fortunately, extracting the data from the web pages does not always require complex text processing and text mining, because the structure of the document can be used to extract the pieces of information that we are interested in. The structure is almost always consistent on all pages of a web site. For example, on a site collecting recipes, the structure can be the following: the name of the recipe is always the title of the document, and it is followed by some meta information (always in the same order), such as the name of the uploader, the difficulty and the required time to cook the dish. After this, we have an unordered list of the ingredients, and finally, there is an ordered list of the steps of preparing the meal. If we know this structure, we can use it to extract the mentioned information from all pages containing recipes on this site.

To help users in extracting data from sites like these, we created a tool that allows them to define the structure using one example page from a web site, and based on the structure, our virtual observatory is capable to extract the required information from all pages that use the same document structure. The tool comes in the form of a browser extension, which the user can download and install from the web front end of our virtual observatory. After installation, if the user views one example page of the website using his browser, he can select the parts of the website that contain information he would like to extract. The browser extension marks the selected parts during the process. If a structure is repeating within the document, we have the opportunity to extract all occurrences of the repeating structure. For example, if we have a hundred-row table, with each row containing information about one item, we do not have to mark all rows, only the first one. The repeating structures can be nested to arbitrary depths, i.e. we can have, for example, ordered lists within a table within a table. After the user finished marking the example document, the extension saves the structure information to a file. To do the actual extraction, the user has to visit the web front end of the virtual observatory, where he can upload the structure file, and the system will then extract the information from the specified sub-pages of the site, and load the extracted data into a standard semantic model.

4 Collaboration of Researchers

One of the most important purposes for virtual observatories is to collect information originating from various different sources, and to support their integration. Our system allows users to upload their own data and share it with others. We applied a multi-level permission system based on user groups. Every user can create groups, and invite other users to them. This way, research groups can be organized. Then, we have two possibilities to share the models containing our data. We can make the model publicly available to every other user, or we can give right to one or more groups to access our model. While the first possibility gives read-only access, in the latter case the group members can have write rights, too. In this case, they can load their own data into the model.

It is also possible to publish queries. This can be useful in several cases: if other researchers would like to use our data, we can help their work by providing example queries, which illustrate the inner structure and relationships of the data. We can formulate basic queries, which can be further refined or specialized later.

5 Use Cases

5.1 OCR Application

The first application is useful in the field of tourism. The main function of the program is to recognize text on street signs with OCR methods, based on pictures taken with mobile phones. Its purpose is to provide extra information about the famous people whose name can be found in the extracted texts. The extra information comes from various data sources converted to semantic format (Hungarian Electronic Library, various online encyclopedias [17]), joined with other public data sets (DBpedia, GeoNames). A user group created for this purpose allows the collaboration between the users. The group has access to the data sets described above. One member of the group was given the task to collect information about the famous people appearing in street names, and then upload them to a model. He then shared the model inside the group. Another member had the same task, but he had to use an online encyclopedia as the data source. He added his data to the shared model. Meanwhile, a third member worked on linking the data in the model to data available in DBPedia, using SPARQL queries. He stored the results in a new, local model, to make it faster to access. (His work was not influenced by the fact that in the meantime, new data has been added to the model.) He also published the queries and the new model to the group. The members of the group created a virtual model over the models mentioned. (A virtual model is not materialized, but it contains the union of the data found in other models, and it is supported by an index structure.) This step was important, because it allowed us to access the data as a single model. Then, using the REST API of our virtual observatory, we were able to run queries from a mobile application.

5.2 Use in Education

We use the virtual observatory during teaching the basic principles of the Semantic Web, within the Modern Databases course. The students of the course are added to a new group, and we share previously loaded models and queries with them. The models contain small data sets, so they could be viewed with the visualization tools, and the students could easily understand their structure. From week to week, they are introduced to the features of the SPARQL language, by solving typical tasks together. The new features can easily be demonstrated with the visual SPARQL editor, since the graphical representation speaks for itself. In some cases, the results of the exercises can be used in practical scenarios. For example, the family tree of a royal family can be created, if each student creates a model with the family tree of a selected king. During their work, they get to know the basic semantic serialization formats (RDF/XML, N3, etc.) and the results can be published to a common group.

6 Conclusion and Future Work

In the paper, we presented a prototype system, which fulfills the requirements of a virtual observatory, and helps the collaboration of researchers by letting them work using the same, shared data and queries. We used the data model of the Semantic Web, thus, the data sets in the virtual observatory can easily be linked to each other and to public data sets. We provided several features which can facilitate the use of the system, such as advanced data and query sharing, visual query building and editing, data visualization, and web data extraction. The system can run on top of any standard relational database system, but if the underlying database has some support for storing and handling semantic data (like Oracle databases), it can make use of those functions as well. We also presented real world use cases, where the existence of the system helped our work on other projects and in education. During further work, we would like to extend the system to be able to work using a Hadoop cluster as backend. In this solution, data storage and query execution would be distributed, thus the efficiency of the data-intensive computations would increase. Our other plans include enhanced visualization, such as the ability to plot geographic locations on a map, and to create charts and diagrams to help the better understanding of the data.

Acknowledgments. This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013). We are grateful to Zsófia Mészáros and Zoltán Vincellér for helpful discussion and comments.

References

1. Gray, J., Szalay., A.: The world-wide telescope. Communications of the ACM 45 11, 50–55 (2002)
2. Szalay, A. S., Budavári, T., Malik, T., Gray, J., Thakar, A. R.: Web services for the virtual observatory. In: Astronomical Telescopes and Instrumentation, pp. 124–132. International Society for Optics and Photonics (2002).
3. Brase, J., Blümel., I.: Information supply beyond text: non-textual information at the German National Library of Science and Technology (TIB) – challenges and planning. Interlending & Document Supply 38 2, 108–117 (2010)
4. Hey, AJG.: The fourth paradigm: data-intensive scientific discovery. In: Stewart Tansley, S., Tolle, K. M. (eds.) Microsoft Research, Redmond (2009)
5. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American 284 5, 28–37 (2001)
6. Lassila, O., Swick, R. R.: Resource Description Framework (RDF) Schema Specification, `http://www.w3.org/TR/rdf-schema`
7. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF, `http://www.w3.org/TR/rdf-sparql-query/`
8. Bizer, C., Jentzsch, A., Cyganiak, R.: State of the LOD Cloud, `http://wifo5-03.informatik.uni-mannheim.de/lodcloud/state/`
9. N-triples, `http://www.w3.org/2001/sw/RDFCore/ntriples/`
10. Turtle, `http://www.w3.org/TR/2012/WD-turtle-20120710/`
11. Notation3, `http://www.w3.org/TeamSubmission/n3/`
12. Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia – A crystallization point for the Web of Data. Web Semantics: Science, Services and Agents on the World Wide Web 7 3, 154–165 (2009)
13. iSparql, `http://oat.openlinksw.com/isparql/index.html`

14. Groppe, J., Groppe, S., Schleifer, A., Linnemann, V.: LuposDate: A semantic web database system. In: Proceedings of the 18th ACM conference on Information and knowledge management, pp. 2083–2084. ACM (2009)
15. Shannon, P., Markiel, A., Ozier, O., Baliga, N. S., Wang, J. T., Ramage, D., Ideker, T.: Cytoscape: a software environment for integrated models of biomolecular interaction networks. Genome research, 13 11, pp. 2498–2504. (2003)
16. Heim, P., Hellmann, S., Lehmann, J., Lohmann, S., Stegemann, T. (2009). Relfinder: Revealing relationships in rdf knowledge bases. In: Semantic Multimedia, pp. 182–187. Springer (2009)
17. Hungarian Electronic Library, `http://mek.oszk.hu/indexeng.phtml`

Service Composition for End-Users

Otto Hylli, Samuel Lahtinen, Anna Ruokonen, and Kari Systä

Department of Pervasive Computing
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
otto.hylli, samuel.lahtinen, anna.ruokonen, kari.systa@tut.fi

Abstract. RESTful services are becoming a popular technology for providing and consuming cloud services. The idea of cloud computing is based on on-demand services and their agile usage. This implies that also personal service compositions and workflows should be supported. Some approaches for RESTful service compositions have been proposed. In practice, such compositions typically present mashup applications, which are composed in an ad-hoc manner. In addition, such approaches and tools are mainly targeted for programmers rather than end-users. In this paper, a user-driven approach for reusable RESTful service compositions is presented. Such compositions can be executed once or they can be configured to be executed repeatedly, for example, to get newest updates from a service once a week.

1 Introduction

In service-oriented approaches, the focus is on the definition of service interfaces and service behavior. Service-oriented architecture (SOA) aims at loosely coupled, reusable, and composable services provided for a service consumer. SOA can be implemented by Web services, which is a technology enabling application integration. Web services can be used for composing high level composite services and business processes. Business processes are often realized as a service orchestrations implemented, for example, as WS-BPEL based processes [1]. WS-BPEL is targeted for composing operation-centric Web services utilizing WSDL and SOAP [2, 3]. WS-BPEL is close to a programming language defining the logic for a service orchestration. Thus, it is mostly used by IT developers.

In cloud computing, resources are provided to the user as services via the Internet. Cloud computing and SOA share similar interests on service reuse and service composition. Moreover, cloud computing emphasis on-demand services, which impose more requirements on flexible service and workflow configurations.

Compared to business processes, typical on-demand processes are personal, simpler, and their lifetime is shorter. Thus, on-demand processes are often characterized as instant service compositions and service configurations. Such processes are typically defined by the end-user instead of the developer of the cloud services. Due to instant nature of the on-demand processes, their usage and specification should be as simple as possible and require no installation of process development and management tools.

An end-user driven approach for WS-BPEL-based business process development has been proposed in [4]. The approach is targeted for providing a method for easy sketching of service orchestrations. In the proposed approach, a set of scenarios, given as UML sequence diagrams, are synthesized into a process description. However, in the context of cloud computing and on-demand processes, the use of UML modeling and standalone tools is not a proper solution.

Software services in the cloud, namely Software-as-a-Service (SaaS) applications, differ from fine-grained IT services, which are typically used to form business processes in SOA systems. SaaS applications are often targeted for end-users. They are self-contained and contain user-interfaces, business rules, and possible some metadata. In addition, such services often provide REST API instead of SOAP interface. Representational State Transfer (REST) is a resource-oriented architectural style developed for distributed environments such as for Web and HTTP based applications [5]. RESTful services provide an unified interface (GET, PUT, POST, DELETE) for data manipulation. Thus, composition of such services often includes combining resources and is characterized as mashup-type of development. Some guidelines for mashup development have been proposed (e.g. [6]). Composing RESTful services is still lacking tool vendor independent practices and description languages. Thus, the development is often done more in an ad-hoc manner.

A recent trend is cloud mashups, which combine resources from multiple services into a single service or application [7]. The provider of these service compositions can enhance the cloud's capabilities by offering new functionalities, which make use of existing cloud services, to clients.

In this paper, a semi-structured approach for developing personal service compositions is presented. The approach is targeted for end-user and allows composition of RESTful cloud services. The approach includes tackling the following issues: (1) easy sketching of service compositions using a simple visual language, (2) a mechanism to export/save composite descriptions for future usage i.e. reusable composite descriptions, and (3) an engine for executing the service compositions, once or repeatedly. The implementation is currently under development. The proposed tool support include a web browser based editor, which can be used to create simple on-demand service compositions.

The rest of the paper is organized as follows. In Section 2, we describe the overall approach and related components. In Section 3, two use cases for end-user driven service composition is presented. The proposed tool support is described in Section 4. In Section 5, related work and topics are discussed. In Section 6, conclusions and plan for future work are presented.

2 User-driven approach for service composition

In this paper, an end-user driven approach for defining personal service compositions is presented. The main goal of the approach is on easy design of service compositions, which requires minimal technical knowledge. The service composition is created by using GUI widgets, which are generated based on an imported service description. Widgets present individual resources and they can be dragged and dropped on the canvas. The user can draw dataflow pipes to connect the widgets. Incoming and outgoing dataflows are mapped to REST methods (e.g. outgoing dataflow for GETting a resource presentation).

The approach is supported by two components, designer Ilmarinen and engine Sampo. Ilmarinen is a client side application running in a web browser. Sampo is a server side application, which is an engine for running the service compositions. The composition description is given in XML-based format, called Aino description. As a service description format, the approach is based WADL descriptions [8]. It defines the resources, i.e., URIs, methods, and parameters. That is, while the Aino description specifies the service logic, the WADL description describe the service interface.

Sampo also plays a role of a service registry. Once a service is registered in Sampo engine, it can be used as a constituent service for future applications. One reason for providing a centralized registry, instead of letting the user search from the web, is that for RESTful services there is no agreement on one service description format. In case a third-party service do not have a compatible WADL description, it can be created afterwards and registered to Sampo. Thus, the approach allows using services, which do not natively provide a WADL description, as a reusable constituents.

The main focus of the approach is on easy design of service compositions, which requires minimal technical knowledge. The service composition is created by using GUI widgets, which are generated based on an imported service description. Widgets present individual resources and they can be dragged and dropped on the canvas. The user can draw dataflow pipes to connect the widgets. Incoming and outgoing dataflows are mapped to REST methods (e.g. outgoing dataflow for GETting a resource presentation).

The approach includes the following steps:

(1) query services from the service registry,
(2) select services to be used as a part of the compositions,
(3) composition described as a data flow between services, and
(4) send the composition description to the server engine to be executed.

The main steps are shown in Fig. 1. It also shows the relations of the main components and descriptions, Aino and WADL, which are used for importing and exporting data (i.e. service and composition descriptions).

3 Use cases

The following two use cases illustrate the possibilities offered by service compositions for regular internet users. They show how after encountering a normally labor intensive internet based task including multiple services, a user can pretty easily create a service composition that takes care of the task.

3.1 Use case 1: photos from Twitter to Flickr selectively

An avid Twitter user has been sending many photos taken with his smart phone directly to Twitter. The user wants a better way to organize and share his photos so he opens an account in Flickr which enables him to save photos to different albums, associate keywords to them and decide which photos are public. Uploading all his photos manually

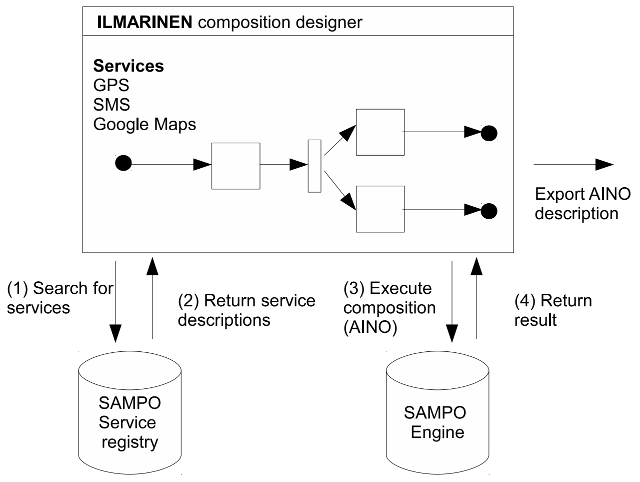


Fig. 1. The main steps of the approach

to Flickr would be tedious for the user. He would have to go through his Twitter time line, download each photo to his computer and then upload it to Flickr.

To automate the upload process the user wants to create a service composition. He opens the service composition editor Ilmarinen and chooses that he wants to get photos. Ilmarinen shows him a list of services from where he can get photos and he chooses Twitter. He also indicates that all photos shouldn't be fetched instead he will select the ones he wants. Then the user tells Ilmarinen that he wants to upload the photos selected in the previous step. From the services list shown by Ilmarinen he chooses Flickr as the upload target. Additionally he specifies that he wants to choose for each photo are they private or public. Lastly, he tells Ilmarinen that he wants to delete photos and chooses Twitter. He specifies that from Twitter he wants to delete those photos he has marked as private for Flickr.

When he executes the composition the execution engine Sampo first asks him to authorize Sampo's use of his Twitter and Flickr accounts. Authorization will be done by using OAuth [9] which means that the user authenticates to both services which then give access tokens to Sampo. Sampo will store these access tokens for later use if the user wants it so that next time a service composition using these services is run the user doesn't need to authenticate to the services. He just has to log in to Sampo. When the actual execution has started Sampo will first show the user all his photos from Twitter and asks him to choose those he wants. After that Sampo shows the user his previously chosen photos and asks which of them he wants to be private in Flickr. After the execution has finished Sampo shows the user a execution results summary which tells that the execution was a success and shows how many photos were processed in each step.

3.2 Use case 2: affordable reading

An enthusiastic book reader uses the Goodreads service in aid of her hobby. Goodreads is an online community for readers where users can search for books, rate and review them. Users can also categorize books in their profile by adding them to different shelves. One of these shelves is to-read where the user has been adding interesting books, which she has found through Goodreads' recommendation system. She wants to buy some new reading from her to-read shelf but due to her current poor economic situation she wants it to be as cheap as possible. Searching for each book's price from her favorite online book retailer Amazon and then comparing the prices manually would be time consuming so she decides to create a service composition to make the process quicker.

The user opens the service composition editor Ilmarinen and chooses that she wants information about books. Ilmarinen gives the user a list of services that deal with books. The user chooses Goodreads and indicates that she wants the content of a particular user's, in this case hers, particular shelf. Ilmarinen asks the user to input the name of the user and the name of the shelf which in this case are the user's Goodreads user name and to-read. Next the user tells Ilmarinen that she wants online shopping services. From the service list she chooses amazon.com. She specifies that she wants product information about the books from the previous step. Lastly she tells Ilmarinen that she wants the results in ascending order by price. When this composition is run the result is a table containing book information from Amazon including the price and a link to the Amazon product page where the book can be bought.

4 Implementation

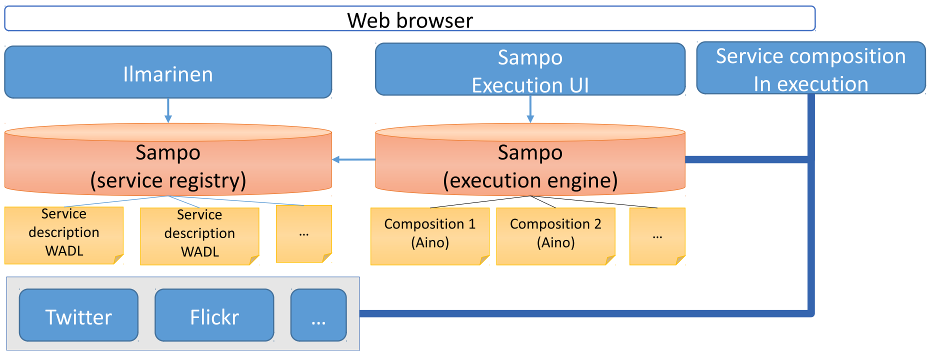


Fig. 2. High level architecture of the system

The prototype implementation consists of two main components: Designer Ilmarinen and Sampo Engine and Service registry. Sampo executes the services compositions, stores the service descriptions and offers Ilmarinen access to the information. Figure 2 illustrates the high-level architecture of the system. The user uses browser-based Ilmarinen to create service compositions. A service composition is a service. Its inteface is defined as a WADL document and its execution instructions are defined as an Aino description. Both XML documents are stored in Sampo. The user interacts with Sampo engine component is used to execute the compositions. The execution and possible user interaction related to the execution is again done in a browser based UI.

4.1 Service description

All the constituent services, as well as the service composition, are described as a WADL description. WADL description defines the web resources, provided methods and their parameters, as well as data types. Data types can be defined as separate XML schema files. An example of a simple service description is shown below. It has a partial definition of Twitter's get user timeline method which returns a specified number of tweets from the given user.

```
<?xml version="1.0" encoding="UTF-8"?>
<application>
    <grammars></grammars>
    <resources base="https://api.twitter.com/1.1">
        <resource path="statuses/user_timeline.json">
            <method href="getTimeline"/>
        </resource>
    </resources>
    <method name="GET" id="getTimeline">
        <request>
            <param name="screen_name" style="query" type="xsd:string" />
            <param name="count" style="query" type="xsd:integer" />
        </request>
        <response>
            <representation mediaType="application/json" />
        </response>
    </method>
</application>
```

4.2 Sampo Engine

Sampo engine is used in two ways, as a service registry and as an engine to execute the service compositions. Services can be added in the service registry as WADL descriptions. It provides the basic functionality for registration of the services, i.e. API for adding, removing, and searching the services. When a new WADL is added to Sampo the part of the categorization of the service and the resources can be done automatically based on the WADL and the user can complete the information and extend the suggested categorizations.

The given meta-information is used to offer Ilmarinen lists of the services. For instance, the user can ask to get a list of services related to pictures. Thanks to the meta-information Ilmarinen only needs to process WADLs of the services user adds to her composition instead of processing every WADL.

The other part of Sampo provides an API for executing Aino service descriptions. The service composition execution uses Aino and the corresponding WADL descriptions for getting the required information on the services and their API. The engine uses this information to invoke correct API calls to the services and combine the tasks to create the complete composite service.

Sampo contains a user interface for handling the compositions. The user can parameterize the composition and define time intervals of execution. In case of a recurring task the service page can be used to start and stop the compositions and change their time intervals. For instance, one could define a service composition that is launched weekly.

Sampo implements simple basic services, for example, for displaying images and news feeds. These are available as components in Ilmarinen and can be added to a service composition in similar fashion as external services.

4.3 Designer Ilmarinen

Ilmarinen is a client side application, which provides a graphical interface for creating the service compositions. The user is provided a simple visual environment for defining the service composition. The composition is done partially in a guided manner. A screenshot of an early prototype version of the tool is shown in Figure 3. The user can choose the services e.g. Twitter, BBC Program guide, Weather) she wants based on the service category (e.g. Social media, file storage, picture, program guides). For the services the user can define the interaction and the resources related to the interaction.

In the service composition key elements are the services and data flow between them. After adding a service one can see the input and output possibilities offered by it. These inputs and outputs are parameterized and services are connected to each other using them. When the user has finished, Ilmarinen generates the Aino description. This is exported to Sampo engine for execution. The composition is stored in Sampo and can be accessed directly using a corresponding link. That allows the users to access and execute the compositions directly without using Ilmarinen. This also enables sharing service compositions among different users.

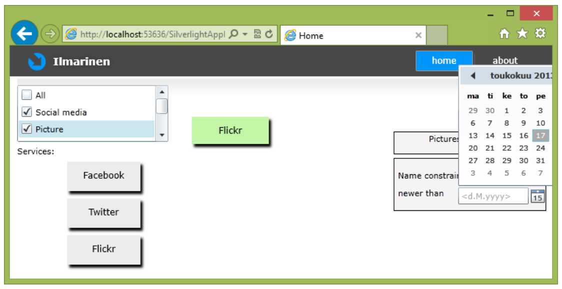


Fig. 3. Screenshot of Prototype of Ilmarinen

4.4 Composite description Aino

Aino description defines the resources involved in the composition and the composite dataflow among resources. A dataflow from one service to another means by getting resource presentation from one service with GET methods and using it as an input to another service using PUT, POST, or GET methods. Composite dataflows include three types of resources: resource out (for GETting a representation), resource in (for PUTting or POSTing), and resource in/out (for PUTting or POSTing and GETting). For data manipulation, control nodes, such as merge and select nodes, are used. In addition, data structures used for the resource presentation can be defined by attaching an XML schema to a dataflow or referring to a corresponding WADL file.

The composite dataflow can be modeled as an acyclic graph structure, which consists of resources, control nodes, and dataflow elements between them. Control nodes are used for manipulating resource representations. The main elements to compose the composite dataflow graph are shown in Fig. 4. Each resource is expected to have at most one incoming and outgoing dataflow element.

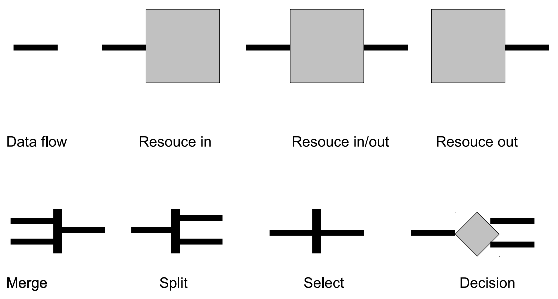


Fig. 4. Dataflow modeling

To enable importing and exporting of the Aino descriptions, composite dataflow graphs are transformed in XML format. The XML description consists of two main parts: resources and dataflow. The former describes all the resources involved in the composition. The latter defines the composite dataflow among the resources.

A simple composite dataflow consists of a sequence of method invocations, which are executed by the composite service on the constituent resources. These are presented as GET, PUT, POST, and DELETE elements in the XML description. In addition, the composite service can receive method calls. These are presented as onPUT, onGET, onPOST, and onDELETE elements. Corresponding request and response message types (including data types) are described in the services' WADL documents. These activities corresponding to REST operations are the same, which are used in BPEL for REST [10] proposal.

An example of Aino description is given in the listing below. It presents an example of uploading photos from Twitter tweets to Flickr. Resources part define two resources, Twitter and Flickr, which participate in the composition. The dataflow consists of a receive message and two message invocations. Execution starts when the client invokes GET method on the composite resource (onGET element). Execution continues with a sequence of two invocations. First the composite service invokes GET method on Twitter and second it invokes POST method on Flickr.

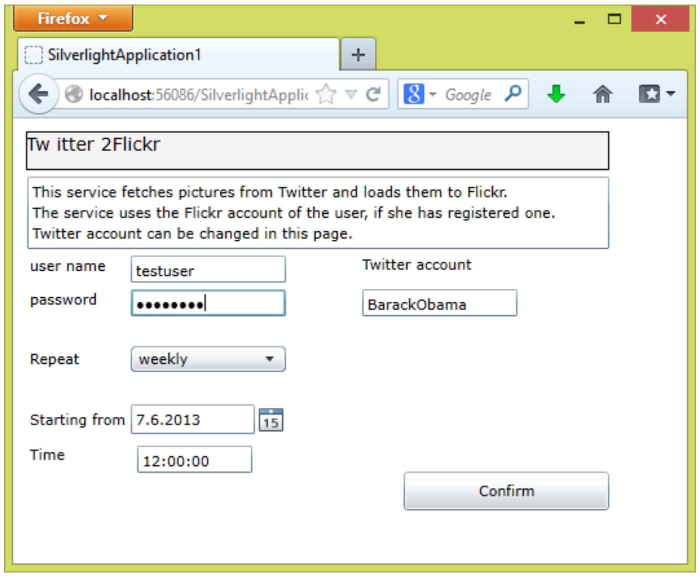
```
<?xml version="1.0" encoding="UTF-8"?>
<description name="tweet2flickr" >
<doc>Upload photos send to twitter to flickr.</doc>
    <services>
        <service name = "twitter" id="id1"/>
        <service name = "flickr" id="id2"/>
    </services>

    <resources>
        <resource uri="https://api.twitter.com/1.1/statuses/
               user_timeline.json"
        resource_id ="r1" service_id = "id1" />
        <resource uri="http://api.flickr.com/services/upload/"
 resource_id ="r2" service_id = "id2" />
    </resources>

    <variables>
        <variable name="screen_name" type="string" />
        <variable name="photos" type="photolist" />
    </variables>

    <dataflow>
    <onGET>
        <request>screen_name</request>
        <response></response>
        <resource_id>r_comp</resource_id>
        <sequence>
            <GET>
                <request>screen_name</request>
                <response>photos</response>
                <resource_id>r1</resource_id>
            </GET>
            <POST>
                <request>photos</request>
                <response></response>
                <resource_id>r2</resource_id>
            </POST>
        </sequence>
    </onGET>
</dataflow>
</description>
```

Variables are used for storing and manipulating message values. For example, the given code listing defines two variables, which correspond to input and output message types of the used GET and POST methods. *screen_name* variable presents a user name and it is passed as an input message for the GET method. A return message of the operation call is stored in *photos* variable and it is passed as an input message to the POST method.

screen_name is initialized, when the user fills-in the required input data, when she decides to run the composition (see Figure 5). A control interface is used for specifying process instance specific information, such as initial value of process variables and repetition information, which is not part of Aino description.

In addition to a sequence flow, Aino supports splitting, merging, and conditional branching of data flows. Example structures for *merge*, *split*, and *if-else* patterns are shown in the following listing.

```
<merge>
   <operand>
        activity
    </operand>
    <operand>
```



Tw itter 2Flickr

This service fetches pictures from Twitter and loads them to Flickr.
The service uses the Flickr account of the user, if she has registered one.
Twitter account can be changed in this page.

user name: testuser

password: ••••••••

Twitter account: BarackObama

Repeat: weekly

Starting from: 7.6.2013

Time: 12:00:00

Confirm

Fig. 5. A Control User Interface for the service Compositions

```
       activity
    <operand>
</merge>
<sequence>
    some activity
</sequence>

<sequence>
    some activity
</sequence>
<split>
    <operand>
        activity
    </operand>
    <operand>
        activity
    </operand>
</split>

<if>
   <condition>some conditon expression</condition>
   activity
   <elseif>*
      <condition>some condition expression</condition>
      some activity
   </elseif>
   <else>?
      some activity
   </else>
</if>
```

5 Related work

The idea of cloud computing is based on on-demand services, which are provided as SaaS applications. In the cloud, traditional business process management tools are already available as SaaS. However, they are targeted for design and management of structured business processes. Requirements for on-demand processes differ from traditional BPM. The ideal situation is to provide easy and instant mechanism to support execution of personal and dynamic processes, which utilize existing SaaS applications available on the cloud.

5.1 Tools for mashup development

Ad-hoc processes are often expected to live only a short time. The lack of documentation and proper design might make them single-use only. Thus, they may not be reusable and flexible, but they always need to be recomposed.

JOpera [11] is an Eclipse-based tool build for composing SOAP/WSDL and RESTful Web services. For software developers it provides many useful features such as process modeling, debugging and execution. For composing RESTful services JOpera uses BPEL for REST [10]. BPEL for REST is an extension to WS-BPEL to support compositions of RESTful Web services. The approach does not rely on usage of WSDL or other service descriptions. Resources are defined in the BPEL for REST description as a resource construct, which defines the resource URI and supported operations.

In [12], Marino *et al.* present HTML5-based prototype tool support for mashup development. They present a visual language for service composition. However, the paper is missing details on the user interface and tool usage. Also, details on the composition description are not given.

In [13], Aghee *et al.* discuss different types of mashups enabled by HTML5. A case example includes a location sensitive mobile mashup. The mashup runs natively in a mobile device and uses GPS sensor build-in the device. In addition, it uses external Web APIs. Location data is sent to a server, which executes API calls to external services. This enables sharing the application between several uses. Mobile mashups enable use of real-time data gathered from the sensors in a mobile phone, e.g. real-time navigation.

Bottaro *et al.* present a simple visual language for composing location-based services [14]. The user uses a repository of web widgets. Widgets are dragged and dropped to build UI for the application. The application logic is defined by drawing connections between data widgets.

In [15], Grönvall *et al.* present ongoing work on user-centric service composition. GUI elements are prototypes of service invocations, which can be chained to compose data flows among services. They present a lightweight tool support for composing simple dynamic workflows, such as for combining SMS, email, and calendar services. Instead of modeling complicated workflows, the emphasis is on the user experience.

In EzWeb project [16, 17], a service-oriented platform for end-user mashups development have been built. The idea is to provide gadgets (e.g. Twitter, Flickr) the user could add to her ”‘application page”‘ creating a set of different applications and web services. The user can also define dataflow between the gadgets by connecting ”‘events”’ the gadgets could give, e.g., an image url could be connected to another image displayer gadget that is able to show the picture. All these gadgets are implemented for EzWeb environment. That is, implementation of their user interface, way of communicate with servers, their events and event slots, are specific for the EzWeb environment. In our approach, the aim is to provide means to compose existing services together and execute these compositions. Thus, our target is to support composition of any third party services by introducing their service descriptions to our system.

5.2 Describing service compositions

Some approaches for modeling and describing RESTful service compositions have been proposed. Guidelines for UML modeling of RESTful service compositions is presented in [18] by Rauf *et al.* The static resource structure is modeled using class diagrams. The behavioral specification of the composite service is given using state chart diagrams.

In [19,20], Zhao *et al.* discuss formal describing of RESTful services and resources as well as RESTful composite services. Their main interests is on supporting automatic service compositions. For service compositions they present a logic-based synthesis approach utilizing linear-logic and pii-calculus.

In [21], Alarcon *et al.* state that many of the recent service composition approaches rely on operation-based models and neglect hypermedia characteristics of REST. As a solution for composing RESTful services, they present a hypermedia-driven approach realized by using resource linking language (ReLL) for service description. The approach aims to support machine-clients by enabling automatic retrieving of resources from a web site. For describing the composite resources PetriNets are used. As an example of a composite resource, a social network application was presented.

6 Conclusions

Cloud computing is based on on-demand services, which should be available as needed. Similarly, it should also enable on-demand service compositions. In this paper, end-user driven approach for personal service composition have been presented. The proposed tool support includes an editor running in a web browser and a server-side engine for storing and executing service compositions. The editor is designed for the end-users and it is used for sketching personal service compositions. It focuses on end-user concepts and aims to hide complicated and unnecessary information, e.g. service descriptions, which are handled by the engine. Instead of handling data types, the user is allowed to use concepts such as a picture or a photo gallery. The presented use cases concentrate on combining social media services into a composite service. Also, the user is allowed to define repeatable executions for checking updates from the services.

To characterize the approach, it is designed for cloud environment providing a browser-based tool for building service compositions. It is based on WADL descriptions, which are also used for generating GUI widgets for the end-user. In addition, it enables defining RESTful workflows as a composite services.

Our future work includes finalizing the implementation and conducting case studies on applying the approach utilizing the developed tool support. Our future plans also include experimenting the tool usage with novice users.

References

1. Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services Version 1.1, May 2003. http://www.ibm.com/developerworks/.
2. W3C, http://www.w3.org/TR/wsdl. *Web Services Description Language (WSDL) 1.1*, 2001.
3. W3C, http://www.w3.org/. *Simple Object Access Protocol (SOAP) 1.2*, 2007. Last visited December 2011.
4. Anna Ruokonen, Lasse Pajunen, and Tarja Systa. Scenario-driven approach for business process modeling. *Web Services, IEEE International Conference on*, 0:123–130, 2009.
5. Roy Thomas Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.
6. Tommi Mikkonen and Arto Salminen. Towards a reference architecture for mashups. In *Proceedings of the 2011th Confederated international conference on On the move to meaningful internet systems*, OTM'11, pages 647–656, Berlin, Heidelberg, 2011. Springer-Verlag.
7. Mukesh Singhal, Santosh Chandrasekhar, Tingjian Ge, Ravi Sandhu, Ram Krishnan, Gail-Joon Ahn, and Elisa Bertino. Collaboration in multicloud computing environments: Framework and security issues. *Computer*, 46(2):76–84, 2013.
8. W3C, http://www.w3.org/Submission/wadl/. *Web Application Description Language (WADL)*, 2009.
9. Internet Engineering Task Force (IETF), http://tools.ietf.org/html/rfc6749. *The OAuth 2.0 Authorization Framework*, 2012.
10. Cesare Pautasso. RESTful web service composition with BPEL for REST. *Data Knowl. Eng.*, 68(9):851–866, September 2009.
11. Cesare Pautasso. Composing RESTful services with JOpera. In *International Conference on Software Composition 2009*, volume 5634, pages 142–159, Zurich, Switzerland, July 2009. Springer.
12. Enrico Marino, Federico Spini, Fabrizio Minuti, Maurizio Rosina, Antonio Bottaro, and Alberto Paoluzzi. HTML5 visual composition of rest-like web services. In *4th IEEE International Conference on Software Engineering and Service Science (ICSESS 2013)*, 2013. To appear.
13. Saeed Aghaee and Cesare Pautasso. Mashup development with HTML5. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, Mashups '09/'10, pages 10:1–10:8, New York, NY, USA, 2010. ACM.
14. Antonio Bottaro, Enrico Marino, Franco Milicchio, Alberto Paoluzzi, Maurizio Rosina, and Federico Spini. Visual programming of location-based services. In *Proceedings of the 2011 international conference on Human interface and the management of information - Volume Part I*, HI'11, pages 3–12, Berlin, Heidelberg, 2011. Springer-Verlag.
15. Erik Grönvall, Mads Ingstrup, Morten Pløger, and Morten Rasmussen. Rest based service composition: Exemplified in a care network scenario. In Gennaro Costagliola, Andrew Jensen Ko, Allen Cypher, Jeffrey Nichols, Christopher Scaffidi, Caitlin Kelleher, and Brad A. Myers, editors, *VL/HCC*, pages 251–252. IEEE, 2011.
16. D. Lizcano, J. Soriano, M. Reyes, and J.J. Hierro. EzWeb/FAST: Reporting on a successful mashup-based solution for developing and deploying composite applications in the "upcoming ubiquitous SOA". In *Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008. UBICOMM '08. The Second International Conference on*, pages 488–495, 2008.
17. David Lizcano, Javier Soriano, Marcos Reyes, and Juan J. Hierro. EzWeb/FAST: reporting on a successful mashup-based solution for developing and deploying composite applications in the upcoming web of services. In *Proceedings of the 10th International Conference on*

Information Integration and Web-based Applications & Services, iiWAS ’08, pages 15–24, New York, NY, USA, 2008. ACM.

18. Irum Rauf, Anna Ruokonen, Tarja Systä, and Ivan Porres. Modeling a composite RESTful web service with UML. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA ’10, pages 253–260, New York, NY, USA, 2010. ACM.
19. Xia Zhao, Enjie Liu, G.J. Clapworthy, Na Ye, and Yueming Lu. RESTful web service composition: Extracting a process model from linear logic theorem proving. In *Next Generation Web Services Practices (NWeSP), 2011 7th International Conference on*, pages 398–403, Oct.
20. Haibo Zhao and P. Doshi. Towards automated RESTful web service composition. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 189–196, July.
21. Rosa Alarcon, Erik Wilde, and Jesus Bellido. Hypermedia-driven RESTful service composition. In *Proceedings of the 2010 international conference on Service-oriented computing*, ICSOC’10, pages 111–120, Berlin, Heidelberg, 2011. Springer-Verlag.

Towards a Reference Architecture for Server-Side Mashup Ecosystem

Heikki Peltola and Arto Salminen

Tampere University of Technology,
Korkeakoulunkatu 10, 33720,
Tampere, Finland
{heikki.peltola, arto.salminen}@tut.fi

Abstract. The Web has more and more services providing resources – data, code, and processing – with possibility to reuse them in other contexts instantly. Many of these services offer an interface that allows other services to access the data or use the provided processing capabilities. Mashups are web applications that act as content aggregates that leverage the power of the Web to support instant, worldwide sharing of content. However, quality and other attributes of the service interfaces used by mashups are diverse. Accessing data from multiple services and transforming the data to a desired format is laborious for the software developer and slow on the client-side. To avoid combining the same data several times, it would be wise to do the combining once and store the result for later use. Server-side mashup offers credential management to external services, preformatted data storage, and interface for retrieving the data with minimal delay. This paper discusses requirements for a server-side mashup and presents a reference architecture for server-side mashup ecosystem. Additionally, an implementation for wellness services based on the reference architecture is presented.

Keywords: Mashup, architecture, server-side.

1 Introduction

Despite its origins in sharing static documents, the Web has become a software platform. Today, majority of new applications intended for desktop computers are released as web-based software. This development has its disadvantages, but numerous benefits as well. The web-based software is available all over the world instantly after the online release. It can be used and updated without the need to install anything. Applications can support user collaboration, i.e., allow users to interact and share the same applications over the Web. In addition, numerous web services allowing users to upload, download, store, and modify private and public resources have emerged. These resources can include private resources, such as personal images, texts, videos, e-mails, etc. as well as public data such as stock quotes, weather data, and news feeds. Typically accessing the personal content is restricted because of privacy reasons in contrast to public data that can be used without such limitations.

An important realization is that applications built on top of the Web do not have to live by the same constraints that have characterized the evolution of conventional desktop software. The ability to dynamically combine content from numerous web sites and local resources, and the ability to instantly publish services worldwide has opened up entirely new possibilities for software development. In general, such systems are referred to as mashups, which are content aggregates that leverage the power of the Web to support instant, worldwide sharing of content. By connecting to multiple source services a mashup becomes a node in so called mashup ecosystem [1].

As expressed by Bosch, "a software ecosystem consists of the set of software solutions that enable, support and automate the activities and transactions by the actors in the associated social or business ecosystem and the organizations that provide these solutions" [2]. Since mashups by definition combine data from multiple sources, the stakeholders that provide this data form an ecosystem, i.e. a set of entities that act as a single unit instead of each participating business acting separately [3]. In [2] mashups are categorized as "End-User Programming Software Ecosystems" and two example ecosystems, Yahoo! Pipes and Microsoft PopFly, are mentioned.

Managing a mashup ecosystem is not trivial. Many of the services offer an interface that allows other services to access the information. However, quality and other attributes of these interfaces are varying. The information may come in different formats, therefore comparing and combining the information is not straightforward. While some of the content is public and can be accessed in a liberal fashion, other resources are accessed through a restricted interface with per-user credentials. Therefore, credential management becomes an issue as well.

Despite the popularity of creating mashups, the current approach towards mashup architecting has been described as "hacking, mashing, and gluing" [4]. It is difficult to find general-purpose tools or uniform development guidelines for mashups. However, there are commonalities in goals of different mashup systems. Important quality attributes, such as security, performance, availability, and modifiability should not be overlooked. To solve these issues, we propose a plug-in based mashup ecosystem architecture where credential management, as well as formatting, converting, and analysing data are the primary design goals. The architecture hides the complexity of accessing multiple services with diverse data formats. It is used as a backend for other services, for instance other mashups acting as clients from the point of view of this system.

Our research approach was Action Design Research method [5]. It emphasizes the organizational context and its impact on the studied artifact during development and use. The research process contains inseparable and interwoven activities of building the artifact, intervening in the organization, and evaluating it concurrently. Our research consists of research cycles with studying a phenomenon, applying acquired information and deriving architecture, as well as implementation. These cycles are repeated to achieve the required results.

This paper presents a reference architecture for a server-side mashup ecosystem. Requirements for the architecture are derived and design decisions are discussed. Additionally, an implementation for wellness services based on the reference architecture is presented. The structure of the paper is as follows, in Section 2 we describe requirements for a server-side mashup architecture. Section 3 presents a reference architecture for server-side mashup. Our implementation based on the reference architecture is described in Section 4. Section 5 discusses the presented architecture and Section 6 presents related work. Finally, we draw conclusions in Section 7.

2 Deriving Requirements for Server-Side Mashup Architecture

While our proposed architecture does not enable end-user programming in the current implementation, it forms a software ecosystem, which consists of infrastructure, data producers, processors, and consumers. Infrastructure includes sensors and other hardware that is used to collect data. Producers are the data source services that act as inputs to the system. The data is further processed to increase its value and to derive emergent information, if possible. Consumers are using the data and visualizing it to the users. For example in the wellness domain there are numerous devices used to collect data including activity trackers, weight scales, and sleep analysers, to name a few. People are using these devices to measure themselves. The created data is stored in the device manufacturers' web services. These web services offer the data through their APIs and act as data producers. The provided data is analysed by the data processors and consumed by the client programs that are offering the data for the users.

Mashups are combining information from multiple sources to offer a new experience to the user. Combining the data can be done either on the client-side or the server-side. Retrieving and combining data from multiple sources on the client-side may add delays in the user interface. To avoid combining the same data several times, it would be wise to do the combining once and store it for later use. With server-side data aggregation, we are moving processing from the client to the server and gain a faster service, with clients depending on responses only from the mashup server. Server-side mashup can get new data from other services as soon as it is available and store it for later use. The clients must have a good connection only to the mashup server to offer a fluent user experience. Furthermore, a deeper analysis with a large dataset cannot be done on the fly on the client-side.

We aim at offering a reference architecture that allows users to get their data, no matter what service their data is located at. The architecture must be extensible and able to combine information from multiple sources. In the following, we discuss requirements for a server-side mashup architecture and things that should be taken into consideration.

Accessing data. Mashups are very dependant on the services that are providing the data. Some data sources are open to all, such as news or weather information, while other sources require authorization and authentication. To access users' personal data from other services, the user must give authorization for the mashup server. If OAuth authentication is used, the server receives an access token, which is then used to make authorized requests to the services. Mashup servers are never finished: sources of data disappear entirely, data formats change, and new sources become available. The server-side architecture must allow data sources to be modified with a reasonable amount of work.

Storing data. Gathering data from several services might be time consuming, and it is heavily reliant on the availability of the services. If a service is unavailable at a certain moment, it may take several seconds or longer to realise it and recover. This would not look good to the user and it would mean the data from that service would be unavailable. In addition, the required dataset might be large, leading to heavy network traffic. With centralized up-to-date data storing from multiple services, we avoid these problems. The data will always be available, providing faster responses to the users.

Unifying data. The data sources may vary for different users, depending on what services the user is actively using or has used in the past. The same type of data may come from multiple sources in different formats. If the data is describing the same information in different formats, the data must be transformed and offered in a uniform way. For the client that is using the mashup server, it does not necessarily matter where the data originates from. Data unification is not only transforming data from one format to another. It can also include data comparison and deriving emergent data. For example calculating averages from a certain period of time, finding minimum or maximum values from a large data-set, or figuring out trends based on how values have changed recently. Furthermore, some data can be converted from one type to another as services are not offering data in all possible types. Unification may add data value for some sources simply by offering derived data.

Providing an API. The most important part of a mashup server is the API it provides to clients. The whole server must be built so that API requests can be answered within a reasonable time. If the API is not easy to use, is poorly documented, or has other serious flaws like constantly changing interfaces, clients will stop using the service and find another one. The API is used to authorize the server to access users' data from external services as well. The API should provide access to raw data in the form it is available in the source services, and also in a unified format. Providing the raw data in parallel with the unified data enables maximized flexibility. When clients use the unified data, additional data transformations are not required, which helps comparing and presenting the data. However, if the original data format is desired, it is also accessible through the mashup back-end.

3 Reference Architecture

Based on the requirements presented in the previous section, this section provides a reference architecture for a server-side mashup ecosystem. The ecosystem

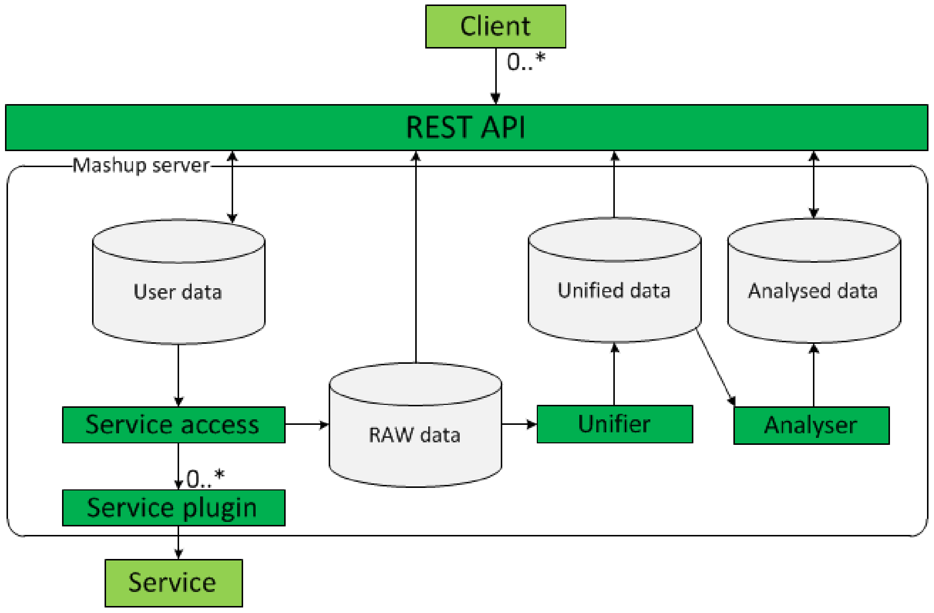


Fig. 1. Server-side mashup reference architecture.

consists of data in different services, the mashup server and client programs. The data in source services include private user data and public data. The mashup server gathers and processes all of the data and offers it through an API to client applications.

Fig. 1 presents our server-side mashup reference architecture. The client programs are making requests to the mashup server using the REST API. Authorization to access users' data from external services must be given and access tokens are stored for later use. Service access component handles accessing the raw data from the external services. The raw data is stored for later use without any data manipulation on the mashup server. All raw data is immediately unified with the unifier component and stored in Unified data. In addition, further data processing can be done by the analyser, for example merging data from different sources. All user's raw, unified, and analysed data are offered through the REST API to client programs. Client programs can do further data analysis and store their results in the Analysed data storage. Client programs can overwrite or remove only data they have entered themselves. However, in the current approach, access to user's data allows accessing all of the user's analysed data, including analysis made by other clients.

Databases. Database structure is divided into four distinct parts: 1) User data, 2) RAW data, 3) Unified data, and 4) Analysed data. User data stores user information and authentication parameters for clients using the REST API. Authorization parameters used for accessing external services are stored here as well. RAW data stores all requested data received from the external services.

It acts as a cache, allowing fast and reliable access to all of the unmodified data. Unified data stores and offers the raw data transformed to a common representation. Analysed data has even further processed data that can have additional input from the users, such as questionnaires.

Database schema for the User data is static, since we know beforehand what kind of information we are expecting, a relational database is well suited for this. The raw data is by nature large amounts of data chunks, which is ideal for NoSQL databases. A relational model is not needed, the focus is on storing great quantities of data using key-value pairs in associative arrays. The data can be stored for example in JSON format, just as it is retrieved from the source services. The database for analyzed data may get new and unexpected fields as the service progresses over time. This suggests that the use of NoSQL database is appropriate for the analysed data as well. Additionally, MapReduce framework [6] is implemented by most of the NoSQL databases and can be used in analysing large volumes of data [7]. Using NoSQL database for the raw, unified, and analysed data gives more freedom on how the data is stored. As noted in [8], NoSQL databases trade consistency and security for performance and scalability. This leads to adding more responsibility on the developers. Even if the database does not have a predefined schema, it must not be used as a bucket that you can throw your data to and expect it to stay organised.

New data. New data is constantly added and available in the data source services. To keep the mashup server data up to date, new data must be fetched and stored on the mashup server whenever it is available. Some services offer Publish-Subscribe functionality (PubSubHubbub) [9], with the service sending notifications whenever there is new data available. When the services do not offer subscribing for notifications, we are forced to use polling. Background workers can be used to handle the polling. To keep the databases coherent, new data is unified immediately. This allows faster responses to the users and removes unnecessary inconsistency with some data unified and some not.

Fig. 2 illustrates data flow from data source service to the user client. Service access subscribes each user for new data notifications (steps 1-2). When new data is available, data source service sends a notification to service access (step 3). Service access requests the new data and stores it to the RAW data-database (steps 4-6). The raw data is unified, analysed and stored (steps 7-10). User client can request unified and analysed data (steps 11-14).

Expendability. Mashups gather information from numerous different sources. The source services may change or disappear, and we may also want to add new services. The server-side implementation must be done so that it is easy to fix issues that emerge related to changing APIs. Further, adding new services and removing used services must be possible. Adding new services is always laborious, due to the fact that all services are different. Services offer different kinds of data, through different kinds of APIs. Access to services is managed using a plug-in architecture. A plug-in is required for each service that is linked to the system. Each plug-in is responsible for handling communication to one service. A plug-in must describe the raw data, so that it can be offered

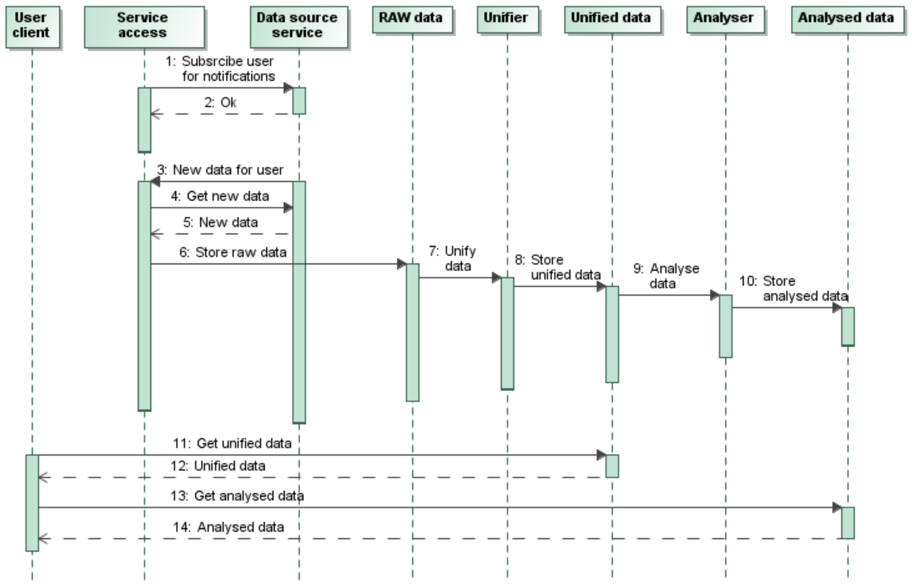


Fig. 2. Data flow from data source service.

through the REST API, and additionally specify how to unify the raw data into a predefined format.

4 Architecture Implementation for Wellness Services

In this section, we present our implementation based on the presented reference architecture. The domain of the implementation is wellness services, with users keeping track of information about their well-being such as activity, weight, blood pressure, and quality of sleep. The aim of the implementation is to provide an API for client programs with centralized access to all of the users' wellness information. The API should provide access to raw data from a wide range of wellness services, and also unified data that is not dependant on the origin of the data. For example, it does not matter where user's activity data is from, the important thing is that it can be shown in a unified way with other relevant information. Fig. 3 presents our server-side architecture for a wellness mashup.

The current implementation concentrates on services that offer an open API and have licenses that allows us to modify and store the provided user data. The services that we selected for the first phase of the implementation are Beddit (http://beddit.com), Withings (http://withings.com), Fitbit (http://fitbit.com), and a weather service (http://wunderground.com). Beddit offers sleep tracking and analysis. Withings provides blood pressure monitors, body scales, and activity trackers. Fitbit offers body scales and activity trackers. The body scales measure weight and body fat, and also calculate the body mass index (BMI).

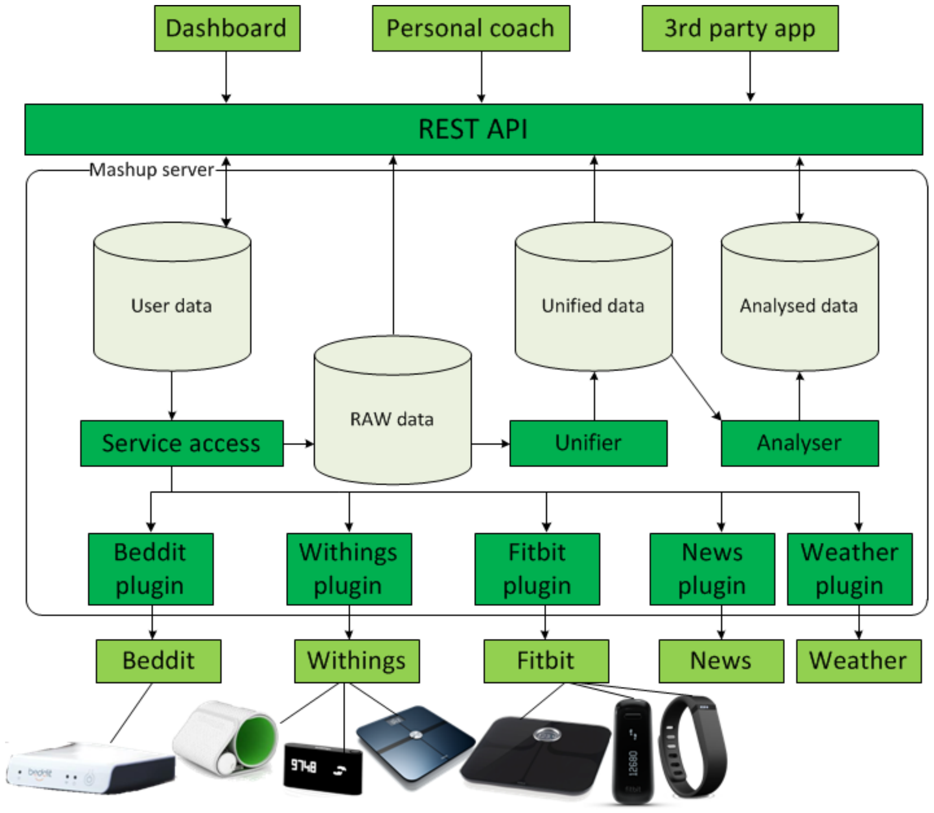
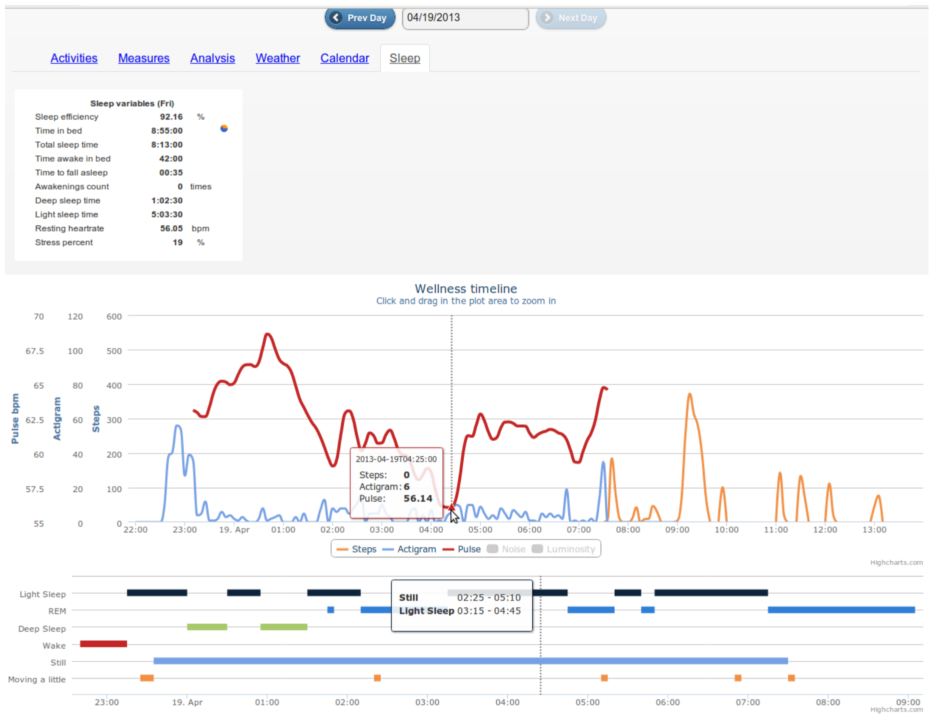


Fig. 3. Server-side mashup architecture for wellness services.

Activity trackers are carried in a pocket all the time and they are measuring steps taken, distance travelled, and amount of stairs climbed. Fitbit and Withings are offering notifications whenever new data is available. The notifications do not have the actual payload, they only report that new data is available. Beddit does not offer such a service, so we have a background worker polling for new data. As soon as new data is stored to the RAW data database, the data goes automatically through the Unifier module and Unified data is created.

There are overlapping devices measuring the same phenomena, for example sleep. The number of parameters are varying, but there are similar attributes such as duration of sleep and time to bed. The Unifier component searches for common parameters that are found in different data sources or can be derived from the data that is offered. Naturally, there are parameters that can be found only in a single data source. For example, Beddit gives luminosity and noise measurements with the sleep data, whereas Fitbit does not measure these. Hence, Unified data has common parameters that are comparable from different sources, and parameters that can only be found in certain sources.



Sleep variables (Fri)		
Sleep efficiency	92.16	%
Time in bed	8:55:00	
Total sleep time	8:13:00	
Time awake in bed	42:00	
Time to fall asleep	00:35	
Awakenings count	0	times
Deep sleep time	1:02:30	
Light sleep time	5:03:30	
Resting heartrate	56.05	bpm
Stress percent	19	%

Fig. 4. Wellness dashboard.

In pursuit of analysing sleep quality we have implemented methods for calculating Pittsburgh Sleep Quality Index (PSQI) [10]. PSQI provides a standardized measure of sleep quality, based on different areas related to sleep, such as sleep latency, sleep disturbance, and daytime dysfunction over the last month. The object of the analysis self rates the areas by filling a questionnaire form. Based on completed questionnaire, we can calculate the PSQI-value.

We have also implemented a Dashboard client to visualize the user's wellness related information, presented in Fig. 4. It shows data gathered from multiple sources and it can be personalised for different needs. The visualization may be used to find out relations between activities on the day and sleep quality during the night. It also motivates users by showing progress and trends on recent meters, such as weight, blood pressure, activity, and quality of sleep.

5 Discussion

We conducted this work using action design research method. During the iteration process, we noticed that the mashup server implementation is heavily affected by organizational structure. For instance, during the early iterations an external unit was assigned to implement the analysis module of the system.

Therefore we designed this part of the system to be separated from the rest and an API to communicate with the analysis module was introduced. However, the analysis was later decided to be implemented as an integrated part, and having a full-fledged API for this purpose was unnecessary. Consequently, the current implementation now includes a method for clients to send analyzed data back to our server, but in restricted fashion.

Forcing users to perform "OAuth dance" at the time of first use setup or when new services are connected to user accounts is not good practice in the user experience point of view. Utilizing so called "two-legged OAuth" approach does not require user action, and it allows a service to access some data with application specific credentials. Typically sensitive user data is not accessible with this approach. Our view is that two-legged OAuth should be used if it enables accessing the relevant data to promote maximized convenience for the user. Another way to avoid bad user experience in service authentication is to promote OpenID (http://openid.net/) and other credential federation approaches. Sometimes this might require partnering with data source service providers.

The most relevant way to compose a mashup is not always obvious for application developers. One indication of this is the fact that mashup creation is often regarded as end-user activity which is supported by dedicated, sometimes domain specific tools. The wellness domain is not an exception. It is difficult to "guess" the end-user's personal desires in his or her physical well-being. Therefore enabling end-users themselves or domain experts, such as personal coaches or other professionals, to determine how the data is processed might be beneficial.

6 Related Work

In general, mashup development has gained a lot of research interest recently. Different patterns and trends in mashup development can be identified. For example, Wong et al. have categorized mashups into five different groups: aggregation, alternate UI & in-situ use, personalization, and focused view of data and real time monitoring [11]. Another paper by Lee et al. presents seven mashup patterns: data source, process, consumer, enterprise, client-side, server-side and developer assembly mashups [12]. In addition, a number of challenges related to mashup development have been pointed out. As stated by Zang et al. [13], mashup developers encounter problems mainly in three areas: API functionality, documentation and coding details. Issues related to API functionality in their research were, for example, authentication and performance problems. Some developers were concerned about the lack of proper documentation at all levels, including API reference, tutorials and examples. The programming skills needed for creating compelling mashups in JavaScript were also identified as hard to learn. Finally, the relation of disciplined software engineering principles and mashup development or, even more generally, the development of web applications remains vague [14].

Our previous research efforts include design of a specialized mobile multimedia mashup ecosystem [15]. The architecture was heavily based on existing backend server, and therefore the approach used is not ideal for a reference architecture. However, the study in [15] clearly shows the benefits of having a server-side backend, especially when there is a desire to analyse user's past actions. In another paper, we have proposed a reference architecture for client-side mashups[16].

Server-side mashup tool architecture based on layers has been studied by López et al. [17]. Their architecture consists of four layers: source access (accesses web resources), data mashup (creates structural presentation of the data), widget (holds all widgets available in the system) and widget assembly (creates a user interface) layer. In addition to the layers, the architecture includes common services that provide general functionalities and can be used from any layer. The result mashup created with this architecture is similar to a web portal with the exception that the widgets are connected.

7 Conclusion

This paper discusses problems and pitfalls in creating a server-side mashup and derived requirements for a reference architecture. A reference architecture was presented, as well as an implementation based on the reference architecture. The reference architecture is focusing on fast and reliable data access for client programs. The data is offered in raw, unified, and analysed formats to serve a broad range of clients with different requirements.

Building a system that gathers data from multiple sources may be time consuming, due to many details that must be taken into consideration when accessing the data. A service that is gathering all of the data from different sources and offering it in a unified format allows developers using the mashup service to focus on other important aspects, such as analysing data and presenting data to the end-users.

Future work of the implementation is to widen the range of data source services, for example Twitter messages and news. We hope to find new and unexpected correlations with data from different sources, for example with news, weather, activity, or sleep. In addition, more elaborate data analysis is planned. The analysis can be based purely on the data from the external services, in addition to information about user's mood and subjective opinions that can be acquired using simple voting systems or questionnaire forms, for example. Furthermore, support for end-user programming with tool support is in the scope of our future research. Finally, we plan to create more client programs that can benefit from the server-side mashup.

References

1. Yu, S., Woodard, C.J.: Innovation in the programmable web: Characterizing the mashup ecosystem. In: Service-Oriented Computing–ICSOC 2008 Workshops, Springer (2009) 136–147

2. Bosch, J.: From software product lines to software ecosystems. In: Proceedings of the 13th International Software Product Line Conference, Carnegie Mellon University (2009) 111–119
3. Messerschmitt, D.G., Szyperski, C.: Software ecosystem: understanding an indispensable technology and industry. MIT Press Books **1** (2003)
4. Hartmann, B., Doorley, S., Klemmer, S.R.: Hacking, mashing, gluing: Understanding opportunistic design. Pervasive Computing, IEEE **7**(3) (2008) 46–54
5. Sein, M., Henfridsson, O., Purao, S., Rossi, M., Lindgren, R.: Action design research. MIS Quarterly **35**(1) (2011) 37–56
6. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM **51**(1) (2008) 107–113
7. Bonnet, L., Laurent, A., Sala, M., Laurent, B., Sicard, N.: Reduce, you say: What NoSQL can do for data aggregation and BI in large repositories. In: Database and Expert Systems Applications (DEXA), 2011 22nd International Workshop on, IEEE (2011) 483–488
8. Okman, L., Gal-Oz, N., Gonen, Y., Gudes, E., Abramov, J.: Security issues in NoSQL databases. In: Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on, IEEE (2011) 541–547
9. Fitzpatrick, B., Slatkin, B., Atkins, M.: Pubsubhubbub core 0.3. http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.3.html (2010)
10. Buysse, D.J., Reynolds, C.F., Monk, T.H., Berman, S.R., Kupfer, D.J.: The Pittsburgh sleep quality index: a new instrument for psychiatric practice and research. Psychiatry research **28**(2) (1989) 193–213
11. Wong, J., Hong, J.: What do we mashup when we make mashups? In: Proceedings of the 4th international workshop on End-user software engineering, ACM (2008) 35–39
12. Lee, C.J., Tang, S.M., Tsai, C.C., Chen, Y.C.: Toward a new paradigm: Mashup patterns in web 2.0. WSEAS Transactions on Information Science and Applications **6**(10) (2009) 1675–1686
13. Zang, N., Rosson, M.B., Nasser, V.: Mashups: who? what? why? In: CHI'08 extended abstracts on Human factors in computing systems, ACM (2008) 3171–3176
14. Mikkonen, T., Taivalsaari, A.: The mashware challenge: bridging the gap between web development and software engineering. In: Proceedings of the FSE/SDP workshop on Future of software engineering research, ACM (2010) 245–250
15. Hartikainen, M., Salminen, A., Kallio, J.: Towards mobile multimedia mashup architecture. In: Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on, IEEE (2012) 439–445
16. Mikkonen, T., Salminen, A.: Towards a reference architecture for mashups. In: On the Move to Meaningful Internet Systems: OTM 2011 Workshops, Springer (2011) 647–656
17. López, J., Pan, A., Bellas, F., Montoto, P.: Towards a reference architecture for enterprise mashups. Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos **2**(2) (2008)

Code Oriented Approach to 3D Widgets

Anna-Liisa Mattila

Department of Pervasive Computing,
Tampere University of Technology,
Korkeakoulunkatu 1, FI-33720 Tampere, Finland
anna-liisa.mattila@tut.fi

Abstract. With newly introduced standards, it has become increasingly feasible to develop interactive 3D applications even with web technologies only. Unfortunately many of the available 3D graphics libraries are built on low-level facilities, which, while well-suited for demos, complicate the design of true applications. The research of interactive 3D applications has mainly focused on developing interaction techniques, input- and output devices and figuring out what would be right metaphors and paradigms to develop usable and still highly interactive 3D user interfaces. Little consideration is put on how to actually ease the programming of an interactive 3D application. In this paper, we explore the current 3D research and tools used for developing interactive 3D applications addressing the missing abstraction of code oriented 3D widgets.

Keywords: Widgets, 3D, 3DUI, WebGL

1 Introduction

As 3D technologies have become more common, it has become increasingly feasible to develop interactive applications even with web technologies only, with no vendor-specific plugins that would require separate installation. However programming 3D applications is still difficult task.

The research of interactive 3D applications has mainly focused on developing interaction techniques, input and output devices and figuring out what would be right metaphors and paradigms to develop usable and still highly interactive 3D user interfaces. Little consideration is put on how to actually ease the programming of an interactive 3D application. Programming of interactive 3D applications is mainly done with low level facilities like 3D engines and modeling tools. [1]

The 3D engines used are well suited for simplifying rendering operations but they do not include functionality for user interaction. When there is no built-in support for user interaction an application developer has to implement not just event handlers but also the event system from the scratch for every application. This takes plenty of work and might risk reusability, maintainability and scalability of the application.

In 2D application development widget libraries and managed graphics tools are used by daily basis, and when building a user interface the WIMP (windows, icons, menus, pointers) paradigm is widely adapted. However there is no single paradigm to follow in 3D UI development, even if in 3D development concept of widgets exists [2]. Still, the concept of 3D widgets has not been adapted to frameworks used for programming 3D graphics. [3] [1]

In [4] we have introduced a code oriented 3D widget library –WebWidget3D– for WebGL enabled browsers. The library is made with JavaScript and it uses WebGL for rendering. The aim of the library is to make programming interactive 3D applications easier. In this paper we reflect our implementation against previous research and address a missing abstraction level in 3D application development.

The paper is structured as follows. In Section 2 the theoretical background of the paper is addressed and following that in Section 3 the motivation of this work is described. In Section 4 related work is addressed. In Section 5 missing abstraction level of 3D application development is presented. In Section 6 the proposed solution –WebWidget3D– is introduced and in Section 7 the evaluation of the solution is done. Discussion and future work are addressed in Section 8 and final conclusions are drawn in Section 9.

2 Background

2.1 Abstractions of graphics programming

Interactive applications can be programmed in various ways. There are plenty of libraries and programming tools for both 2D and 3D graphics programming and user interface designing. In Figure 1 a rough simplified division of abstraction levels are introduced. Most of the libraries and tools for graphics programming fall into this division.

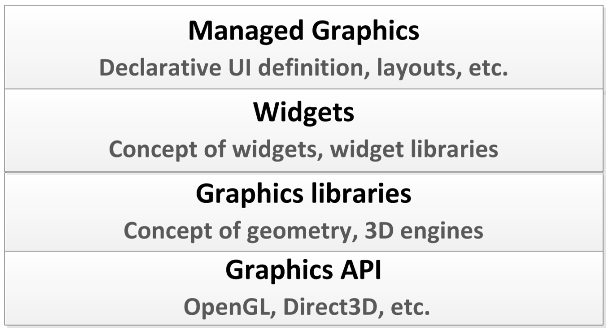

Managed Graphics
Declarative UI definition, layouts, etc.
Widgets
Concept of widgets, widget libraries
Graphics libraries
Concept of geometry, 3D engines
Graphics API
OpenGL, Direct3D, etc.

Fig. 1. Layers of graphics programming abstractions

Graphics API. Graphics API is low level programming interface that abstracts the underlying graphics hardware. OpenGL, WebGL and Direct3D are graphics APIs in this sense. When developing applications in this abstraction level, application developer must have good knowledge on graphics pipeline and graphics programming in general and she might need also some information about the underlying graphics hardware.

Graphics libraries. Graphics libraries in this context covers up libraries that provide basic functionality for drawing primitive shapes such as lines, triangles, points, spheres and custom geometry. Libraries also cover basic functionality for changing materials and applying basic transformations to the geometry drawn. Graphics libraries are considered for simplifying rendering operations and hiding technical details of used graphics API and hardware. Graphics libraries abstractions vary a lot between each other but in this simplification the nominating factor is that none of these libraries provides tools for user interaction. Handling events is entirely programmers' responsibility from determining if an event was addressed to any object and deciding what actions should take place. E.g. most of 3D engines used for game development and 2D drawing libraries and APIs belongs to this abstraction level.

Widgets. In [2] widget is defined as *"an encapsulation of geometry and behavior used to control or display information about application objects"*. For instance, a button is a common widget that is available in almost any widget library. Button has a predefined representation (geometry) and it can be pressed (behavior). In addition to widget sets, widget libraries can also provide tools for building custom widgets. Widgets simplify the programming by combining user interaction and geometry together. Widget libraries usually provide event handling mechanisms or such for user interaction. Application developer needs only to define the action that takes place when a certain event is addressed to a widget. Compared to graphics libraries using widgets makes it easier to separate the user interface code from application logic and thus results more structured applications. The abstraction level of widgets covers up all libraries and other tools that fulfill the definition. Thus this category includes not only full scale widget libraries but also tools that enable binding geometry and interaction together.

Managed graphics. With managed graphics, widgets can be placed in a hosting context, where individual widgets are managed by the context. This simplifies numerous operations such as transformations and scaling, as widgets do not require individual handling. Moreover, it can be helpful to define the basic principles of layouting by e.g. determining the proportion of the window that is used for text instead of defining the exact size of the text field in pixels. Due to the increasing level of abstraction, the programmer has even less control over what takes place upon rendering, but everything takes place in a managed fashion.

2.2 WebGL

There are many different ways to implement 3D graphics inside the browser without plug-in components, which would require separate installation. However, most of them are intended for 2D use and 3D use is a later extension. The only technology that is truly intended for 3D use, and does not require plug-in components, is WebGL, which is true reason we have decided to use it. In the following, we give a brief overview to WebGL, the technology used in the implementation of the WebWidget3D.

WebGL is a standard being developed by Mozilla, Khronos Group, and a consortium of additional companies. The standard is based on OpenGL ES 2.0 [5], and it uses OpenGL shading language GLSL. WebGL runs in the HTML5's canvas element. [6]

For practical purposes, WebGL means that a comprehensive JavaScript API is provided to open up OpenGL programming capabilities to JavaScript programmers, although it is meant for low-level rendering operations. To make it easier and faster to use WebGL, several additional JavaScript frameworks and APIs have been introduced, including Copperlicht[1], C3DL[2], GLGE[3], and three.js[4], to name a few commonly used systems. All these frameworks have their own JavaScript API through which the actual WebGL API is used. In general, the goal of these libraries is to hide the majority of technical details and thus make it simpler to write applications using the framework APIs. Furthermore, these WebGL frameworks provide functions for performing basic 2D and 3D rendering operations such as drawing a rotating cube on the canvas. The more advanced libraries also include functions for performing animations, adding lighting and shadows, calculating the level of detail, collision detection, and so forth. Such rich capabilities enable the creation of more compelling effects relatively easily. However, all these libraries are intended for rendering level operations, and support for more abstract needs of applications are not addressed.

3 Motivation

Widget libraries like WxWidgets and tools for managed graphics are commonly used in 2D application development. However programming of interactive 3D applications is still made with the facilities of rendering level 3D engines and modeling software. 3D models are first made with the 3D modeling tool after that the functionality is programmed using a 3D engine, a physics engine and other application-specific tools. Even if the concept of 3D widgets was first introduced in 1992 by Conner et al [2] the concept is not widely adapted in 3D graphics libraries.

The 3D engines used are well suited for simplifying rendering operations but do not include functionality for user interaction. Binding mouse event handlers or corresponding interaction mechanism to objects using 2D widget libraries or similar tools is pretty simple. Application developer defines the action that takes place in certain event for certain object. However when developing 3D applications using just 3D engines facilities the situation is more complicated. Application developer is responsible also for defining the logic that detects whether an object was e.g. clicked and what actions should take place.

[1] http://www.ambiera.com/copperlicht/
[2] http://www.c3dl.org/
[3] http://www.glge.org/
[4] http://threejs.org/

Detecting whether an object was clicked in 3D environment application developer must typically do the following steps:

1. Determine if the mouse hit a 3D object.
   - Transform the mouse position coordinates into the 3D world's coordinate system.
   - Do ray casting from the mouse's 3D world coordinates.
   - Run collision detection between the ray and objects in 3D world.
2. If a 3D object was hit, deduce the type of the object that was hit.
   - Deduce the action that take place for the object hit when it was clicked with a mouse.

Applying the interaction logic and actions separately for every application allows highly customized and application-specific user interfaces but the code of the user interface is tightly bound to the application and therefore code reuse can be difficult. Also designing a highly interactive 3D application can be complicated when support of higher level of abstraction concepts are missing. Designing an application where every object can be interacted with same manners (e.g. clicked or dragged with mouse) is quite simple even if we do not have any support for interaction. However when designing complicated applications, where different objects can have different actions and where actions can be dependent from other actions, the designing gets quickly very complex. In worst case, developing interactive 3D applications with low abstraction level tools result to spaghetti code that does not scale up and is hard to maintain. [2] [7] [8]

When designing a 3D widget library there are several things that have to be noted. First of all, interactive 3D applications are not similar to most of the 2D applications. 3D is most used for immersive games and virtual reality applications which are graphically intense but also for data visualization. The look and feel of a 3D application is usually dependent on the application. There is no standardized 3D user interface paradigm like WIMP to follow and there may never be [1]. When in 2D environments WIMP metaphor is proven to be effective, it is not applicable in 3D environments. Applying WIMP metaphor to 3D environments as is would limit look and feel of applications and it might restrict degrees of freedom that can be applied on 3D objects in 3D space [1] [3] [7].

In 2D user interface design separation of user interface definition and application logic is considered preferable [7] [2]. However in no WIMP interfaces which 3D user interfaces mainly are it is stated that tight separation between application logic code and user interface code is not necessarily desirable because it might limit interaction techniques that can be used in the interface [3] [7] [2]. Therefore the development of 3D applications is significantly different from 2D applications.

4 Related work

The research of interactive 3D applications has mainly focused on developing interaction techniques, input and output devices and figuring out what would be right metaphors and paradigms to develop usable and still highly interactive 3D user interfaces. Little consideration is put on how to actually ease the programming of an interactive 3D application. [1]

The basic concept of 3D widgets as first-class objects is presented in [2]. We agree on the introduced approach at the principal level. However the research is focused on solving problems in 3D interaction and constructing 3D widgets whereas our work's aim is to study how programming of 3D applications could be made easier. The 3D widget system UGA, built as a research artifact, introduces a new scripting language for 3D widgets construction and provides toolkit for constructing widgets [9].

In [7] a concept-oriented design (COD) approach to 3D user interface development and Chasm tool as an implementation of the approach are introduced. Chasm is an executable user interface description language (UIDL) for 3D user interface development. The presented approach reduces the complexity of 3D user interface development. Chasm has been shown to be useful for creating complex full scale 3D system user interfaces. However it does not targets to simplify the programming of 3D applications.

Declarative technologies are commonly used for creating virtual environments. Virtual Reality Modeling Language, VRML, is a file format for representing interactive 3D graphics in web. Concept of VRML was introduced in 1994 in [10]. In VRML the 3D world is defined in declarative manner and actions, such as user interaction and animations, are scripted. Most of 3D modeling software can export models in VRML format. X3D [11] is a XMl based successor of VRML. To view VRML and X3D scenes, additional software, i.e. BS Contact, is needed.

X3DOM [12] is a webGL based implementation of X3D which can run in web browser without plug-in components. The goal of the X3DOM is to embed 3D content into DOM and thus enable the use of 3D content in web applications in same manner as 2D content. XML3D [13] is another implementation of declarative 3D graphics based on WebGL. XML3D also integrates 3D content into DOM but in difference to X3DOM, XML3D is not based on X3D. X3DOM and XML3D are in development state at the moment.

CONTIGRA is X3D based widget library, which offers high level of separation between 3D widget declaration and application logic. CONTRIGRA also has high support on component reuse and it provides tools for widget distribution. [14] [15] [8]

BEHAVIOR3D, introduced in [16], is a X3D framework for designing 3D graphics behavior. It is used for implementing and designing behavior to 3D objects. BEHAVIOR3D is designed to be used with CONTIGRA. While in X3D and in CONTIGRA behavior of 3D objects are implemented using scripts, with BEHAVIOR3D behavior can be implemented by declarative way using XML.

A benefit for VRML and X3D is that those are widely used, established, technologies for creating virtual environments. X3DOM and XML3D are also promising new technologies in development of 3D web applications. CONTIGRA and BEHAVIOR3D introduce a strong declarative toolset for 3D widget creation and distribution. However all these tools and technologies stand for declarative approach to 3D development while our interest is on code oriented approach.

While declarative technologies are used for creating virtual environments, as well code oriented approaches, e.g. 3D engines and other graphics libraries, are widely used. 3D engines are mainly focused on abstracting rendering operations and support for user interaction is lacking. We have reviewed a number of 3D engines including Unity[5], OGRE[6], three.js[7], Copperlicht[8], C3DL[9], GLGE[10] and SpiderGL[11]. Unlike X3D or X3DOM, none of these 3D engines have support for binding user interaction directly to 3D objects which makes programming interactive 3D applications more difficult. Our interest in particular is to develop the code oriented approach further by extending 3D engines facilities to support concept of 3D widgets.

5 Missing Abstraction

In [14] it is stated that X3D and VRML are declarative counterpart for 3D engines which are code oriented. This means that in [14] the abstraction level of X3D and 3D engines is stated to be same. The definition of widget, introduced earlier in Section 2.1 is: *"Widget is an encapsulation of geometry and behavior used to control or display information about application objects"* [2]. Hence X3D and VRML have support for binding user interaction to 3D objects and 3D engines do not have that support, abstraction level of these tools are not same. 3D engines operates in *Graphics libraries* abstraction level introduced previously in Figure 1 while X3D and VRML operates actually in *Widgets* abstraction level albeit those are not exactly thought to be widget libraries.

One level of abstraction is missing. There is no support for code oriented way to do 3D widgets. When figuring out which tools to use for implementing interactive 3D applications, the application developer can choose between code oriented and declarative approaches. If she chooses code oriented approach, there is no support for interaction. We claim that this is one of the reasons why programming interactive 3D applications is still difficult.

Declarative XML based tool always has to have a program that executes it [17]. If programming tools do not support the same abstraction level concepts that the intended declarative tool does, development of the underlying software is more difficult and can cause problems if the software needs to be ported to other technology.

In [8] difficulties in making the widget toolkit portable for other 3D technologies were reported. We state that these difficulties were due bypassing code

[5] http://unity3d.com/
[6] http://www.ogre3d.org/
[7] http://threejs.org/
[8] http://www.ambiera.com/copperlicht/
[9] http://www.c3dl.org/
[10] http://www.glge.org/
[11] http://spidergl.org/

oriented 3D widget abstraction totally and undermining the differences between different 3D technologies. Use of XML in 3D UI tools are argued for being independent from the underlying technologies and it's argued that therefore the 3D UI tool is easy to port to different 3D technologies. Still 3D technologies underneath the XML might not be compatible at all. Hence XML do not actually give any real advantage compared to other programming languages on portability issues. Programming a whole 3D engine and widget concept from scratch to port the high level UI system for a different 3D technology is a lot of work. If the higher abstraction levels were built on top of the existing lower abstraction level facilities without bypassing abstraction levels, the gap between 3D technologies could be reduced and portability made easier.

In Figure 2 simplification of current relations between declarative and code oriented approaches based on our observations is introduced. In Figure 3 our suggestion of relations is introduced.

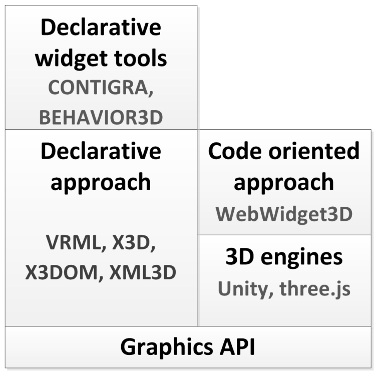


Fig. 2. Relations of declarative and code oriented approaches.

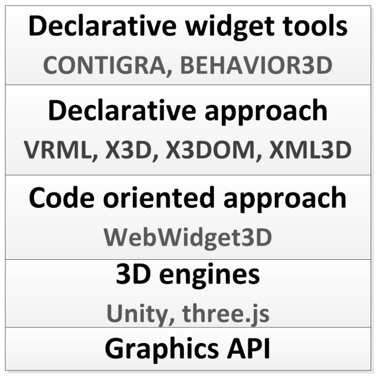


Fig. 3. Suggested relations between different approaches.

From the Figure 2 it can be deducted that at the moment code oriented approach and declarative approach is tightly separated. The only common nominator is the low level graphics API. However the declarative approach does not have to be separated from code oriented approach. It can be built upon already existing code oriented abstractions as is shown in Figure 3.

6 WebWidget3D

6.1 User interaction

The WebWidget3D library [4] aims at making developing of interactive 3D applications to web easier. The library supports input devices that generate DOM events. At this point these input devices are commonly mouse, keyboard and touch screen.

In traditional web applications, DOM event handlers are usually bound to HTML elements, such as buttons and text boxes, for instance. In a WebGL based 3D world, however, binding events to a certain 3D widget is more complicated. The whole 3D world is rendered into a single canvas element. Therefore, when using the traditional approach, DOM events can be bind to the canvas but not directly to 3D objects inside the canvas, as one would most commonly prefer. Instead, handling events in a WebGL based 3D world is more complicated than in ordinary web applications.

To handle events in WebGL 3D world the first part is to bind the wanted handlers to the canvas element. When a mouse event is triggered we need to know the point where the mouse hits to determine if the mouse event hits a 3D object or not. Consequently, we need to perform the steps of mouse click detection described before in Section 3. Interaction can be done also with other than pointing devices, e.g. with keyboard. For keyboard event handling, main event handlers an additional logic for deducing the right actions is also needed.

The WebWidget3D provides an event system which enables the application developer to attach event handlers directly to the 3D objects trough an API that is similar to the API for binding DOM event handlers to HTML elements. The application developer can also define her own events, trigger events and pass events to designated widgets using the event system.

In WebWidget3D there are three main types of events which are supported:

1. Pointing device events
2. Keyboard events
3. Custom events / messages

Pointing device event objects must contain coordinate data in window coordinate system. The WebWidget3D event system handles the ray casting and passing the event to right widgets. Keyboard events are passed to all widgets that are focused and have the listener for the event. Custom events are events that are not pointing device events or keyboard events. Also other DOM events than pointing device events and keyboard events are custom events. Custom events are passed to all widgets that have the listener for the event. Custom events can be used e.g. for passing messages between widgets. In WebWidget3D all types of events can also be bind to the application. The application's event handler is called always when the event occurs. The amount of event handlers for one event for one object is not limited.

To simplify application developers work WebWidget3D introduces also some predefined controls that can be applied to widgets. These predefined controls are a set of event handlers that perform e.g. object drag functionality. At the moment WebWidget3D has only two predefined controls which are roll- and drag controls. With roll control a widget can be rotated around its x- and y-axis by mouse. With drag control widget can be dragged around the 3D world with mouse. The drag is done always coaxial to the camera so that the object moves as in 2D space. This is most intuitive drag control when the input device is regular mouse. More controls are under consideration but not yet implemented.

The controls are designed for regular 2D mouse and keyboard which produces some complexity for combining controls. Drag and roll are done with same mouse gestures and if there is no information from where system could deduce which action the user would prefer, both actions take place simultaneously. For combining controls application developer can define parameters for example to roll only if shift key is pressed or disabling unwanted controls in certain situations.

6.2 Concept of 3D widgets

The WebWidget3D library applies the concept of 3D widgets by providing widget building blocks that can be instantiated, specialized, refined and composite to create a desired 3D widget. The widget building blocks have different roles and responsibilities in the design. Common for all building blocks is the capability of receiving events and the predefined controls can be applied to them. None of the building blocks include a concrete graphical representation. Graphical representation of a widget can be designed with modeling tools to ensure desired look that is suitable for the application.

The building blocks offered are: *Basic*, *Text*, *Group* and *CameraGroup*. Basic is the base for all widgets. Basic can receive events and a 3D model can be attached to it. It has no special features and it is the simplest building block in the set. Text includes simple string handling functionality so that it can be used to store dynamic text, e.g. users input from keyboard. Group can host child components of any type. Group provides also some utilities to manage its child components i.e. focusing and hiding all children. Also rotations, translations and changes in visibility are always propagated to Group's children. With CameraGroup widgets can be attached to camera so that the widgets orientation and distance to the camera doesn't change when the camera is moved.

These widget building blocks and the event system form the core of the WebWidget3D. The core of the library is designed so that it can be used with any JavaScript 3D engine. There is a specialized adapter component that is used between the 3D engine and the WebWidget3D core. The proof-of-concept implementation done uses three.js 3D engine for rendering. WebWidget3D does not hide the API of 3D engine from application developer so application developer can use all the functionality and visual effects 3D engine has to offer.

6.3 Predefined Widgets

To prove that WebWidget3D can be used for creating concrete reusable 3D widgets, a set of ready-to-use widgets were made. These widgets are built using the widget building blocks described previously and facilities of three.js 3D engine. The widgets follow closely WIMP paradigm and are designed mainly for Lively3D 3D desktop environment, which is introduced in [4].

Grid Window and *Grid Icon* are used to form a *grid widget.* The grid size is dynamically derived from the amount of its children and grid icons are automatically placed to first free slot on the grid when created. Grid Window can be rotated by its axis using roll controls. Also drag controls and custom controls can be applied to Grid Window.

Titled Window is similar to common WIMP windows. Its representation is a quadrangle and it has a title bar at the top of the window and a close button at the upper right corner. The content of the window can be anything that can be rendered into texture. Titled Window has drag controls built-in as GridWindow has roll controls.

Dialog is a widget that can be used to form dialogs or forms. *Menu* widget has multiple choice buttons and a description text.

All of the widgets introduced have been used in Lively3D 3D desktop environment. In Figure 4 a screenshot from Lively3D environment is presented. Most of the widgets described above are present in the figure.

Grid widget is used successfully also in picture explorer application where the preview images are rendered to the icons and clicking the icon will open the full sized image. Titled window is also used in some example applications to host video content.

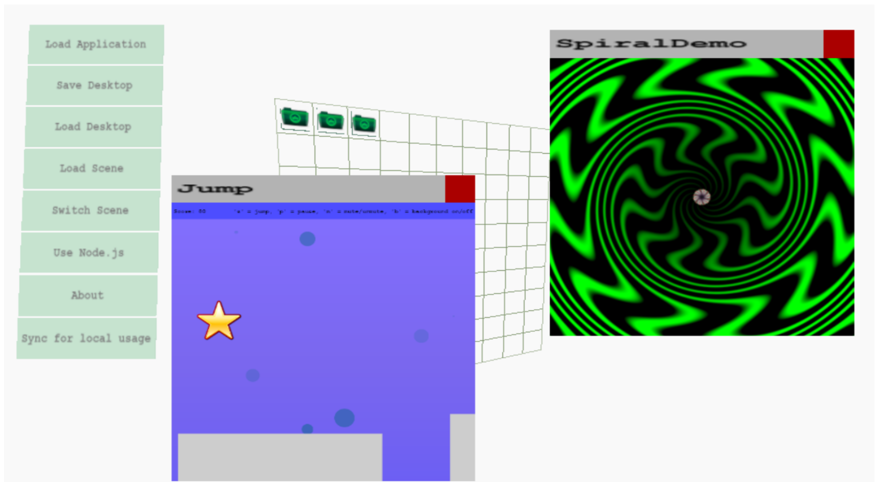


Fig. 4. Screenshot from Lively3D

Predefined widgets are handy when building applications like Lively3D or when implementing small demonstrations. However, when designing graphically highly intensive and immersive applications, using predefined widgets might not be desired. The application designer can design her own set of widgets that she uses in that particular application. Widgets designed for one purpose might not be suitable for other applications even if the code reuse is possible.

Separating widget definition from application logic makes the application code structured and easier to maintain [14]. The user interaction part can be programmed either as a part of the application logic or as a part of the widget definition. The application developer can choose the level of separation between the user interface and the application logic. If the application logic and widget definition were forced to be separated it might limit the possibilities of interaction mechanisms used [7].

7 Evaluation of WebWidget3D

In our previous work we focused on introducing the architecture and implementation of WebWidget3D. On this paper our aim is to evaluate the work towards the previous research pointed out in Section 4.

In Table 1 wanted features for 3D widget library are introduced. Three.js, X3D and Widget3D are compared against these features. In the table *x* means that feature is fulfilled and *?* means that feature is not tested. Blank means that technology does not fulfill requirements.

Table 1. Desired features of 3D widget library

Feature	three.js	X3D	WebWidget3D	CONTIGRA
Combines the 3D objects' geometry and behavior together [2] [14]		x	x	x
Does not limit the look and feel of the application [1] [7]	x		x	
Supports code reuse [14] [2]			x	x
Does not limit the degrees of freedom that can be used in input devices and applied to widgets [1] [7] [3]	x	x	?	x
Offers predefined widgets [14]			x	x
Independent from used 3D technology [8]		x		?

WebWidget3D succeeds to fulfill four features from six listed in Table 1. The library combines the 3D objects' geometry and behavior together forming 3D widgets. Also CONTIGRA and X3D fulfills this feature. Three.js however operates in lower abstraction level and hence does not support the concept of 3D widgets.

WebWidget3D doesn't hide the 3D engines API so application developer can use all the features provided by the 3D engine. The application developer can design her own widget set and controls using the library. Thus webWidget3D does not limit the look and feel of application. WebWidget3D enables code reuse but application logic and user interface code can also be combined if it is necessary.

In X3D available visual effects are dependent from the implementation of X3D used. This is why using X3D might limit capabilities of underlying 3D technology and also the look of the application. CONTIGRA separates the user interface definition from application logic tightly and thus supports code reuse but might limit the look and feel of the application [7] [3]. CONTIGRA is also based on X3D.

WebWidget3D does not necessary limit the degrees of freedom that can be used in input devices but at the moment library is used and tested only with standard 2D input devices which are mouse and keyboard. The library can be used with other kind of input devices as data glove or 3D mouse if the device can fire DOM events and the input of the device can be read from the DOM event object in the format which WebWidget3D's event system can understand.

Using WebWidget3D with other than web technologies is not possible without lots of work because the built-in event system is based on DOM. Nevertheless the concept of 3D widget library that can use already existing 3D engine for rendering is applicable in other environments too. At the moment WebWidget3D does not fulfill the requirement of being independent from 3D technology used.

Also three.js is designed on top of WebGL and web technologies so it is not portable to other 3D technologies as is. However, corresponding 3D engines exists on other platforms too. X3D is portable to other 3D technologies as X3DOM, webGL based implementation of X3D shows. However porting X3D can also demand lots of work. CONTIGRA is designed so that it can be ported to other technologies than X3D. According to [8] this feature was never tested because it required more work than planned.

WebWidget3D's goal is to make programming of interactive 3D applications easier. It is not a 3D widget design tool or 3D UI creator. At this state Web-Widget3D is just a research artifact that needs to be refined. However even in its current state the WebWidget3D can be used for programming interactive 3D applications. In [4] we made measurements with WebWidget3D that clearly advocates the widget approach over the use of only 3D engines facilities. Concept of 3D widgets is powerful even if it wouldn't result high code and widget reuse between applications.

8 Discussion and Future Work

In [1] it is stated that in 3D user interface research the discussion is kept on a high abstraction level on purpose because of instability of underlying 3D technology. This might be one of the reasons why code oriented approach to 3D widgets is not studied. On 2D application development the programming tools and also UI tools are highly developed and of course it would be fantastic to have 3D application development in the same level. When approaching the issue from the higher abstraction levels we might think that the lower levels actually exists and forget to consider gaps between used technologies.

Also 3D engines have developed a lot in a couple of years. Before programming 3D applications was even harder than it is now. Application developers who are used to develop applications with low abstraction level facilities might not see why the widget abstraction is needed.

For a future work we are building more adapters to WebWidget3D to validate its use with different 3D engines. Also implementing similar architecture to desktop environments is planned. Portability issues of 3D tools are studied further. Code camp where students test programming 3D applications with WebWidget3D is under consideration. Also further research on which tools are most commonly used in 3D development will be done.

9 Conclusions

In this paper we have addressed a missing abstraction of 3D programming and made literature based evaluation of WebWidget3D.

It is evident that for developing complex interactive 3D applications better programming tools are needed. The widget level of abstraction programming tools are still lacking and we state that this is one of the reasons why programming of interactive 3D applications is still difficult.

The research about 3D environments and tools has long been considered on the higher abstraction level issues undermining the importance of decent programming tools for 3D applications. This has led the 3D application development into a state where application developer must choose between declarative 3D tools and 3D graphics libraries, which have no support for interaction. This is a generic area where improvement is expected to happen.

References

1. Bowman, D., Kruijff, A., Jr., J.L.: 3D User Interfaces: Theory and Practice. Addison Wesley (2005)
2. Conner, B.D., Snibbe, S.S., Herndon, K.P., Robbins, D.C., Zeleznik, R.C., Van Dam, A.: Three-dimensional widgets. In: Proceedings of the 1992 symposium on Interactive 3D graphics, ACM (1992) 183–188
3. Green, M., Jacob, R.: SIGGRAPH'90 Workshop report: software architectures and metaphors for non-WIMP user interfaces. ACM SIGGRAPH Computer Graphics **25** (1991) 229–235
4. Mattila, A.L., Mikkonen, T.: Designing a 3d widget library for webgl enabled browsers. In: In proceedings of the 28th Symposium On Applied Computing. Volume 1., ACM (March 2013) 757–760
5. : OpenGL ES Common Profile Specification. Technical report, Khronos Group (2010) http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf/.
6. : WebGL Specification. Technical report, Khronos Group (2011) http://www.khronos.org/registry/webgl/specs/1.0/.
7. Wingrave, C.A., Laviola Jr, J.J., Bowman, D.A.: A natural, tiered and executable uidl for 3d user interfaces based on concept-oriented design. ACM Transactions on Computer-Human Interaction (TOCHI) **16**(4) (2009) 21

8. Dachselt, R., Hinz, M., Meißner, K.: Contigra: an xml-based architecture for component-oriented 3d applications. In: Proceedings of the seventh international conference on 3D Web technology, ACM (2002) 155–163
9. Zeleznik, R.C., Herndon, K.P., Robbins, D.C., Huang, N., Meyer, T., Parker, N., Hughes, J.F.: An interactive 3d toolkit for constructing 3d widgets. In: Proceedings of the 20th annual conference on Computer graphics and interactive techniques, ACM (1993) 81–84
10. Raggett, D., et al.: Extending www to support platform independent virtual reality. In: Proc. Internet Society/European Networking. (1994) 242
11. Brutzman, D., Daly, L.: X3D: extensible 3D graphics for Web authors. Morgan Kaufmann (2010)
12. Behr, J., Jung, Y., Drevensek, T., Aderhold, A.: Dynamic and interactive aspects of x3dom. In: Proceedings of the 16th International Conference on 3D Web Technology, ACM (2011) 81–87
13. Sons, K., Klein, F., Rubinstein, D., Byelozyorov, S., Slusallek, P.: Xml3d: interactive 3d graphics for the web. In: Proceedings of the 15th International Conference on Web 3D Technology, ACM (2010) 175–184
14. Dachselt, R.: Contigra towards a document-based approach to 3d components. In: Workshop'Structured Design of Virtual Environments and 3D-Components' at the ACM Web3D 2001 Symposium, Citeseer (2001)
15. Dachselt, R.: Contigra: A high-level xml-based approach to interactive 3d components. In: SIGGRAPH 2001 Conference Abstracts and Applications. Volume 163. (2001)
16. Dachselt, R., Rukzio, E.: Behavior3d: an xml-based framework for 3d graphics behavior. In: Proceedings of the eighth international conference on 3D Web technology, ACM (2003) 101–ff
17. : Extensible Markup Language (XML) 1.1 (Second Edition). Technical report, W3C (2006) http://www.w3.org/TR/2006/REC-xml11-20060816/.

The Browser as a Host Environment for Visually Rich Applications

Jari-Pekka Voutilainen and Tommi Mikkonen

Department of Pervasive Computing, Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland
{jari-pekka.voutilainen,tommi.mikkonen}@tut.fi

Abstract. The World Wide Web has rapidly evolved from a simple document browsing and distribution environment into a rich software platform where desktop-style applications are treated as first class citizens. Despite the technical difficulties and limitations, it is not unusual for complex applications to use the web as their only platform, with no traditional installable application for the desktop environment – the system is simply accessed via a web page that is downloaded inside the browser. With the recent standardization efforts, such as HTML5 in particular, applications are increasingly being supported by the facilities of the browser. In this paper, we demonstrate the new facilities of the web as an visualization tool, going beyond what is expected of browser based applications. In particular, we demonstrate that with mashup technologies, which enable combining already existing content from various sites into an integrated experience, the new graphics facilities unleashes unforeseen potential for visualizations.

Keywords: Web applications, visualization, scene graph, window management

1 Introduction

Over the past years, the World Wide Web has evolved from a simple document browsing and distribution environment into a rich software platform where desktop-style applications are increasingly often treated as first class citizens. However, the document-centric origins of the Web are still visible in many areas, though, and traditionally it has been difficult to compose truly interactive web applications without using plug-in components or browser extensions such as Adobe Flash or Microsoft Silverlight, to name two examples. Despite the technical difficulties and limitations, it is not unusual for complex applications to use the web as their only platform, with no traditional installable application for the desktop environment – the system is simply accessed via a web page that is downloaded inside the browser, whose runtime resources are then used by the application. We believe that the transition of applications from the desktop computer to the web has only started, and the variety, number, and importance of web applications will be constantly rising during the next several years to come.

In comparison to desktop applications, the benefits of web applications are many. Web applications are easy to adopt, because they need neither installation nor updating - one simply enters the URL into the browser and the latest version is always run. Furthermore, web applications are easy and cheap to publish and maintain; there is no need for intermediates like shops or distributors. Furthermore, in comparison to conventional desktop applications, web applications have a whole new set of features available, like online collaboration, user created content, shared data, and distributed workspace. Finally, with the whole content of the web acting as the data repository, the new application development opportunities, unleashed by the newly introduced facilities of the web technologies that make the browser increasingly capable platform for running interactive applications, are increasing the potential of the web as an application platform.

In this paper, we demonstrate the new facilities of the web as an information visualization tool, going beyond what is expected of browser based applications. In particular, we demonstrate that together with mashup technologies, which enable combining already existing content from various sites into an integrated, usually more compelling experience, the new graphics facilities results in unforeseen potential for visualization of conceptual data.

The rest of the paper is structured as follows. In Section 2, we discuss the evolution of the web and the main phases that can be identified in the process, and briefly address two important web standards - HTML5 and WebGL - and their role in the development of new types of web applications, building on already available resources. In Section 3, we introduce our technical contribution, a host environment that is capable of intregating multiple applications within single 3D-scene and visualize the environment in three different ways. In Section 4, we discuss the lessons learned from the design and experimentation of the composed system. Finally, in Section 5 we draw some conclusions and directions for future work.

2 Background

The World Wide Web has undergone a number of evolutionary phases. Initially, web pages were little more than simple textual documents with limited user interaction capabilities. Soon, graphics support and form-based data entry were added. Gradually, with the introduction of DHTML, it became possible to create increasingly interactive web pages with built-in support for advanced graphics and animation. Today, the browser is increasingly used as a platform for real applications, with services such as Google Docs paving the way towards more complex systems. One way to categorize the evaluation of the web is presented in the following, based on [4].

2.1 Evolution of the Web

In the first phase, web pages were truly pages, that is, page-structured documents primarily including text with some interspersed static images, without animation or any interactive content. Navigation between pages was based on hyperlinks, and a new web page was fully loaded from the web server each time the user clicked on a link. Some pages were presented as forms, with simple textual fields and the possibility to use basic widgets such as buttons, radio buttons or pull-down menus.

In the second phase, web pages became increasingly interactive, with animated graphics and plug-in components that allowed richer content to be displayed. This phase coincided with the commercial takeoff of the Web, when companies realized that they could create commercially valuable web sites by displaying advertisements or by selling merchandise or services over the Web. Navigation was no longer based solely on links, and communication between the browser and the server became increasingly advanced. The JavaScript scripting language, introduced in Netscape Navigator version 2.0B in December 1995, made it possible to build animated, interactive content more easily. The use of plug-in components such as Flash, Quicktime, RealPlayer and Shockwave spread rapidly, allowing advanced animations, movie clips and audio tracks to be inserted in web pages. In this phase, the Web started moving in directions that were unforeseen by its designers, with web sites behaving more like multimedia presentations rather than conventional pages. However, these systems are commonly based with proprietary presentations, and linking information from different origins was still difficult. Consequently, creating a mashup system, where data from a set of available services was used as basis for an animation, for example, remained superfluously complex.

Today, we are in the middle of another major evolutionary step towards desktop-style web applications, also known as Rich Internet Applications or simply as web applications. The technologies intended for the creation of such applications are also often referred to collectively as "Web 2.0" technologies. Fundamentally, Web 2.0 technologies combine two important characteristics or features: collaboration and interaction. By collaboration, we refer to the "social" aspects that allow a vast number of people to collaborate and share the same data, applications and services over the Web. However, an equally important, but publicly less noted aspect of Web 2.0 technologies is interaction. Web 2.0 technologies make it possible to build web sites that behave much like desktop applications, for example, by allowing web pages to be updated one user interface element at a time, rather than requiring the entire page to be updated each time something changes. Web 2.0 systems often eschew link-based navigation and utilize direct manipulation techniques familiar from desktop-style applications.

We expect that as more and more data becomes available online, the capabilities of the browser will be increasingly often harnessed to filter and further process the data into a form that can be more easily consumed. In this context, two recent initiatives form an important perspective. These are the open web, perhaps best manifested in Mozilla Manifesto[1], which centers around the idea that the web that is a global public resource that must remain open, accessible,

[1] http://www.mozilla.org/about/manifesto.html

interoperable and secure, and open data, which according to Wikipedia[2], builds on the idea that certain data should be freely available to everyone to use and republish as they wish, without restrictions from copyrights, patents, or other mechanisms of control.

To support the above initiatives, the need to use plugins is being seriously challenged by two recently introduced technologies, HTML5 and WebGL, as already pointed out in [5]. These new technologies provide support for creating desktop-like applications that run inside the browser (HTML5) and enable direct access to graphics facilities from web pages (WebGL). This, together with already well-known techniques for mashupping, are paving the way towards the next generation of web applications, with increasing capabilities for modeling and visualizing data and conceptual information.

2.2 HTML5

The forthcoming HTML5 standard[3] complements the capabilities of the existing HTML standard with numerous new features. Although HTML5 is a general-purpose web standard, many of the new features are aimed squarely at making the Web a better place for desktop-style web applications. There are numerous additions when compared to the earlier versions of the HTML specification. To begin with, the new standard will extend the set of available markup tags with important new elements. These new elements make it possible, e.g., to embed audio and video directly into web pages. This will eliminate the need to use plug-in components such as Flash for such types of media. The HTML5 standard will also introduce various new interfaces and APIs that will be available for JavaScript applications. Some of the new features are listed in the following, based on [1].

- Browser history management. In order to manage browsing history in web applications, the traditional mechanism is clearly inadequate. HTML5 introduces an API that can be used for manipulating the history.
- Canvas element and API. A procedural (as opposed to declarative) 2D graphics API for defining shapes and bitmaps that are rendered directly in the web browser.
- ContentEditable attribute. The attribute makes it possible to create editable web documents.
- Drag-and-drop. Drag and drop capabilities that are commonly needed in numerous applications.
- Geolocation. The Geolocation API defines a set of operations and data elements for accessing geographical location (such as GPS positioning) information.
- Indexed hierarchical key-value store (formerly WebSimpleDB).
- MIME type and protocol handler registration.

[2] http://en.wikipedia.org/wiki/Open_data
[3] http://www.w3.org/TR/html5/

– Microdata. The goal of microdata is to provide a straightforward way to embed semantic information into HTML documents.
– Offline storage database. The offline storage database will enable applications to access their data even when an online connection is not available[4].
– Timed media playback.

In addition, HTML5 is an enabler for a number of other specifications that further enrich the browser as the platform. Of particular interest in our work is the WebGL specification, which will play a major role by introducing powerful, hardware accelerated graphics in web applications.

2.3 WebGL

WebGL[5] is a cross-platform web standard for hardware accelerated 3D graphics API developed by Mozilla, Khronos Group, and a consortium of additional companies including Apple, Google and Opera. The main feature that WebGL brings to the Web is the ability to display 3D graphics natively in the web browser without any plug-in components. WebGL is based on OpenGL ES 2.0[6], and it uses the OpenGL shading language GLSL. WebGL runs in the HTML5's canvas element, and WebGL data is generally accessible through the web browser's Document Object Model (DOM) interface. A comprehensive JavaScript API is provided to open up OpenGL programming capabilities to JavaScript programmers.

The possibility to display 3D graphics natively in a web browser is one of the most exciting things happening on the Web for quite a while. Displaying 3D graphics content on the Web has been possible even in the past with APIs such as Flash, O3D, VRML and X3D, but only with certain browsers or if the necessary browser plug-in components has been installed explicitly. However, with WebGL the 3D capabilities are integrated directly in the web browser, meaning that 3D content can run smoothly in any standard-compliant browser without application installation or additional components.

The present WebGL specification was released on March 1, 2013 and WebGL support has already been implemented and included in the current versions of Apple Safari, Mozilla Firefox and Google Chrome, with Microsoft Internet Explorer being the only major browser not offering any support. The work continues towards the next version of the specification and this draft is widely supported in forthcoming versions of these browsers.

In combination with HTML5 and other web standards, the web browser will have support for web sockets, video streaming, audio, CSS, SVG, web workers, file handling, fonts and many other features. With all these capabilities it is relatively simple to port existing OpenGL applications into the web browser environment. For example, game engine Unreal Engine 3 has been ported with Emscripten[7] to the browser[8] using WebGL and the new features supported by

[4] http://www.w3.org/TR/offline-webapps/
[5] http://www.khronos.org/webgl/
[6] http://www.khronos.org/opengles
[7] https://github.com/kripken/emscripten/wiki
[8] http://www.unrealengine.com/html5/

HTML5. We take this as early evidence that WebGL is powerful enough to challenge the dominance of binary gaming software.

As a technical detail, it is important to notice that the WebGL API is implemented at a lower level compared to the equivalent OpenGL APIs. This increases the software developers' burden as they have to implement some commonly used OpenGL functionality themselves. To make it easier and faster to use WebGL, several additional JavaScript frameworks and APIs have been introduced, including Three.js[9], Copperlicht[10], GLGE[11], SceneJS[12], and SpiderGL[13]. Such frameworks introduce their own JavaScript API through which the lower-level WebGL API is used. The goal of these libraries is to hide the majority of technical details and thus make it simpler to write applications using the framework APIs. Furthermore, these WebGL frameworks provide functions for performing basic 2D and 3D rendering operations such as drawing a rotating cube on the canvas. The more advanced libraries also have functions for performing animations, adding lighting and shadows, calculating the level of detail, collision detection, object selection, and so forth.

3 Lively3D: Host environment for web applications

In this section, we introduce a proof-of-concept implementation designed to demonstrate the new facilities of the browser as a platform. The goal of the experiment was to create a 3D environment in which applications of different kind – including data processing, visualization, and interactive applications in particular – can be embedded as separate elements within a single environment. Furthermore, the design is based on using facilities that are commonly used in the web already, implying that to a large extent it is possible to reuse already existing content in the system.

3.1 Overview

Web app, by simple definition[14], is an application utilizing web and [web] browser technologies to accomplish one or more tasks over a network, typically through [web] browser. Canvas application is a subset of web app, which uses a single canvas html-element as graphical interface.

Lively3D[15] is a framework, where embedded canvas applications are displayed in a three dimensional windowing environment. Individual applications embedded in the system can thus be composed using the Canvas API, offered

[9] http://threejs.org/
[10] http://www.ambiera.com/copperlicht/
[11] http://www.glge.org/
[12] http://scenejs.org/
[13] http://spidergl.org/
[14] http://web.appstorm.net/general/opinion/what-is-a-web-app-heres-our-definition/
[15] http://lively3d.cs.tut.fi/

by HTML5. In general, this enables the creation of graphically rich small applications that are capable of interacting with the user in a desktop like fashion.

The Lively3D framework itself is based on GLGE[16], a WebGL library by Paul Brunt, which abstracts numerous implementation details of WebGL from the developer. Embedding the applications to the framework was designed in such a way that the developer of a canvas application needs to implement minimal interfaces towards the Lively3D system in order to integrate the application within the environment. Existing canvas applications are easily converted to Lively3D app by wrapping the existing code to the Lively3D interfaces.

In addition to the applications, the 3D environment that displays the applications can be redefined using Lively3D interfaces. The applications and different 3D environments are deployed in a shared Dropbox folder, so that multiple developers can collaborate in implementing applications and environments without constantly updating the files on the server hosting Lively3D.

Lively3D is implemented as Single-Page Application (SPA) where the whole application is loaded with a single page load. This provides the user interface and the basic mechanics of 3D enviroments. The design of Lively3D was considerably affected by the browser security model, which limits the possibilities of resource usage. The security model denies access both to the local file system and external resources in different domain with its Same-origin policy[17]. The policy is upheld in Lively3D with server-side proxies, so that the browser sees all the content in same domain. The main components of the system are illustrated in Figure 1. All components are designed with easy-to-use interfaces and require minimal knowledge of inner working of the framework.

Applications and 3D scenes are developed in JavaScript using Lively3D API, deployed to Dropbox using the official Dropbox client, and downloaded into Lively3D through PHP or Node.js proxies, depending on the situation. The Lively3D API provides resource loaders, which enable deployment of application and 3D-scene specific resources to the Dropbox so that complete applications and 3D scenes can be downloaded through the server hosting Lively3D, thus in essence circumventing browser security restrictions.

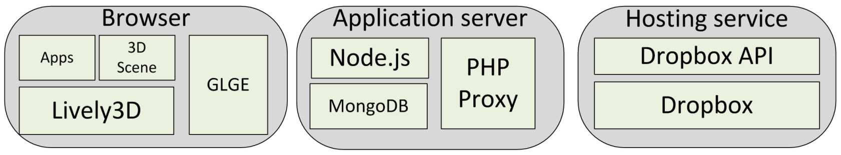


Fig. 1. Structure of the Lively3D framework

When a new 3D scene is designed and implemented, the developer has to define the essential functions that are called by the Lively3D environment, sim-

[16] http://www.glge.org/

[17] http://www.w3.org/Security/wiki/Same_Origin_Policy

ilarly to many other graphical user interface frameworks. The functions enable redefining mouse interaction, the creation of a 3D object in the GLGE system that represents the application, and automatic updates of the scene between frames. Additionally, the initial state of the scene is defined in GLGE's XML format, which can be generated with 3D modeling software, like Blender (http://www.blender.org/) for example.

3.2 Lively3D apps

A Lively3D app consists of canvas application and its data structures in Lively3D host environment. Usable existing web apps are limited to canvas applications, because Lively3D is implemented in WebGL and the WebGL specification permits the use of canvas, image and video html-elements as the only source for textures within the 3D-environment. Most of the data structures are provided by Lively3D, but some conventions must be followed when converting existing canvas application to Lively3D app.

Since web apps are usually developed with expectancy that the app will be the only app in web page, the app structure can be pretty much anything the developer desires. But since Lively3D is implemented in Single Page Application paradigm, Lively3D apps are separated from each other with simulated namespaces as much as the browser model permits. To achieve this, the canvas application must have clearly separated initialization code. Additionally all the browser elements the app uses, must be created dynamically with a single canvas-element functioning as the only graphical element of the application. To mitigate these restrictions Lively3D offers API for canvas applications, which is presented in figure 2. In the following, we briefly list the most important features of the API.

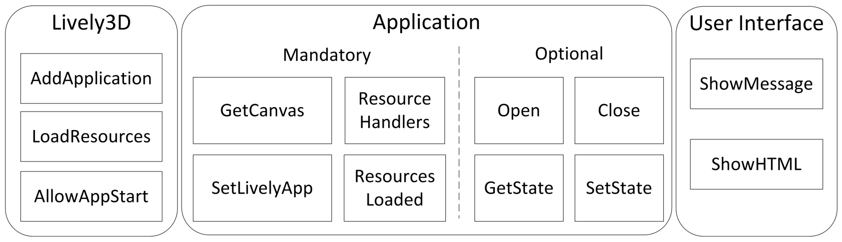


Fig. 2. Lively3D API for applications.

To convert existing application to Lively3D app, the application must implement mandatory function of the figure. To embed the converted app to environment, the initialization code of the app must start the embedding process with calling the AddApplication-function. The process is presented in Figure 3.

As illustrated in the figures, each application must implement a few mandatory functions and call Lively3D functions in certain order to advance the integra-

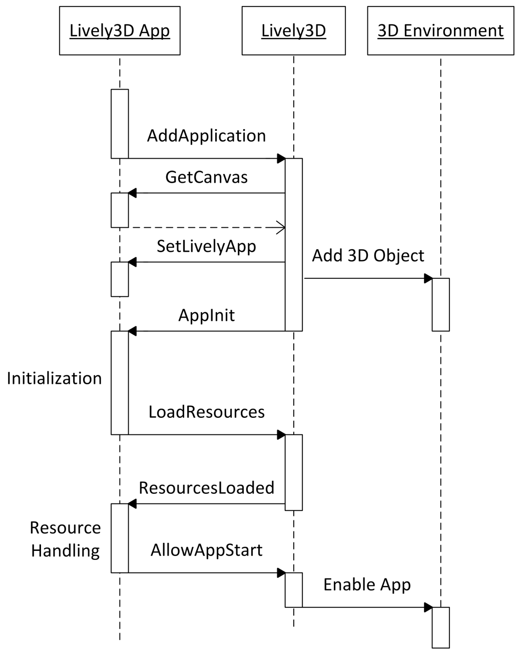


Fig. 3. Sequence for embedding new Lively3D app.

tion with the environment. During the integration, the canvas app is created and hidden with CSS-styling. Lively3D creates 3D objects representing the app and texturizes them with the canvas element. Additionally to the mandatory functions, apps can provide optional functions which react to events like opening and closing the application within the enviroment. These function have default functionalty if they are unimplemented, but if they are provided the developer can define what happens to the application status during these events. Additionally, inner state of the application can be serialized and de-serialized to developer's desired format.

Since the canvas element is defined as the only graphical element allowed for Lively3D Apps, the API also provides user interface functions to display messages and HTML in Lively3D provided dialogs. This provides consistent user interface, since Lively3D itself is rendered in a full browser window and possibilities of displaying text or other web interface elements within the environment are limited due to the WebGL specification. Figure 4 illustrates the existing canvas application in the left and the conversion to Lively3D app in the right with another app in the same environment.

3.3 Redefining the 3D environment

As is common in various 3D applications, including in particular the genre of computer games, the visualization in our system is based on so-called scene

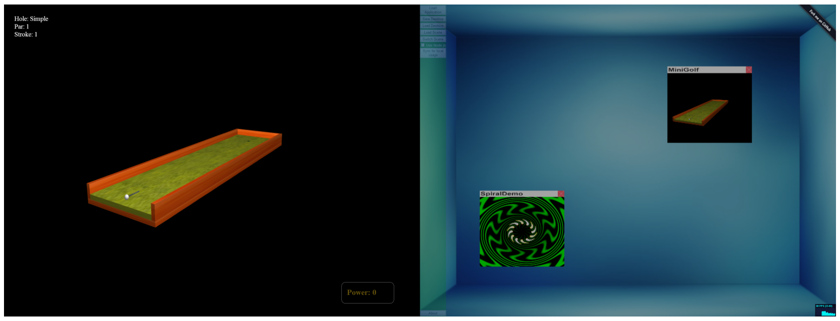


Fig. 4. Conversion of existing application.

graph, a generic tree-like data structure containing a collection of nodes. Nodes in the scene graph may have many children but most often they only need a single parent. In this structure, any operation performed to the parent is further propagated to its children. This flexible data structure enables numerous different visualizations, where the parent-children role can be benefited from.

The 3D environments in Lively3D are implemented dynamically, so that user can load new environments and change between them at will. As default only one environment is initialized in Lively3D and after adding more environments, the process of switching between environments is presented in Figure 5. Closing the applications and rebinding the events is done, so that the environment is in known initial state. Changing of the 3D-objects is required since GLGE allows 3D-object to be present only in one scene at a time.

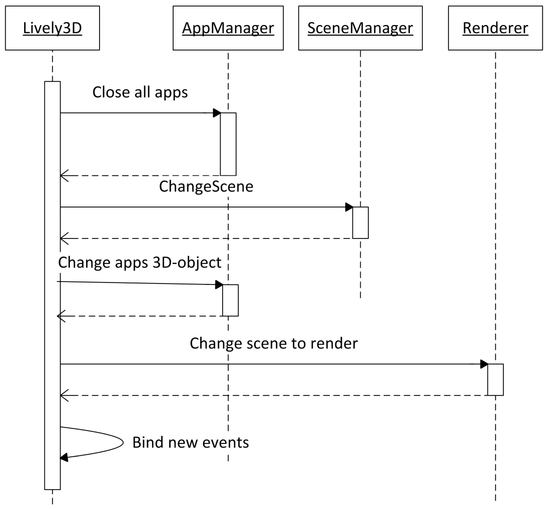


Fig. 5. Sequence of switching environment.

In our experiment, we have created three different ways to visualize a scene graph where the children are applications and the root node is the 3D environment hosting the children. Example host environments include a conventional desktop, a planetary system where applications rotate a sun like in a solar system, and a true 3D virtual world, where applications move in a 3D terrain. These are introduced in the following in more detail, together with a set of screen shots to demonstrate their visual appearance.

Desktop. The conventional desktop consists of three dimensional room, cubes that represent closed applications, and planes that act as individual applications, with the ability to execute JavaScript code, render to the screen, and so forth. A screenshot of the desktop environment, with three opened and two closed applications, is presented in Figure 6. The scene mimics all traditional desktop features, including dragging applications within the desktop and application interaction with opening, closing, maximizing and minimizing them with mouse controls.

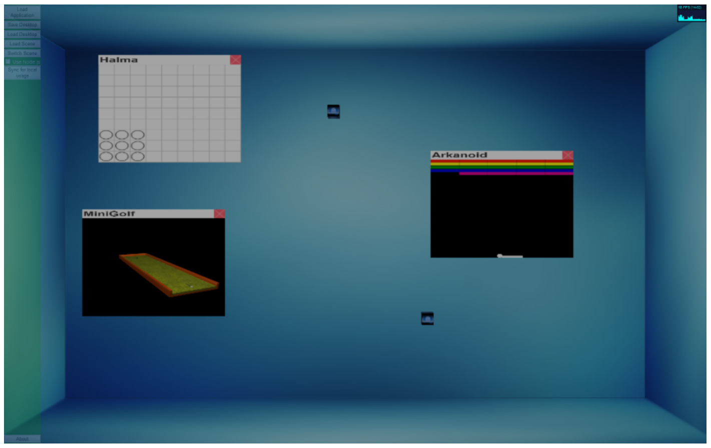


Fig. 6. Visualizing the system as a conventional desktop.

Solar system. The solar system scene modifies the presentation of applications. In this scene, applications are presented as spheres that revolve around the central sun. Each revolving sphere generates a white trace in accordance to its path, and the trace is removed when the trace reaches maximum length. Each sphere uses the texture of the application canvas it is representing, and therefore each sphere has a different look within the scene. An example scene with 4 applications is demonstrated in Figure 7. Application windows retain their default functionality with dragging around, maximizing, minimizing, and so on. When an application that has been moved around is closed, the application returns to its position revolving around the central sun, in comparison to the conventional desktop scene where the application simply retains its current position.

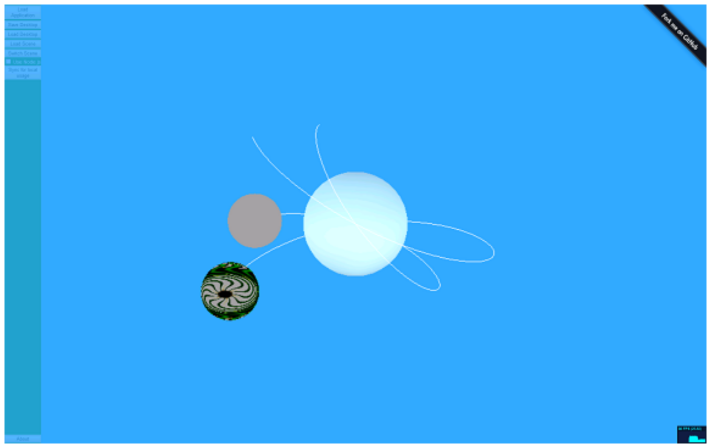


Fig. 7. Visualizing the system as a solar system.

Virtual world. The 3D virtual world scene goes even further from the conventional desktop. The only thing retained from the desktop concept are the application windows, and the only remaining controls for the windows are opening and closing the application, which then of course can introduce more controls within the application. The world itself consists of three dimensional terrain, where the user can wander around using the keyboard and the mouse. In this setting, applications are presented as spheres that roam the terrain in random directions, with their textures simplified to single image for performance reasons - experiences where application textures were used quickly showed that the resources of the test computer would no longer be adequate for such cases. Using this visualization, the 3D terrain and seven sample application spheres are illustrated in Figure 8.

All of the above visualizations are based on the same JavaScript code, with the only difference being the rendering strategy associated with the scene graph. Consequently, in all of these systems applications are runnable, and can in fact run even when they are inactive and being managed by the different host environments, except when explicitly disabled for performance reasons.

4 Lessons learned

While our prototype demonstrates that integrating individual applications within single web-page is possible and achievable without complex structures from the application developer, there still are some problems with the implementation. One of the main goals for the prototype was enabling the use of existing content. In particular, we would have liked to include complete web sites in the system as applications, creating a truly virtual world of web-based applications. However, due to the WebGL specification limitations, the use of existing content as textures is limited to image, video, and canvas elements, whereas in

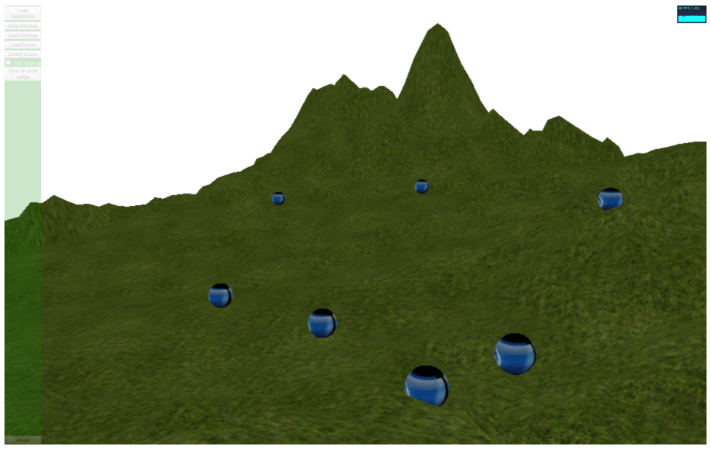


Fig. 8. Visualizing the system as a 3D virtual world.

order to render existing web pages within 3D environment, the WebGL specification should to support IFrames as a source for textures. Currently, this option is associated with security issues - using the WebGL API gives loaded applications a direct access to the host devices hardware - which must be resolved before extending the rendering capabilities. Until then, applications are limited to the functionality of canvas element to produce graphics. In principle, it would be possible to perform the necessary rendering inside canvas applications, but this option led to performance problems even in simplest cases. Additionally, the current implementation relies on individual canvas-textures for each application. This causes performance issues since large texture size is required for any meaningful application and swapping large textures in the graphics card slows down the rendering. Furthermore applications share the same JavaScript namespace which causes problems with variable overwriting. Even though each application has a simulated private namespace, variables might bleed through to the global namespace if the variable is missing var keyword. Applications can access global variables and overwrite them, including Lively3D namespace, other used JavaScript libraries and even browsers' default JavaScript functionality. This especially causes accidental problems with generic JavaScript libraries, since they are usually bound in $ variable, which is overwritten when new library is loaded and basic functionality of the environment brakes down as result. These problems could be fixed with proper process model where each application has its own private namespace and rendering context. With these improvements performance issues would be limited to individual application and application interference with each other and Lively3D would be prevented. There has been some advances in browser implementations such as faster rendering even in mobile phones and experimental browser web elements which would enable individual processes for applications, but the use cases for these are currently very limited.

One of the goals of Lively3D was minimal overhead code while embedding existing applications. We consider that this requirement was achieved quite well, although comprehensive analysis between converted applications is useless since amount of overhead code depends on coding conventions. In Lively3D most of the application initialization must be done dynamically in JavaScript code, as opposed to convential browser where HTML-tags can handle some of the resource downloading. The minimal overhead code amounts to about 50 lines of extra code, which is quite well for the goal.

In the course of the design, we were alarmed by the fact that the circumvention of security restrictions became one of the key design drivers in the experiment. In this field, the problems arise from the combination of the current "one size fits all" browser security model and the general document-oriented nature of the web browser. Decisions about security are determined primarily by the site (origin) from which the web document is loaded, not by the specific needs of the document or application. Such problems could be alleviated by introducing a more fine-grained security model, e.g., a model similar to the comprehensive security model of the Java SE platform [2] or the more lightweight, permission-based, certificate-based security model introduced by the MIDP 2.0 Specification for the Java Platform, Micro Edition (Java ME) [3]. As already pointed out in [4], the biggest challenges in this area are related to standardization, as it is difficult to define a security solution that would be satisfactory to everybody while retaining backwards compatibility. Also, any security model that depends on application signing and/or security certificates involves complicated business issues, e.g., related to who has the authority to issue security certificates, which further contribute complications. Therefore, it is likely that any resolutions in this area will still take years. Meanwhile, a large number of security groups and communities, including the Open Web Application Security Project (OWASP), the Web Application Security Consortium (WASC), and the W3C Web Security Context Working Group, are working on the problem.

Finally, there are numerous new methodological issues associated with the transition. The transition from conventional applications to web applications will result in a shift away from static programming languages such as C, C++ or C# towards dynamic programming languages. Since mainstream software developers are often unaware of the fundamental development style differences between static and dynamic programming languages, they need to be educated about the evolutionary, exploratory programming style associated with dynamic languages. Furthermore, techniques associated with dealing with big data - datasets that are too large to work with using on-hand database management tools - data mining, and mashup development will be increasingly important.

5 Conclusions

Considering the humble beginnings of the web browser as a simple document viewing and distribution environment, and the fact that programmatic capabilities on the Web were largely an afterthought rather than a carefully designed feature, the transformation of the Web into an extremely popular software deployment platform is amazing. This transformation is one of the most profound changes in the modern history of computing and software engineering. In this paper, we are demonstrating the effect of new ways to visualize content in a fashion where the browser's new extensions are based on new web protocols rather than plugins, which has been the traditional way to create richer media inside the browser. Since no plugins that commonly introduce restrictions associated with their proprietary origins, the new technologies are manifesting the open web and open data. This, together with open data that is be available to everyone to freely use and republish as they wish without mechanisms of control, in turn liberates the developers to create increasingly compelling applications, building on the facilities that already exist in the web as well as their own innovative ideas.

References

1. Anttonen, M., Salminen, A., Mikkonen, and Taivalsaari, A. Transforming the web into a real application platform: New technologies, emerging trends, and missing pieces. In Proceedings of the 26th ACM Symposium on Applied Computing (SAC'2011, TaiChung, Taiwan, March 21-25, 2011), ACM Press, Proceedings Vol 1, pp.800-807.
2. Gong, L., Ellison, G., Dageforde, M., Inside Java 2 Platform Security: Architecture, API Design, and Implementation, 2nd Edition. Addison-Wesley (Java Series), 2003.
3. Riggs, R., Taivalsaari, A., Van Peursem, J., Huopaniemi, J., Patel, M., Uotila, A., Programming Wireless Devices with the Java 2 Platform, Micro Edition (2nd Edition). Addison-Wesley (Java Series), 2003.
4. Taivalsaari, A., Mikkonen, T., Ingalls, D., and Palacz, K. Web browser as an application platform. 293-302, Proceedings of the 34th EuroMicro Conference on Software Engineering and Advanced Applications, IEEE Computer Society, 2008.
5. Taivalsaari, A., Mikkonen, T., Anttonen, M., Salminen, A. The death of binary software: End user software moves to the web. In Proceedings of the 9th International Conference on Creating, Connecting and Collaborating through Computing (C5'2011, Kyoto, Japan, January 18-20, 2011), IEEE Computer Society, pp.17-23.

Random number generator for C++ template metaprograms[*]

Zalán Szűgyi, Tamás Cséri, and Zoltán Porkoláb

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
`{lupin, cseri, gsd}@caesar.elte.hu`

Abstract. Template metaprogramming is a widely used programming paradigm to develop libraries in C++. With the help of cleverly defined templates the programmer can execute algorithms at compilation time. C++ template metaprograms are proven to be Turing-complete, thus wide scale of algorithms can be executed in compilation time. Applying randomized algorithms and data structures is, however, troublesome due to the deterministic nature of template metaprograms. In this paper we describe a C++ template metaprogram library that generates pseudorandom numbers at compile time. *Random number engines* are responsible to generate pseudorandom integer sequences with a uniform distribution. *Random number distributions* transform the generated pseudorandom numbers into different statistical distributions. Our goal was to provide similar functionality to the run-time random generator module of the Standard Template Library, thus programmers familiar with STL can easily adopt our library.

1 Introduction

Template metaprogramming is a modern, still developing programming paradigm in C++. It utilizes the instantiation technique of the C++ templates and makes the C++ compiler execute algorithms in compilation time. Template metaprograms were proven to be a Turing-complete sublanguage of C++ [4], which means that a wide set of algorithms can be executed at compile time within the limits of the C++ compiler resources.

We write template metaprograms for various reasons, like *expression templates* [23] replacing runtime computations with compile-time activities to enhance runtime performance, *static interface checking*, which increases the ability of the compile-time to check the requirements against template parameters, i.e. they form constraints on template parameters [11,19], *active libraries* [24], acting dynamically during compile-time, making decisions based on programming contexts and making optimizations. The *Ararat* system [5], *boost::xpressive*[13], and *boost::proto*[14] libraries provide metaprogramming solutions to embed DSLs into C++. Another approach to embed DSLs is to reimplement the Haskell's parser generators library with C++ template metaprograms [17].

[*] The project was supported by Ericsson Hungary.

Boost metaprogram library (MPL) [6] provides basic containers, algorithms and iterators to help in basic programmer tasks, similarly to their runtime correspondents in the Standard Template Library (STL). Fundamental types, like string are also exists in MPL, and can be extended with more complex functionality [22]. Since C++ template metaprograms follow the functional paradigma, non-STL-like approaches exists in metaprogram development too: functional programming idioms, like *nested lambda* and *let expressions* are also introduced in paper [20].

Not only libraries, but C++11, the new standard of the C++ programming language also supports template metaprogramming. There are new keywords and language structures such as *constant expressions*, *decltype*, *variadic templates* which makes writing metaprograms easier.

It is common in all the methods, techniques and libraries we discussed that C++ template metaprograms are inheritably *deterministic*. Algorithms, data types are fully specified by the source code and do not depend on any kind of external input. This means that the same metaprogram will always be executed the same way and will produce the same results (generated code, data types, compiler warnings, etc.) every time.

Therefore, it is very difficult to implement randomized algorithms and data structures with C++ template metaprograms, due to the deterministic behavior described above. Nevertheless, undeterministic algorithms and data structures are important in a certain class of tasks as they are often simpler, and more efficient than their deterministic correspondents.

Finding a minimal cut on an undirected graph is a fundamental algorithm in network theory for partitioning elements in a database, or identifying clusters of related documents. The deterministic algorithm is very complex and difficult [9], a much smaller and easier algorithm can be written using random choice [8]. A skip list [18] is an alternative to search trees. The rotation methods in search trees are algorithmically complex. Based on random numbers, skip list provides simpler way to reorganize the data structure, which is essential in template metaprograms due to their complex syntax and burdensome debug possibilities. Algorithms that selects pivot elements to partition their input sequence such as the quick sort algorithm and the similar k^{th} minimal element selection algorithm provides better worst case scenarios if we select the pivot element randomly.

In this paper we describe our C++ template metaprogram library that generates pseudorandom numbers at compile time. *Random number engines* are responsible to generate pseudorandom integer sequences with a uniform distribution. *Random number distributions* transform the generated pseudorandom numbers into different statistical distributions. Engines and distributions can be used together to generate random values. The engines are created using user defined seeds, allowing to generate repeatable random number sequences.

Our goal was to design our library similar to the runtime random library provided by the STL. Thus a programmer familiar with the STL can easily adopt our library to their metaprograms.

Our paper organizes as follows: In Section 2 we discuss those C++ template metaprogramming constructs which form the implementation base of our library. Section 3 introduces our compile time random number generator library with implementational details. In Section 4 we show how our library can be applied for real life problems and we evaluate the results. Section 5 mentions a project related to code obfuscation using some randomization in C++ template metaprograms. Future works are discussed in Section 6. Our paper concludes in Section 7.

2 Template metaprogramming

The template facilities of C++ allow writing algorithms and data structures parametrized by types. This abstraction is useful for designing general algorithms like finding an element in a list. The operations on lists of integers, characters or even user defined classes are essentially the same. The only difference between them is the stored type. With templates we can parametrize these list operations by abstract type, thus, we need to write the abstract algorithm only once. The compiler will generate the integer, double, character or user defined class version of the list by replacing the abstract type with a concrete one. This method is called *instantiation*.

The template mechanism of C++ enables the definition of partial and full *specializations*. Let us suppose that we would like to create a more space efficient type-specific implementation of the `list` template for `bool` type. We may define the following specialization:

```
template<typename T>
struct list
{
  void insert(const T& e);
  /* ... */
};

template<>
struct list<bool>
{
  //type-specific implementation
  void insert(bool e);
  /* ... */
};
```

Programs that are evaluated at compilation time are called *metaprograms*. C++ supports metaprogramming via preprocessor macros and templates. Preprocessor macros run before the C++ compilation and therefore they are unaware of the C++ language semantics. However, *template metaprograms* are evaluated during the C++ compilation phase, therefore the type safety of the language is enforced.

Template specialization is essential practice for template metaprogramming [1]. In template metaprograms templates usually refer to themselves with different type arguments. Such chains of recursive instantiations can be terminated by a template specialization. See the following example of calculating the factorial value of 5:

```
template<int N>
struct factorial
{
  enum { value = N * factorial<N-1>::value };
};

template<>
struct factorial<0>
{
  enum { value = 1 };
};

int main()
{
  int result = factorial<5>::value;
}
```

To initialize the variable `result`, the expression `factorial<5>::value` has to be evaluated. As the template argument is not zero, the compiler instantiates the general version of the `factorial` template with 5. The definition of `value` is `N * factorial<N-1>::value`, hence the compiler has to instantiate the `factorial` again with 4. This chain continues until the concrete value becomes 0. Then, the compiler choses the special version of `factorial` where the `value` is 1. Thus, the instantiation chain is stopped and the factorial of 5 is calculated and used as initial value of the `result` variable in `main`. This metaprogram “runs" while the compiler compiles the code.

Template metaprograms therefore consist of a collection of templates, their instantiations and specializations, and perform operations at compilation time. Basic control structures like iterations and conditions are represented in a functional way [21]. As we can see in the previous example, iterations in metaprograms are applied by recursion. Besides, the condition is implemented by a template structure and its specialization.

```
template<bool cond_, typename then_, typename else_>
struct if_
{
  typedef then_ type;
};

template<typename then_, typename else_>
```

```
struct if_<false, then_, else_>
{
  typedef else_ type;
};
```

The if_ structure has three template arguments: a boolean and two abstract types. If the cond_ is false, then the partly-specialized version of if_ will be instantiated, thus the type will be bound by the else_. Otherwise the general version of if_ will be instantiated and type will be bound by then_.

Complex data structures are also available for metaprograms. Recursive templates store information in various forms, most frequently as tree structures, or sequences. Tree structures are the most common forms of implementation of expression templates [23]. The canonical examples for sequential data structures are Typelist [2] and the elements of the boost::mpl library [6].

We define a type list with the following recursive template:

```
class NullType {};
struct EmptyType {};        // could be instantiated

template <typename H, typename T>
struct Typelist
{
  typedef H head;
  typedef T tail;
};
typedef Typelist< char, Typelist<signed char,
             Typelist<unsigned char, NullType> > > Charlist;
```

In the example we store the three character types in our Typelist. We can use helper macro definitions to make the syntax more readable.

```
#define TYPELIST_1(x)           Typelist< x, NullType>
#define TYPELIST_2(x, y)        Typelist< x, TYPELIST_1(y)>
#define TYPELIST_3(x, y, z)     Typelist< x, TYPELIST_2(y,z)>
// ...
typedef TYPELIST_3(char, signed char, unsigned char)  Charlist;
```

Essential helper functions – like Length, which computes the size of a list at compilation time – have been defined in Alexandrescu's Loki library[2] in pure functional programming style. Similar data structures and algorithms can be found in the boost::mpl metaprogramming library. The Boost Metaprogramming Library [6] is a general-purpose, high-level C++ template metaprogramming framework of algorithms, sequences and metafunctions. The architecture is similar to the Standard Template Library (STL) of C++ with containers, algorithms and iterators but boost::mpl offers this functionality in compilation time.

3 Compile-time random number generation

Our random number generator library for template metaprograms is designed to be similar to the runtime random number library provided by Standard Template Library (STL) of the new standard of C++. Our library provides:

– *random number engines*, that generate pseudorandom integer sequences with uniform distribution.
– *random number distributions*, that transform the output of the random number engines into different statistical distributions.

Engines and distributions can be used together to generate random values. The engines are created using user defined seeds, allowing to generate repeatable random number sequences.

3.1 Basic metafunctions

Before we detail our engines and distributions, we present some basic metafunctions which are commonly used in our implementation. The `Random` metafunction initializes a random engine or distribution, and returns with the first random number in a sequence. See the code below:

```
template<typename Engine>
struct Random
{
  typedef typename init<Engine>::type type;
  static const decltype(type::value) value = type::value;
};
```

The initialization is done by the `init` metafunction, which is partially specialized for all engines and distributions. The first random number is stored in static field `value`.

The `Next` metafunction computes the next random number in a sequence. See its code below:

```
template<typename R>
struct Next
{
  typedef typename eval<R>::type type;
  static const decltype(type::value) value = type::value;
};
```

The next element is computed by the `eval` metafunction, which is, similarly to `init`, partially specialized for all engines and distributions. The return value of this metafunction is stored in static field `value`.

3.2 Random number engines

Similarly to STL, we implemented three random number engines:

- *linear congruential engine* [15], which requires very small space to store its state, and moderately fast.
- *subtract with carry engine* [3], which produces a better random sequence, very fast, but requires more state storage.
- *Mersenne twister engine* [10], which is slower and has even more state storage requirements but with the right parameters provides the longest non-repeating sequence of random numbers.

The implementation of these engines contain three major entities. The first one is a metatype, that contains the state of the engine. This metatype is the argument of the partially specialized `init` and `eval` metafunctions, which do some initialization steps and evaluate the next random number, respectively. For the *linear congruential engine*, these entities defined as below:

```
template<typename UIntType,
         UIntType seed = defaultseed,
         UIntType a = 16807,
         UIntType c = 0,
         UIntType m = 2147483647>
struct linear_congruential_engine
{
  static const UIntType value = seed;
  static const UIntType maxvalue = m-1;
};
```

The first template argument specifies the type of the random numbers. The type can be any unsigned integer type. The second argument is the random seed. This is an optional parameter. If the programmer does not specify it, an automatically generated random seed is applied for each compilation. See Subsection 3.4 for more details. The other arguments are parameters of the linear congruential equation.

The `init` and `eval` metafunctions can be seen below:

```
template<typename UIntType,
         UIntType seed,
         UIntType a,
         UIntType c,
         UIntType m>
struct init<linear_congruential_engine<UIntType, seed, a, c, m>>
{
  typedef typename eval<linear_congruential_engine<
                            UIntType,seed,a,c,m>>::type type;
  static const UIntType value = type::value;
};
```

```
template<typename UIntType, UIntType seed,
         UIntType a, UIntType c, UIntType m>
struct eval<linear_congruential_engine<UIntType, seed, a, c, m>>
{
  static const UIntType value = (a * seed + c) % m;
  typedef linear_congruential_engine<
    UIntType,
    (a * seed + c) % m,
    a,
    c,
    m
  > type;
};
```

The metafunction `eval` computes the next random number, stores it in the static field `value` and modifies the state of the *linear congruential engine*. Metafunction `init` just invokes `eval` to compute the first random number.

3.3 Random number distributions

A *random number distribution* transforms the uniformly distributed output of a random number engine into a specific statistical distribution. Although the STL defines discrete and continuous distributions, we implement only the discrete ones in our library, because of the lack of support of floating point numbers in the template facility of C++. However, we plan to extend the template system with rational and floating point number types that we can use to implement continuous distributions. Our library implements the following discrete probability distributions: *uniform integer distribution*, *Bernoulli-distribution*, *binomial distribution*, *negative binomial distribution*, *geometric distribution*, *Poisson-distribution*, *discrete distribution* [7]. Several distributions require a real number as argument. As the template system in C++ does not accept floating point numbers, our library deals with these parameters as rational numbers: receives the numerator and the denominator as integers. The compiler can approximate the quotient inside the metafunctions. See the implementation of Bernoulli distribution below:

```
template<typename Engine, int N, int D, bool val = false>
struct Bernoulli
{
  static const bool value = val;
};

template<typename Engine, int N, int D, bool b>
struct eval<Bernoulli<Engine, N, D, b>>
{
  typedef typename Next<Engine>::type tmptype;
  static const bool value =
    (static_cast<double>(tmptype::value) / tmptype::maxvalue)
    <
    (static_cast<double>(N) / D);
  typedef Bernoulli<tmptype, N, D, value> type;

};
```

The template parameter `Engine` refers to any kind of random number engine. The integers `N` and `D` represent the parameter of Bernoulli distributions, and `val` stores the computed result. The partially specialized `eval` metafunction transforms the result of the engine to a boolean value according to the parameter and sets the new state of the `Bernoulli` class. The other distributions are implemented in similar way.

3.4 Random seed

Pseudorandom number generators require an initial state, called *random seed*. The random seed determines the generated random sequence. The random number engines in our library optionally accept seed. Specifying the seed results in reproducible sequences of random numbers. However, if the seed is omitted, the library will generate a random seed based on the current time of the system. The time is received using the `__TIME__` macro, which is preprocessed to a `constexpr` character array. Our library will compute a seed using the elements of this array. See the code below:

```
constexpr char inits[] = __TIME__;
const int defaultseed = (inits[0]-'0')*100000+(inits[1]-'0')*10000 +
                        (inits[3]-'0')*1000+(inits[4]-'0')*100+
                        (inits[6]-'0')*10+inits[7]-'0';
```

The C++ standard defines that the preprocessor of C++ should translate the `__TIME__` macro to a character string literal in *"hh:mm:ss"* form. We transform it into a six-digit integer, excluding the colons.

3.5 Example

In this subsection we show a basic usage of our library. We print ten boolean values having Bernoulli distribution with parameter 0.1. We use the *linear congruential engine* to generate the random sequence.

```
template<int cnt, typename R>
struct print_randoms
{
  static void print()
  {
    typedef typename Next<R>::type RND;
    std::cout << RND::value << " ";
    print_randoms<cnt-1, RND >::print();
  }
};

template<typename R>
struct print_randoms<0, R>
{
  static void print()
  {
    std::cout << Next<R>::value << " " << std::endl;
  }
};

int main()
{
  print_randoms<10,
                typename Random<
                  Bernoulli<
                    linear_congruential_engine<
                      uint_fast32_t>,
                    1,
                    10
                  >
                >::type
  >::print();
}
```

4 Evaluation

The Boost MPL uses deterministic algorithms in its implementation. For example the `boost::mpl::sort` metafunction always selects the first element of the current range as its pivot element. This strategy leads to worst-case scenario when the data is already sorted. It has also a performance overhead if parts of the input data are sorted. However, if we select the pivot element randomly, the worst-case scenario will occur on least common patterns.

We combined the sort algorithm with our library: we chose the pivot element randomly. We evaluated both methods with random and ordered data. Experiments show that we achieved great speedup on the ordered input sample, whereas the performance loss on the random sample is minor. (Figure 1) Compiling the templates requires not only CPU power, but the instantiations must be stored the memory as well. Because the random algorithm requires less instantiation steps, we also need less memory for the compilation. (Figure 2)

One fundamental strategy lacking from the current implementation of our library is splitting the random number generator. Metaprograms are written in functional programming style and therefore it is not possible to use a global random generator variable. This shortcoming can be overcome if the random sequencer supports splitting.

Note that the lack of splitting is not a problem in the quicksort algorithm as the two parts of the sequence become independent after separating the sequence based on the pivot element and therefore the same random numbers provide as much randomness as different ones. Therefore, this technique here is rather an optimization as it requires less template instantiation steps.

5 Related work

Neves et. al. developed a code obfuscator library for C++ programming language [12]. The strength of this library is that the obfusctation steps are template metaprograms, thus the programmer does not need to deal with obfuscation, this process is done automatically by the C++ compiler during code compilation. They used randomness to avoid the usage of the same transformation repeatedly. The implemented a very simple linear congruential method to generate template random number. Our library can be easily adopted into their solution providing a more sophisticated random number generation.

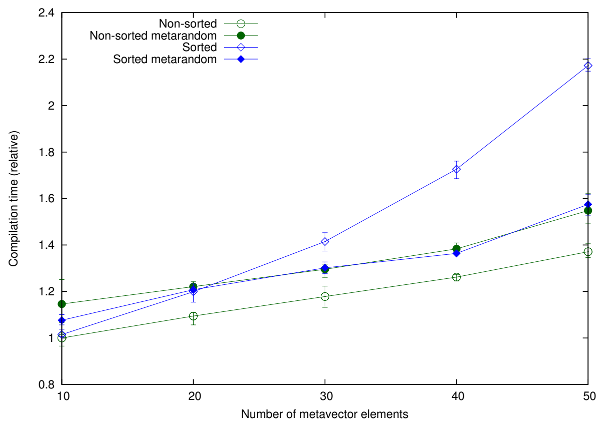


Fig. 1. Compilation times of the quicksort of the `boost::mpl::vector_c` using the original `boost::mpl::sort` and the modified sort, where the pivot element is selected randomly, on sorted and non-sorted (random) data for various vector sizes. Measured with g++ 4.7.3 on Ubuntu x86.

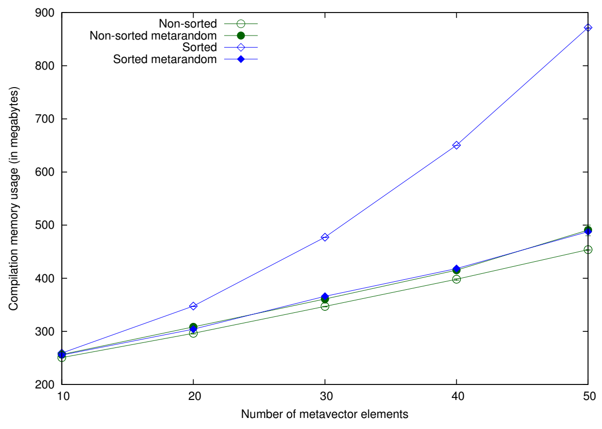


Fig. 2. Memory usages of the quicksort of the `boost::mpl::vector_c` using the original `boost::mpl::sort` and the modified sort, where the pivot element is selected randomly, on sorted and non-sorted (random) data for various vector sizes. Measured with g++ 4.7.3 on Ubuntu x86.

Meredith L. Patterson mentions a "simple compile-time pseudo-random number generator" he implemented [16], but no further details are available.

6 Future works

Our goal was to provide the metaprogamming counterpart of the random number generator library of the STL. We ported the random number generator engines and all the random number distributions that generate integral values. However, we need to extend the library with distributions that generate floating point values e.g. *normal distribution*. As the language supports only integers in template arguments, we have to find a way to circumvent this limitation. Neither the standard library, nor third party libraries offer a ready solution, therefore we need to implement a floating point metatype first. Based on this metatype we can implement the remaining statistical distributions provided by the STL.

Our library is designed to be extensible. Further random number engines and distributions can be added. The engines provide a clean and simple interface so new engines and distributions can be created orthogonally.

7 Conclusion

Template metaprogramming plays essential role in library design in C++. Several language features and third party libraries supports that paradigm. However, due to the deterministic nature of template metapograms it was difficult to implement algorithms and data structures in undeterministic way. Since sometimes randomized algorithms and data structures are often less complex and more efficient than their deterministic correspondents, it is important to generate (pseudo-)random numbers in a maintainable and effective way for template metaprograms.

We implemented random number engines that generate pseudorandom integer sequences with a uniform distribution and random number distributions that transform the generated pseudo-numbers into different statistical distributions. The library has a similar, but compile-time interface like the run-time random number generator of the STL to reduce the learning curve.

In this paper we presented our library and discussed its applicability with an example using `boost::mpl`. Besides, our library is designed to be extensible, thus one can easily add further engines and distributions to our library.

References

1. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional (2004)
2. Alexandrescu, A.: Modern C++ design: generic programming and design patterns applied. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
3. Brent, R.P.: Uniform random number generators for supercomputers. In: Proc. Fifth Australian Supercomputer Conference. pp. 95–104 (1992)
4. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000)
5. Gil, J.Y., Lenz, K.: Simple and safe sql queries with C++ templates. Sci. Comput. Program. 75, 573–595 (July 2010), `http://dx.doi.org/10.1016/j.scico.2010.01.004`
6. Gurtovoy, A., Abrahams, D.: Boost.mpl (2004), `http://www.boost.org/doc/libs/1_53_0/libs/mpl/doc/index.html`, http://www.boost.org/doc/libs/1_53_0/libs/mpl/doc/index.html
7. Johnson, N.L., Kemp, A.W., Kotz, S.: Univariate discrete distributions, vol. 444. Wiley-Interscience (2005)
8. Kleinberg, J., Tardos, É.: Algorithm Design. Alternative Etext Formats, Pearson/Addison-Wesley (2006), `http://books.google.hu/books?id=OiGhQgAACAAJ`
9. Lawler, E.: Combinatorial Optimization: Networks and Matroids. Dover Books on Mathematics Series, DOVER PUBN Incorporated (1976), `http://books.google.hu/books?id=m4MvtFenVjEC`
10. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. 8(1), 3–30 (Jan 1998), `http://doi.acm.org/10.1145/272991.272995`
11. McNamara, B., Smaragdakis, Y.: Static interfaces in C++. In: C++ Template Programming Workshop (Oct 2000)
12. Neves, S., Araujo, F.: Binary code obfuscation through C++ template metaprogramming. In: Lopes, A., Pereira, J.O. (eds.) INForum 2012. pp. 28–40. Universidade Nova de Lisboa, Portugal (September 2012), `http://eden.dei.uc.pt/~sneves/pubs/2012-snfa2.pdf`

13. Niebler, E.: Boost.xpressive (2007), http://www.boost.org/doc/libs/1_53_0/doc/html/xpressive.html, http://www.boost.org/doc/libs/1_53_0/doc/html/xpressive.html
14. Niebler, E.: The boost proto library (2011), http://www.boost.org/doc/libs/1_53_0/doc/html/proto.html, http://www.boost.org/doc/libs/1_53_0/doc/html/proto.html
15. Park, S.K., Miller, K.W.: Random number generators: good ones are hard to find. Commun. ACM 31(10), 1192–1201 (Oct 1988), http://doi.acm.org/10.1145/63039.63042
16. Patterson, Meredith, L.: Patterson's remark in stackoverflow, http://stackoverflow.com/questions/1224306/template-metaprogramming-i-still-dont-get-it, http://stackoverflow.com/questions/1224306/template-metaprogramming-i-still-dont-get-it
17. Porkoláb, Z., Sinkovics, Á.: Domain-specific language integration with compile-time parser generator library. In: Visser, E., Järvi, J. (eds.) GPCE. pp. 137–146. ACM (2010)
18. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM 33(6), 668–676 (Jun 1990), http://doi.acm.org/10.1145/78973.78977
19. Siek, J.G., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: Proceedings of the First Workshop on C++ Template Programming. Erfurt, Germany (Oct 2000), citeseer.nj.nec.com/siek00concept.html
20. Sinkovics, Á.: Nested lamda expressions with let expressions in C++ template metaprorgams. In: Porkoláb, Z., Pataki, N. (eds.) WGT'11. WGT Proceedings, vol. III, pp. 63–76. Zolix (2011)
21. Ádám Sipos, Porkoláb, Z., Zsók, V.: Meta<fun> - towards a functional-style interface for C++ template metaprograms. Studia Universitatis Babes-Bolyai Informatica LIII(2008/2), 55–66 (2008)
22. Szűgyi, Z., Sinkovics, Á., Pataki, N., Porkoláb, Z.: C++ metastring library and its applications. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE. Lecture Notes in Computer Science, vol. 6491, pp. 461–480. Springer (2009)
23. Veldhuizen, T.: Expression templates. C++ Report 7, 26–31 (1995)
24. Veldhuizen, T.L., Gannon, D.: Active libraries: Rethinking the roles of compilers and libraries. In: In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing OO'98. SIAM Press (1998)

The Asymptotic Behaviour of the Proportion of Hard Instances of the Halting Problem

Antti Valmari

Tampere University of Technology, Department of Mathematics
PO Box 553, FI-33101 Tampere, FINLAND

Abstract. Although the halting problem is undecidable, imperfect testers that fail on some instances are possible. Such instances are called *hard* for the tester. One variant of imperfect testers replies "I don't know" on hard instances, another variant fails to halt, and yet another replies incorrectly "yes" or "no". Also the halting problem has three variants. The failure rate of a tester for some size is the proportion of hard instances among all instances of that size. This publication investigates the behaviour of the failure rate as the size grows without limit. Earlier results are surveyed and new results are proven. Some of them use C++ on Linux as the computational model. It turns out that the behaviour is sensitive to the details of the programming language or computational model, but in many cases it is possible to prove that the proportion of hard instances does not vanish.

ACM Computing Classification System 1998: F.1.1 Models of Computation–Computability theory
Mathematics Subject Classification 2010: 68Q17 Computational difficulty of problems

1 Introduction

Turing proved in 1936 that undecidability exists by showing that the halting problem is undecidable [10]. Rice extended the set of known undecidable problems to cover all questions of the form "does the (partial) function computed by the given program have property X", where X is any property that at least one computable partial function has and at least one does not have [7]. For instance, X could be "returns 1 for all syntactically correct C++ programs and 0 for all remaining inputs." In other words, it may be impossible to find out whether a given weird-looking program is a correct C++ syntax checker. These results are basic material in such textbooks as [3].

On the other hand, imperfect halting testers are possible. For any instance of the halting problem, a *three-way tester* eventually answers "yes", "no", or "I don't know". If it answers "yes" or "no", then it must be correct. We say that the "I don't know" instances are *hard instances* for the tester. Also other kinds of imperfect testers have been introduced, as will be discussed in Section 2.1. Each tester has its own set of hard instances. No instance is hard for all testers.

A useless three-way tester answers "I don't know" for every program and input. A much more careful tester simulates the program at most 9^{9^n} steps, where n is the joint size of the program and its input. If the program stops by then, then the tester answers "yes". If the program repeats a configuration (that is, a complete description of the values of variables, the program counter, etc.) by then, then the tester answers "no". Otherwise it answers "I don't know".

The proofs by Turing and Rice may leave the hope that only rare artificial contrived programs yield hard instances. One could dream of a three-way tester that answers very seldom "I don't know". This publication analyses this issue, by surveying and proving results that tell how the proportion of hard instances behaves when the size of the instance grows without limit.

Section 2 presents the variants of the halting problem and imperfect testers surveyed, together with some basic results and notation. Earlier research is discussed in Section 3. The section contains some proofs to bring results into the framework of this publication. Section 4 presents some new results in the case that information can be packed densely inside the program without assuming that the program has access to it. A natural example of such information is dead code. In Section 5, results are derived for C++ programs with inputs from files. Section 6 briefly concludes this publication.

To meet the page limit, three proofs have been left out. A longer version of this publication with the missing proofs and some other additional material can be found in the Cornell University arXiv Computing Research Repository open-access e-print service http://arxiv.org/corr/home.

2 Concepts and Notation

2.1 Variants of the Halting Problem

The literature on hard instances of the halting problem considers at least three variants of the halting problem:

(A) does the given program halt on the empty input [2],
(B) does the given program halt when given itself as its input [6, 8], and
(C) does the given program halt on the given input [1, 4, 9].

Each variant is undecidable. Variant C has a different notion of instances from others: program–input pairs instead of just programs.

The literature also varies on what the tester does when it fails. Three-way testers, that is, the "I don't know" answer is used implicitly by [6], as it discusses the union of two decidable sets, one being a subset of the halting and the other of the non-halting instances. In *generic-case decidability* [8], instead of the "I don't know" answer, the tester itself fails to halt. Yet another idea is to always give a "yes" or "no" answer, but let the answer be incorrect for some instances [4, 9]. Such a tester is called *approximating*. One-sided results, where the answer is either "yes" or "I don't know", were presented in [1, 2]. For a tester of any of the three variants, we say that an instance is *easy* if the tester correctly answers "yes" or "no" on it, otherwise the instance is *hard*.

These yield altogether nine different sets of testers, which we will denote with three-way(X), generic(X), and approx(X), where X is A, B, or C. Some simple facts facilitate carrying some results from one variant of testers to another.

Proposition 1. *For any three-way tester there is a generic-case tester that has precisely the same easy "yes"-instances, easy "no"-instances, hard halting instances, and hard non-halting instances. There also is an approximating tester that has precisely the same easy "yes"-instances, at least the same easy "no"-instances, precisely the same hard halting instances, and no hard non-halting instances.*

Proof. A three-way tester can be trivially converted to the promised tester by replacing the "I don't know" answer with an eternal loop or the reply "no". □

Proposition 2. *For any generic-case tester there is a generic-case tester that has at least the same "yes"-instances, precisely the same "no"-instances, no hard halting instances, and precisely the same hard non-halting instances.*

Proof. In parallel with the original tester, the instance is simulated. (In Turing machine terminology, parallel simulation is called "dovetailing".) If the original tester replies something, the simulation is aborted. If the simulation halts, the original tester is aborted and the reply "yes" is returned. □

Proposition 3. *For any $i \in \mathbb{N}$ and tester T, there is a tester T_i that answers correctly "yes" or "no" for all instances of size at most i, and similarly to T for bigger instances.*

Proof. Because there are only finitely many instances of size at most i, there is a finite bit string that lists the correct answers for them. If $n \leq i$, T_i picks the answer from it and otherwise calls T. (We do not necessarily know what bit string is the right one, but that does not rule out its existence.) □

2.2 Notation

We use Σ to denote the set of characters that are used for writing programs and their inputs. It is finite and has at least two elements. There are $|\Sigma|^n$ character strings of size n. If $\alpha \in \Sigma^*$ and $\beta \in \Sigma^*$, then $\alpha \sqsubseteq \beta$ denotes that α is a prefix of β, and $\alpha \sqsubset \beta$ denotes proper prefix. A set A of finite character strings is *self-delimiting* if and only if membership in A is decidable and $\alpha \not\sqsubset \beta$ whenever $\alpha \in A$ and $\beta \in A$. The *shortlex ordering* of any set of finite character strings is obtained by sorting the strings in the set primarily according to their sizes and strings of the same size in the lexicographic order.

Not necessarily all elements of Σ^* are programs. The set of programs is denoted with Π, and the set of all (not necessarily proper) prefixes of programs with Γ. So $\Pi \subseteq \Gamma$. For tester variants A and B, we use $p(n)$ to denote the number of programs of size n. Then $p(n) = |\Sigma^n \cap \Pi|$. For tester variant C, $p(n)$ denotes the number of program–input pairs of joint size n. The numbers of halting and non-halting (a.k.a. diverging) instances of size n are denoted with $h(n)$ and $d(n)$, respectively. We have $p(n) = h(n) + d(n)$.

If T is a tester, then $\underline{h}_T(n)$, $\overline{h}_T(n)$, $\underline{d}_T(n)$, and $\overline{d}_T(n)$ denote the number of its easy halting, hard halting, easy non-halting, and hard non-halting instances of size n, respectively. Obviously $\underline{h}_T(n) + \overline{h}_T(n) = h(n)$ and $\underline{d}_T(n) + \overline{d}_T(n) = d(n)$. The smaller $\overline{h}_T(n)$ and $\overline{d}_T(n)$ are, the better the tester is.

When referring to all instances of size at most n, we use capital letters. So, for example, $P(n) = \sum_{i=0}^{n} p(i)$ and $\overline{D}_T(n) = \sum_{i=0}^{n} \overline{d}_T(i)$.

3 Related Work

3.1 Early Results by Lynch

Nancy Lynch [6] used *Gödel numberings* for discussing programs. In essence, it means that each program has at least one index number (which is a natural number) from which the program can be constructed, and each natural number is the index of some program.

Although the index of an individual program may be smaller than the index of some shorter program, the overall trend is that indices grow as the size of the programs grows, because otherwise we run out of small numbers. On the other hand, if the mapping between the programs and indices is 1–1, then the growth cannot be faster than exponential. This is because $p(n) \leq |\Sigma|^n$. With real-life programming languages, the growth is exponential, but (as we will see in Section 5.2) the base of the exponent may be smaller than $|\Sigma|$.

To avoid confusion, we refrain from using the notation $\overline{H}_T$, etc., when discussing results in [6], because they use indices instead of sizes of programs, and their relationship is not entirely straightforward. Fortunately, some results of [6] can be immediately applied to programming languages by using the *shortlex Gödel numbering*. The shortlex Gödel number of a program is its index in the shortlex ordering of all programs.

The first group of results of [6] reveals that a wide variety of situations may be obtained by spreading the indices of all programs sparsely enough and then filling the gaps in a suitable way. For instance, with one Gödel numbering, for each three-way tester, the proportion of hard instances among the first i indices approaches 1 as i grows. With another Gödel numbering, there is a three-way tester such that the proportion approaches 0 as i grows. There even is a Gödel numbering such that as i grows, the proportion oscillates in the following sense: for some three-way tester, it comes arbitrarily close to 0 infinitely often and for each three-way tester, it comes arbitrarily close to 1 infinitely often.

In its simplest form, spreading the indices is analogous to defining a new language SpaciousC++ whose syntax is identical to that of C++ but the semantics is different. If the first $\lfloor n/2 \rfloor$ characters of a SpaciousC++ program of size n are space characters, then the program is executed like a C++ program, otherwise it halts immediately. This does not restrict the expressiveness of the language, because any C++ program can be converted to a similarly behaving SpaciousC++ program by adding sufficiently many space characters to its front. However, it makes the proportion of easily recognizable trivially halting instances overwhelm. A program that replies “yes” if there are fewer than $\lfloor n/2 \rfloor$ space characters at the front and “I don’t know” otherwise, is a three-way tester. Its proportion of hard instances vanishes as the size of the program grows.

As a consequence of this and Proposition 3, one may choose any failure rate above zero and there is a three-way halting tester for SpaciousC++ programs with at most that failure rate. Of course, this result does not tell anything about how hard it is to test the halting of interesting programs. This is the first example in this publication of what we call *an anomaly stealing the result*. That is, a proof of a theorem goes through for a reason that has little to do with the phenomenon we are interested in.

Indeed, the first results of [6] depend on using unnatural Gödel numberings. They do not tell what happens with untampered programming languages. Even so, they rule out the possibility of a simple and powerful general theorem that applies to all models of computation. They also make it necessary to be careful with the assumptions that are made about the programming language.

To get sharper results, *optimal Gödel numberings* were discussed in [6]. They do not allow distributing programs arbitrarily. A Gödel numbering is optimal if and only if for any Gödel numbering, there is a computable function that maps it to the former such that the index never grows more than by a constant factor.[1] The most interesting sharper results are opposite to what was obtained without the optimality assumption. We now apply them to programming languages.

We say that a programming language is *end-of-file data segment*, if and only if each program consists of two parts in the following way. The first part is the actual program written in a self-delimiting language, so its end can be detected. The second part, called the data segment, is an arbitrary character string that extends to the end of the file. The language has a construct via which the actual program can read the contents of the data segment. The data segment is thus a data literal in the program, packed with maximum density. It is not the same thing as the input to the program.

Corollary 4. *For each end-of-file data segment language,*

$$\exists c > 0 : \exists T \in \text{three-way(B)} : \forall n \in \mathbb{N} : \frac{\underline{H}_T(n) + \underline{D}_T(n)}{P(n)} \geq c \textit{ and}$$

$$\exists c > 0 : \forall T \in \text{three-way(B)} : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\overline{H}_T(n) + \overline{D}_T(n)}{P(n)} \geq c \ .$$

Proof. Let a and d be the sizes of the actual program and data segment. Given any Gödel numbering, let the actual program read the data segment, interpret its content as a number i in the range from $\frac{|\Sigma|^d-1}{|\Sigma|-1} + 1$ to $\frac{|\Sigma|^{d+1}-1}{|\Sigma|-1}$, and simulate the corresponding program. The shortlex index of this program is at most $i' =$

[1] The definition in [6] seems to say that the function must be a bijection. We believe that this is a misprint, because each proof in [6] that uses optimal Gödel numberings obviously violates it.

$\sum_{j=0}^{a+d} |\Sigma|^j \leq |\Sigma|^{a+d+1}$. We have $d \leq \log_{|\Sigma|} i + 1$, so $i' \leq |\Sigma|^{a+2} i$. The shortlex numbering of the language is thus an optimal Gödel numbering. From this, Proposition 6 in [6] gives the claims. □

A remarkable feature of the latter result compared to many others in this publication is that c is chosen before T. That is, there is a positive constant that only depends on the programming language (and not on the choice of the tester) such that all testers have at least that proportion of hard instances, for any big enough n. On the other hand, the proof depends on the programming language allowing to pack raw data very densely. Real-life programming languages do not satisfy this assumption. For instance, C++ string literals `"..."` cannot pack data densely enough, because the representation of `"` inside the literal (e.g., `\"` or `\042`) requires more than one character.

The result cannot be generalized to $\overline{h}_T$, $\overline{d}_T$, and p, because the following anomaly steals it. We can first add `1` or `01` to the beginning of each program π and then declare that if the size of 1π or 01π is odd, then it halts immediately, otherwise it behaves like π. This trick does not invalidate optimality but introduces infinitely many sizes for which the proportion of hard instances is 0.

3.2 Results on Domain-Frequent Programming Languages

In [4], the halting problem was analyzed in the context of programming languages that are *frequent* in the following sense:

Definition 5. *A programming language is (a)* frequent *(b)* domain-frequent*, if and only if for every program π, there are $n_\pi \in \mathbb{N}$ and $c_\pi > 0$ such that for every $n \geq n_\pi$, at least $c_\pi p(n)$ programs of size n (a) compute the same partial function as π (b) halt on precisely the same inputs as π.*

Instead of “frequent”, the word “dense” was used in [4], but we renamed the concept because we felt “dense” a bit misleading. The definition says that programs that compute the same partial function are common. However, the more common they are, the less room there is for programs that compute other partial functions, implying that the smallest programs for each distinct partial function must be distributed more sparsely. “Dense” was used for domain-frequent in [9].

Any frequent programming language is obviously domain-frequent but not necessarily vice versa. On the other hand, even if a theorem in this field mentions frequency as an assumption, the odds are that its proof goes through with domain-frequency. Whether a real-life programming language such as C++ is (domain-)frequent, is surprisingly difficult to find out. We will discuss this question briefly in Section 4.

As an example of a frequent programming language, BF was mentioned in [4]. Its full name starts with “brain” and then contains a word that is widely considered inappropriate language, so we follow the convention of [4] and call it BF. Information on it can be found on Wikipedia under its real name. It is an exceptionally simple programming language suitable for recreational and illustrational but not for real-life programming purposes. In essence, BF programs describe Turing machines with a read-only input tape, write-only output tape, and one work tape. The alphabet of each tape is the set of 8-bit bytes. However, BF programs only use eight characters.

As a side issue, a non-trivial proof was given in [4] that only a vanishing proportion of character strings over the eight characters are BF programs. That is, $\lim_{n\to\infty} p(n)/8^n$ exists and is 0. It trivially follows that if failure to compile is considered as non-halting, then the proportion of hard instances vanishes as n grows.

The only possible compile-time error in BF is that the square brackets `[` and `]` do not match. Most, if not all, real-life programming languages have parentheses or brackets that must match. So it seems likely that compile-time errors dominate also in the case of most, if not all, real-life programming languages. Unfortunately, this is difficult to check rigorously, because the syntax and other compile-time rules of real-life programming languages are complicated. Using another, simpler line of argument, we will prove the result for both C++ and BF in Section 5.1.

In any event, if the proportion of hard instances among all character strings vanishes because the proportion of programs vanishes, that is yet another example of an anomaly stealing the result. It is uninteresting in itself, but it rules out the possibility of interesting results about the proportion of hard instances of size n among all character strings of size n. Therefore, from now on, excluding Section 5.1, we focus on the proportion of hard instances among all programs or program–input pairs.

In the case of program–input pairs, the results may be sensitive to how the program and its input are combined into a single string that is used as the input of the tester. To avoid anomalous results, it was assumed in [4, 9] that this "pairing function" has a certain property called "pair-fair". The commonly used function $x + (x+y)(x+y+1)/2$ is pair-fair. To use this pairing function, strings are mapped to numbers and back via their indices in the shortlex ordering of all finite character strings.

A proof was sketched in [9] that, with domain-frequency and pair-fairness,

$$\forall T \in \mathrm{approx(C)} : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\overline{h}_T(n) + \overline{d}_T(n)}{p(n)} \geq c_T \ .$$

That is, the proportion of wrong answers does not vanish. However, this leaves open the possibility that for any failure rate $c > 0$, there is a tester that fares better than that for all big enough n. This possibility was ruled out in [4], assuming frequency and pair-fairness. (It is probably not important that frequency instead of domain-frequency was assumed.) That is, there is a positive constant such that for any tester, the proportion of wrong answers exceeds the constant for infinitely many sizes of instances.

$$\exists c > 0 : \forall T \in \mathrm{approx(C)} : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\overline{h}_T(n) + \overline{d}_T(n)}{p(n)} \geq c \qquad (1)$$

The third main result in [4], adapted and generalized to the present setting, is the following. We present its proof in the arXiv CoRR version of this publication, to obtain the generalization and to add a detail that the proof in [4] lacks, that is, how $T_{i,j}$ is made to halt for "wrong sizes". Generic-case testers are not mentioned, because Proposition 2 gave a related result for them.

Theorem 6. *For each programming model and variant A, B, C of the halting problem,*

$$\forall c > 0 : \exists T \in \text{approx(X)} \quad : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\overline{h}_T(n)}{p(n)} \leq c \wedge \frac{\overline{d}_T(n)}{p(n)} = 0 \textit{ and}$$

$$\forall c > 0 : \exists T \in \text{three-way(X)} : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\overline{h}_T(n)}{p(n)} \leq c \ .$$

For a small enough c and the approximating tester T in Theorem 6, (1) implies that the failure rate of T oscillates.

3.3 Results on Turing Machines

For Turing machines with one-way infinite tape and randomly chosen transition function, the probability of falling off the left end of the tape before halting or repeating a local state approaches 1 as the number of local states grows [2]. (A local state is a state of the finite automaton component of the Turing machine, and not the configuration consisting of a local state, the contents of the tape, and the location of the head on the tape.) The tester simulates the machine until it falls off the left end, halts, or repeats a local state. If falling off the left end is considered as halting, then the proportion of hard instances vanishes as the size of the machine grows. This can be thought of as yet another example of an anomaly stealing the result.

Formally, $\exists T \in \text{three-way(X)} : \lim_{n\to\infty}(\overline{h}_T(n) + \overline{d}_T(n))/p(n) = 0$, that is,

$$\exists T \in \text{three-way(X)} : \forall c > 0 : \exists n_c \in \mathbb{N} : \forall n \geq n_c : \frac{\overline{h}_T(n) + \overline{d}_T(n)}{p(n)} \leq c \ .$$

Here X may be A, B, or C. Although A was considered in [2], the proof also applies to B and C. Comparing the result to Theorem 7 in Section 4 reveals that the representation of programs as transition functions of Turing machines is not domain-frequent.

On the other hand, independently of the tape model, the proportion does not vanish exponentially fast [8]. There, too, the proportion is computed on the transition functions, and not on some textual representations of the programs. The proof relies on the fact that any Turing machine has many obviously similarly behaving copies of bigger and bigger sizes. They are obtained by adding new states and transitions while keeping the original states and transitions intact. So the new states are unreachable. These copies are not common enough to satisfy Definition 5, but they are common enough to rule out exponentially fast vanishing. Generic-case decidability was used in [8], but the result applies also to three-way testers by Proposition 1.

The results in [1] are based on using weighted running times. For every positive integer k, the proportion of halting programs that do not halt within time $k+c$ is less than 2^{-k}, simply because the proportion of times greater than $k+c$ is less than 2^{-k}. The publication presents such a weighting that c is a computable constant.

Assume that programs are represented as self-delimiting bit strings on the input tape of a universal Turing machine. The smallest three-way tester on the empty input that answers "yes" or "no" up to size n and "I don't know" for bigger programs, is of size $n \pm O(1)$ [11].

4 More on Domain-Frequent Programming Languages

The assumption that the programming language is domain-frequent (Definition 5) makes it possible to use a small variation of the standard proof of the non-existence of halting testers, to prove that each halting tester of variant B has a non-vanishing set of hard instances. For three-way and generic-case testers, one can also say something about whether the hard instances are halting or not. Despite its simplicity, as far as we know, the following result has not been presented in the literature. However, see the comment on [9] in Section 3.2.

Theorem 7. *If the programming language is domain-frequent, then*

$$\forall T \in \text{three-way(B)} : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\overline{h}_T(n)}{p(n)} \geq c_T \land \frac{\overline{d}_T(n)}{p(n)} \geq c_T \ ,$$

$$\forall T \in \text{generic(B)} : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\overline{d}_T(n)}{p(n)} \geq c_T \ , \textit{ and}$$

$$\forall T \in \text{approx(B)} : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\overline{h}_T(n) + \overline{d}_T(n)}{p(n)} \geq c_T \ .$$

(The proof is in the arXiv CoRR version of this publication.)

The second claim of Theorem 7 lacks a $\overline{h}_T(n)$ part. Indeed, Proposition 2 says that with generic-case testers, $\overline{h}_T(n)$ can be made 0. With approximating testers, $\overline{h}_T(n)$ can be made 0 at the cost of $\overline{d}_T(n)$ becoming $d(n)$, by always replying "yes". Similarly, $\overline{d}_T(n)$ can be made 0 by always replying "no".

The next theorem applies to testers of variant A and presents some results similar to Theorem 7. To our knowledge, it is the first theorem of its kind that applies to the halting problem on the empty input. It makes a somewhat stronger assumption than Theorem 7. We say that a programming language is *computably domain-frequent* if and only if there is a decidable equivalence relation "$\approx$" between programs such that for each programs π and π', if $\pi \approx \pi'$, then π and π' halt on precisely the same inputs, and there are $c_\pi > 0$ and $n_\pi \in \mathbb{N}$ such that for every $n \geq n_\pi$, at least $c_\pi p(n)$ programs of size n are equivalent to π. If $\pi \approx \pi'$, we say that π' is a *cousin* of π. It can be easily seen from [4] that BF is computably domain-frequent.

Theorem 8. *If the programming language is computably domain-frequent, then*

$$\forall T \in \text{three-way(A)} : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\overline{d}_T(n)}{p(n)} \geq c_T \ .$$

The result also holds for generic-case testers but not for approximating testers.

Proof. Given any three-way tester T, consider a program P_T that behaves as follows. First it constructs its own code and stores it in a string variable. Hard-wiring the code of a program inside the program is somewhat tricky, but it is well known that it can be done. With Gödel numberings, the same can be obtained with Kleene's second recursion theorem.

Then P_T starts constructing its cousins of all sizes and tests each of them with T. By the assumption, there are $c_T > 0$ and $n_T \in \mathbb{N}$ such that for every $n \geq n_T$, P_T has at least $c_T p(n)$ cousins of size n. If T ever replies "yes", then P_T jumps into an eternal loop and thus does not continue testing its cousins. If T ever replies "no", then P_T halts immediately. If T replies "I don't know", then P_T tries the next cousin.

If T ever replies "yes", then P_T fails to halt on the empty input. The tested cousin halts on the same inputs as P_T, implying that also it fails to halt on the empty input. So the answer "yes" would be incorrect. Similarly, if T ever replies "no", that would be incorrect. So T must reply "I don't know" for all its cousins. They are thus hard instances for T. Because there are infinitely many of them, P_T does not halt, so they are non-halting.

To prove the result for generic-case testers, it suffices to run the tests of the cousins in parallel, that is, go around a loop where each test that has been started is executed one step and the next test is started. If any test ever replies "yes" or "no", P_T aborts all tests that it has started and then does the opposite of the reply.

A program that always replies "no" is an approximating tester with $\overline{d}_T(n) = 0$ for every $n \in \mathbb{N}$. □

The results in this section and Section 3.2 motivate the question: are real-life programming languages domain-frequent? For instance, is C++ domain-frequent? Unfortunately, we have not been able to answer it. We try now to illustrate why it is difficult.

Given any C++ program, it is easy to construct many longer programs that behave in precisely the same way, by adding space characters, line feeds (denoted with ↵), comments, or dead code such as `if(0!=0)`{...}. It is, however, hard to verify that many enough programs are obtained in this way.

For instance, any program of size n can be converted to $(|\Sigma| - 3)^k$ identically behaving programs of size $n + k + 12$ by adding {`char*s="`σ`";`} to the beginning of some function, where $\sigma \in (\Sigma \setminus \{", \backslash, ↵\})^k$. More programs are obtained by including escape codes such as `\"` to σ. However, it seems that this is a vanishing instead of at least a positive constant proportion when $k \to \infty$. In the absence of escape codes, it certainly is a vanishing proportion. This is because one can add {`char*s="`σ`",*t="`ρ`";`} instead, where $|\sigma| + |\rho| = k - 6$. Without escape codes, this yields $(k-5)(|\Sigma|-3)^{k-6}$ programs. The crucial issue here is that information can be encoded into the size of σ, while keeping $\sigma\rho$ intact. Counting the programs in the presence of escape codes is too difficult, but it seems likely that the phenomenon remains the same.

We conclude this section by showing that if dead information can be added extensively enough, a tester with an arbitrarily small positive failure rate exists. An *end-of-file dead segment language* is defined otherwise like end-of-file data segment language, but the actual program cannot read the data segment. This is the situation with any self-delimiting real-life programming language, whose compiler stops reading its input when it has read a complete program.

Theorem 9. *For each end-of-file dead segment language when X is A or B,*

$$\forall c > 0 : \exists T \in \text{three-way(X)} : \forall n \in \mathbb{N} : \frac{\overline{h}_T(n) + \overline{d}_T(n)}{p(n)} \leq c \ .$$

The result also holds with approximating and generic testers.

Proof. Let $r(n)$ denote the number of programs whose data segment is not empty. For each $n \in \mathbb{N}$, $r(n+1) = |\Sigma|p(n) \geq |\Sigma|r(n)$. So $r(n)|\Sigma|^{-n}$ grows as n grows. On the other hand, it cannot grow beyond 1, because $r(n) \leq p(n) \leq |\Sigma|^n$. So it has a limit. We call it ℓ. Because programs exist, $\ell > 0$. For every $c > 0$ we have $\ell c > 0$, so there is $n_c \in \mathbb{N}$ such that $r(n_c)|\Sigma|^{-n_c} \geq \ell - \ell c$. On the other hand, $p(n) = r(n+1)/|\Sigma| \leq \ell|\Sigma|^n$.

These imply $p(n_c-1)|\Sigma|^{n-n_c+1}/p(n) = r(n_c)|\Sigma|^{n-n_c}/p(n) \geq 1-c$. Let n_a be the size of the actual program. Consider a three-way tester that looks the answer from a look-up table if $n_a < n_c$ and replies "I don't know" if $n_a \geq n_c$ (cf. Proposition 3). It has $(\underline{h}_T(n) + \underline{d}_T(n))/p(n) \geq 1-c$, implying the claim.

Proposition 1 generalizes the result to approximating and generic testers. □

5 Results on C++ without Comments and with Input

5.1 The Effect of Compile-Time Errors

We first show that among all character strings of size n, those that are not C++ programs — that is, those that yield a compile-time error — dominate overwhelmingly, as n grows. In other words, a random character string is not a C++ program except with vanishing probability. The result may seem obvious until one realizes that a C++ program may contain comments and string literals which may contain almost anything. Therefore, it is worth the effort to prove the result rigorously, in particular because the effort is small. We prove it in a form that also applies to BF.

C++ is not self-delimiting. After a complete C++ program, there may be, for instance, definitions of new functions that are not used by the program. This is because a C++ program can be compiled in several units, and the compiler does not check whether the extra functions are needed by another compilation unit. Even so, if π is a C++ program, then π`0` is definitely not. If π is a BF program, then π`]` is not.

Proposition 10. *If for every $\pi \in \Pi$ there is $c \in \Sigma$ such that $\pi c \notin \Pi$, then*

$$\lim_{n\to\infty} \frac{p(n)}{|\Sigma|^n} = 0 \ .$$

Proof. Let $q(n) = |\Sigma^n \cap \Gamma|$. Obviously $0 \leq p(n) \leq q(n) \leq |\Sigma|^n$.

Assume first that for every $\varepsilon > 0$, there is $n_\varepsilon \in \mathbb{N}$ such that $p(n)/q(n) < \varepsilon$ for every $n \geq n_\varepsilon$. Because $p(n)/|\Sigma|^n \leq p(n)/q(n)$, we get $p(n)/|\Sigma|^n \to 0$ as $n \to \infty$.

In the opposite case there is $\varepsilon > 0$ such that $p(n)/q(n) \geq \varepsilon$ for infinitely many values of n. Let they be $n_1 < n_2 < \ldots$. By the assumption, $q(n_i + 1) \leq |\Sigma|q(n_i) - p(n_i) \leq (|\Sigma| - \varepsilon)q(n_i)$. For the remaining values of n, obviously $q(n + 1) \leq |\Sigma|q(n)$. These imply that when $n > n_i$, $p(n)/|\Sigma|^n \leq q(n)/|\Sigma|^n \leq q(n_i)/|\Sigma|^{n_i} \leq (1 - \varepsilon/|\Sigma|)^i \to 0$ when $i \to \infty$, which happens when $n \to \infty$. □

Consider a tester T that replies "no" if the compilation fails and "I don't know" otherwise. If compile-time error is considered as non-halting, then Proposition 10 implies that $\underline{h}_T(n) \to 0$, $\overline{h}_T(n) \to 0$, $\underline{d}_T(n) \to 1$, and $\overline{d}_T(n) \to 0$ when $n \to \infty$. As we pointed out in Section 3.2, this is yet another instance of an anomaly stealing the result.

5.2 The C++ Language Model

The model of computation we study in this section is program–input pairs, where the programs are written in the widely used programming language C++, and the inputs obey the rules stated by the Linux operating system. Furthermore, Σ is the set of all 8-bit bytes. To make firm claims about details, it is necessary to fix some language and operating system. The validity of the details below has been checked with C++ and Linux. Most likely many other programming languages and operating systems could have been used instead.

There are two deviations from the real everyday programming situation. First, of course, it must be assumed that unbounded memory is available. Otherwise everything would be decidable. (However, at any instant of time, only a finite number of bits are in use.) Second, it is assumed that the programs do not contain comments. This assumption needs a discussion.

Comments are information that is inside the program but ignored by the compiler. They have no effect to the behaviour of the compiled program. With them, programmers can write notes inside the program that help understand the program code, etc. We show next that most long C++ programs consist of a shorter C++ program and one or more comments.

Lemma 11. *At most $(|\Sigma| - 1)^n$ comment-less C++ programs are of size n.*

Proof. Everywhere inside a C++ program excluding comments, it is either the case that `@` or the case that ↵ cannot occur next. That is, for every character string α, either α`@` or α↵ is not a prefix of any comment-less C++ program. □

Lemma 12. *If $n \geq 16$, then there are at least $((|\Sigma| - 1)^4 + 1)^{(n-19)/4}$ C++ programs of size n.*

Proof. Let $A = \Sigma \setminus \{\texttt{*}\}$, and let $m = \lfloor n/4 - 4 \rfloor = \lceil (n-19)/4 \rceil$. Consider the character strings of the form `int main(){/*`$\alpha\beta$`*/}` , where α consists of at most three space characters and β is any string of the form $\beta_1\beta_2\cdots\beta_m$, where $\beta_i \in A^4 \cup \{\texttt{*//*}\}$ for $1 \leq i \leq m$. Each such string is a syntactically correct C++ program. Their number is $((|\Sigma|-1)^4+1)^m \geq ((|\Sigma|-1)^4+1)^{(n-19)/4}$. □

Corollary 13. *The proportion of comment-less C++ programs among all C++ programs of size n approaches 0, when $n \to \infty$.*

Proof. Let $s = |\Sigma| - 1$. By Lemmas 11 and 12, the proportion is at most $s^n/(s^4+1)^{(n-19)/4} = s^{19}(s^4/(s^4+1))^{(n-19)/4} \to 0$, when $n \to \infty$. □

As a consequence, although comments are irrelevant for the behaviour of programs, they have a significant effect on the distribution of long C++ programs. To avoid the risk that they cause yet another anomaly stealing the result, we restrict ourselves to C++ programs without comments. This assumption does not restrict the expressive power of the programming language, but reduces the number of superficially different instances of the same program.

The input may be any finite string of bytes. This is how it is in Linux. Although not all such inputs can be given directly via the keyboard, they can be given by directing the so-called standard input to come from a file. There is a separate test construct in C++ for detecting the end of the input, so the end of the input need not be distinguished by the contents of the input. There are 256^n different inputs of size n.

The sizes of a program and input are the number of bytes in the program and the number of bytes in the input file. This is what Linux reports. The size of an instance is their sum. Analogously to Section 4, the size of a program is additional information to the concatenation of the program and the input. This is ignored by our notion of size. However, the notion is precisely what programmers mean with the word. Furthermore, the convention is similar to the convention in ordinary (as opposed to self-delimiting) Kolmogorov complexity theory [5].

Lemma 14. *With the programming model in Section 5.2, $p(n) < |\Sigma|^{n+1}$.*

Proof. By Lemma 11, the number of different pairs of size n is at most

$$\sum_{i=0}^{n}(|\Sigma|-1)^i|\Sigma|^{n-i} \;=\; |\Sigma|^n \sum_{i=0}^{n}\Big(\frac{|\Sigma|-1}{|\Sigma|}\Big)^i \;<\; |\Sigma|^n \sum_{i=0}^{\infty}\Big(\frac{|\Sigma|-1}{|\Sigma|}\Big)^i \;=\; |\Sigma|^{n+1} \,.$$

□

5.3 The Proportions of Hard Instances

The next theorem says that with halting testers of variant C and comment-less C++, the proportions of hard halting and hard non-halting instances do not vanish.

Theorem 15. *With the programming model in Section 5.2,*

$$\forall T \in \text{three-way(C)} : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\overline{h}_T(n)}{p(n)} \geq c_T \wedge \frac{\overline{d}_T(n)}{p(n)} \geq c_T \;.$$

Proof. We prove first the $\overline{h}_T(n)/p(n) \geq c_T$ part and then the $\overline{d}_T(n)/p(n) \geq c_T$ part. The results are combined by picking the bigger n_T and the smaller c_T.

There is a program P_T that behaves as follows. First, it gets its own size n_p from a constant in its program code. The constant uses some characters and thus affects the size of P_T. However, the size of a natural number constant m is $\Theta(\log m)$ and grows in steps of zero or one as m grows. Therefore, by starting with $m = 1$ and incrementing it by steps of one, it eventually catches the size of the program, although also the latter may grow.

Then P_T reads the input, counting the number of the characters that it gets with n_i and interpreting the string of characters as a natural number x in base $|\Sigma|$. We have $0 \leq x < |\Sigma|^{n_i}$, and any natural number in this range is possible. Let $n = n_p + n_i$.

Next P_T constructs every program–input pair of size n and tests it with T. In this way P_T gets the number $\underline{h}_T(n)$ of easy halting pairs of size n.

Then P_T constructs again every pair of size n. This time it simulates each of them in parallel until $\underline{h}_T(n) + x$ of them have halted. Then it aborts the rest and halts. It halts if and only if $\underline{h}_T(n) + x \leq h(n)$. (It may be helpful to think of x as a guess of the number of hard halting pairs.)

Among the pairs of size n is P_T itself with the string that represents x as the input. We denote it with (P_T, x). The time consumption of any simulated execution is at least the same as the time consumption of the corresponding genuine execution. So the execution of (P_T, x) cannot contain properly a simulated execution of (P_T, x). Therefore, either (P_T, x) does not halt, or the simulated execution of (P_T, x) is still continuing when (P_T, x) halts. In the former case, $h(n) < \underline{h}_T(n) + x$. In the latter case (P_T, x) is a halting pair but not counted in $\underline{h}_T(n) + x$, so $h(n) > \underline{h}_T(n) + x$. In both cases, $x \neq h(n) - \underline{h}_T(n)$.

As a consequence, no natural number less than $|\Sigma|^{n_i}$ is $\overline{h}_T(n)$. So $\overline{h}_T(n) \geq |\Sigma|^{n_i} = |\Sigma|^{n-n_p}$. By Lemma 14, $p(n) < |\Sigma|^{n+1}$. So for any $n \geq n_p$, we have $\overline{h}_T(n)/p(n) > |\Sigma|^{-n_p-1}$.

The proof of the $\overline{d}_T(n)/p(n) \geq c_T$ part is otherwise similar, except that P_T continues simulation until $p(n) - \underline{d}_T(n) - x$ pairs have halted. (Now x is a guess of $\overline{d}_T(n)$, yielding a guess of $h(n)$ by subtraction.) The program P_T gets $p(n)$ by counting the pairs of size n whose program part is compilable. It turns out that $p(n) - \underline{d}_T(n) - x \neq h(n)$, so x cannot be $\overline{d}_T(n)$, yielding $\overline{d}_T(n) \geq |\Sigma|^{n_i}$. □

Next we adapt the second main result in [4] to our present setting, with a somewhat simplified proof (see the arXiv CoRR version of this publication) and obtaining the result separately for hard halting and hard non-halting instances.

Theorem 16. *With the programming model of Section 5.2,*

$$\exists c > 0 : \forall T \in \text{three-way(C)} : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\overline{h}_T(n)}{p(n)} \geq c \wedge \frac{\overline{d}_T(n)}{p(n)} \geq c \quad \textit{and}$$

$$\exists c > 0 : \forall T \in \text{generic(C)} \quad : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\overline{d}_T(n)}{p(n)} \geq c\ .$$

6 Conclusions

This study did not cover all combinations of a programming model, variant of the halting problem, and variant of the tester. So there is a lot of room for future work. The results highlight what was already known since [6]: the programming model has a significant role. With some programming models, a phenomenon of secondary interest dominates the distribution of programs, making hard instances rare. Such phenomena include compile-time errors and falling off the left end of the tape of a Turing machine.

Many results were derived using the assumption that information can be packed very densely in the program or the input file. Often it was not even necessary to assume that the program could use the information. Intuition suggests that if the program can access it, testing halting is harder than in the opposite case. A comparison of Corollary 4 to Theorem 9 seems to support this intuition.

Acknowledgements

I thank professor Keijo Ruohonen for helpful discussions, and the anonymous reviewers for their comments.

References

1. Calude, C.S., Stay, M.A.: Most Programs Stop Quickly or Never Halt. Advances in Applied Mathematics 40, 295–308 (2008)
2. Hamkins, J.D., Miasnikov, A.: The Halting Problem is Decidable on a Set of Asymptotic Probability One. Notre Dame Journal of Formal Logic 47(4), 515–524 (2006)
3. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
4. Köhler, S., Schindelhauer, C., Ziegler, M.: On Approximating Real-World Halting Problems. In: Liśkiewicz, M., Reischuk, R. (eds.): Proc. 15th Fundamentals of Computation Theory, Lecture Notes in Computer Science 3623, 454–466 (2005)
5. Li, M., Vitányi, P.: An Introduction to Kolmogorov Complexity and Its Applications. Springer-Verlag (2008)
6. Lynch, N.: Approximations to the Halting Problem. Journal of Computer and System Sciences 9, 143–150 (1974)
7. Rice, H.G.: Classes of Recursively Enumerable Sets and Their Decision Problems. Trans. AMS 89, 25–59 (1953)
8. Rybalov, A.: On the Strongly Generic Undecidability of the Halting Problem. Theoretical Computer Science 377, 268–270 (2007)
9. Schindelhauer, C., Jakoby, A.: The Non-recursive Power of Erroneous Computation. In: Pandu Rangan, C., Raman, V., Ramanujam, R. (eds.): Proc. 19th Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 1738, 394–406 (1999)
10. Turing, A.M.: On Computable Numbers with an Application to the Entscheidungsproblem. Proc. London Math. Soc. 2: 42, 230–265 (1936)
11. Valmari, A.: Sizes of Up-to-n Halting Testers. In: Halava, V., Karhumäki, J., Matiyasevich, Y. (eds.): Proceedings of the Second Russian Finnish Symposium on Discrete Mathematics, TUCS Lecture Notes 17, Turku, Finland, 176–183 (2012)

Implementation of Natural Language Semantic Wildcards using Prolog[*]

Zsolt Zsigmondi, Attila Kiss

Department of Information Systems,
Eötvös Loránd University,
Pázmány Péter sétány 1/C, Budapest, Hungary, H-1117
zsolt.zsigmondi@hotmail.com, kiss@inf.elte.hu

Abstract. This paper introduces the concept of semantic wildcards which is the idea of generalizing full-text search in a notion that enables user to search within additional layers of syntactic or semantic information retrieved from natural language texts. We distinguish two approaches to use semantic wildcards in search expressions: *pre-defined wildcards* which offers a straightforward and accurate query-syntax, frequently used in corpus search engines by computational linguists, and the concept of natural language wildcards which enables the user to specify the wildcards as a part of a natural language query. We will show that there are two ways to define the matching behavior in the field of natural language wildcards and we will see that the Prolog programming language offers a clean and declarative solution for implementing a search engine when using dependency-based matching.

1 Introduction

There is an increasing demand for information retrieval from natural language texts [12]. In this paper we present an alternative with which we can improve the flexibility of keyword search so that it does not change the fundamental principle of operation of these engines. This idea is called semantic wildcard search. In information retrieval systems, we provide software that satisfies the user's information needs based on the (information) sources. These systems are classified according to several criteria: now we are dealing with systems where the user's information needs take the form of queries on information resources containing natural language texts. On the query form we distinguish answering systems or keyword search systems. Keyword search engines are usually implemented in a way which means that we are searching in a mostly schema-less database, generally in a keyword search-optimized index instead of highly structured (e.g., relational) databases.

[*] This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013).

2 Concepts and motivations

From user's perspective, the operating principles of a keyword search are easy to understand, intuitive to use - but the lack of appropriate search terms often imply an unsatisfactory result set.
The problem of incomplete search terms can be treated with so-called wildcard characters. Wildcard characters appear in various forms in different search engines: syntax and semantics vary depending on the specifications of the search engine, on the mathematical basis of the search engine and on the model representation. For example, in regular expressions, the dot (.) character represents any single character. Such regular expressions based on wildcards are used by several corpus search engines, such as the WebCorp [1] or KwiCFinder [2]. With their help one can search for different forms of a word (the search term play* substitutes all forms of the verb play). The concept of wildcards does not necessarily have to exist on the level of abstraction of characters. The semantic vectors package [3], based on the Apache Lucene, for example, allows the construction of permutation indices and wildcard searches in them. The support of wildcard search functions in the semantic vectors package is unfortunately limited because it is currently only possible to use only a single wildcard in a search expression.
The concept of semantic wildcard generalizes the notion of wildcard [11]. The semantic vectors package is a good example of using of wildcards and adequate representation we may be able to reveal the hidden semantic relationships between certain texts. After that comes a logical generalization direction: expanding the number of possible wildcards so that different wildcards can convey various types of semantic information. The discussion of the search task can be specified in several ways, you can reach from a less general definition to a flexible user search terms. But before turning to this, we summarize the key concepts.
The basic unit of information retrieval is the document. Documents consist of natural language fields. The goal of the search engine is to determine the list of relevant documents on the basis of the user's query. The relevance expresses the extent that the document meets the information needs of the users. Quantitative measure of a document is its relevance level - this can be binary (relevant / irrelevant), or expressed in a continuous scale. For example, in the interval [0,1], 0 represents the total irrelevancy and 1 is the entirely relevant level. Relevance is an elusive concept, as it is difficult to determine exactly what kind of documents satisfy an information need.
The users (or experts) are responsible for the determination of levels of relevance. The search system estimates the relevance level of each document during the operation. This process is called scoring, during which the search engine assigns score values to the retrieved documents.

2.1 Pattern search with pre-defined wildcards

The idea of semantic wildcards allows the user to specify wildcards having meaningful semantic information in the search expression. Formally, this means the following.

If T is the set of possible words in a natural language (e.g., English), then a *natural language sentence* of *n+1* words is

$$S = t_0 t_1 t_2 \dots\ t_n, \text{ where } \forall t_i \in T.$$

A *query* is

$$Q = q_0 q_1 \dots q_m, \text{ where } \forall q_i \in T \cup W,$$

where *W* is the set of the possible semantic wildcards. The set *W* determines only the possible syntax of the search expressions (the grammar is unambiguous, provided that $W \cap T = \emptyset$), but not the semantics.

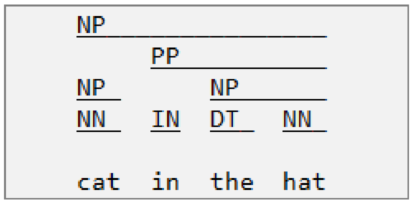
The elements of the set *W* specify when a semantic wildcard matches by a part of a natural language sentence, and how this might affect the document's score. The definition of matching we have great freedom depending on which type of semantic content we want to be recognized for the analysis of natural language texts. As an example, consider a case where only automatic named entity recognition (NER) is performed. Now $W = \{entity, person, quantity, date, organization\}$ is a rational choice, where each element in the set matches the words in the original sentence, which are recognized as that type of entity by the language processing module of the indexer of the search engine. This simple example also shows that one may want to define relations between the elements of *W*: in our example *entity* is the most general semantic wildcard - the matching of the others implies the matching of *entity* as well. In corpus search systems it is more reasonable to recognize syntax units, word structures instead of NER. In this case *W* contains the wildcards which obey the word structures. For example, in the GloWbE [4] corpus we can apply Part Of Speech (POS) tags as wildcards, which are defined by the Penn Treebank II. The drawback of these approaches is that the set *W* is pre-defined, and the number of the available semantic wildcards is often too big. For example, the Penn Treebank II label format [5] defines 21 different types of phrases. Another disadvantage is that the users (in the absence of linguistic skills) are often unable to determine the exact structure type of the query, which leads to the use of incorrectly formulated search terms, reducing precision and recall of the system.

2.2 Natural language semantic wildcards

As described above, the usage of wildcards from a pre-defined set can be cumbersome, so we further generalize the notion of semantic wildcards. The main idea of the generalization is that natural language expressions can be used as patterns. The *query* is now

$$Q = q_0 q_1 \dots q_m, \text{ where } \forall q_i \in T \cup W, \text{ but } W \equiv * T^+ *.$$

The * denotes the language which contains a single word *. In the case of natural language semantic wildcards the matching can be defined in two ways. On the one hand we can stay the above approach, that is, using NLP techniques we can analyze the syntactic structure of the query term. For example, consider the query the *cat in *the hat**. The following information can be obtained from the text by analyzing the query.



Thus, the above query can be reduced to the search with pre-defined wildcards *cat in NP* or *cat in DT NN*. For example, the strings *cat in the hat* or *cat in the rain* will match the search term, but the string *cat in the freezing rain* fits only the more general *cat in NP*, so the score for this match could be reduced. Another approach is to represent the semantic information with dependency-graphs where the nodes are the words, and the edges are the semantic dependencies between them. In this case, a test for matching means checking the edges. If the word order is indifferent, then the search term matching test is equivalent to a similarity test between the semantic graphs.

3 Implementation options for natural language semantic wildcards

For the implementation of search engine with the pre-defined wildcards there are effective solutions. Data structures called Parallel Suffix Arrays [6, 7] offer a time-efficient solution to serve queries of a much richer query language than the above defined. In the case of natural language wildcards the implementation depends on the definition of matching. If matching is based purely on comparing dependency graphs, then we found it's reasonable to represent these graphs in a Prolog database. For matching test on the grammatical constituent level the Apache Lucene full-text search system can be used. We will briefly show what problems we encountered during the different approaches and then describe experimental results. Figure 1 depicts the general approach.

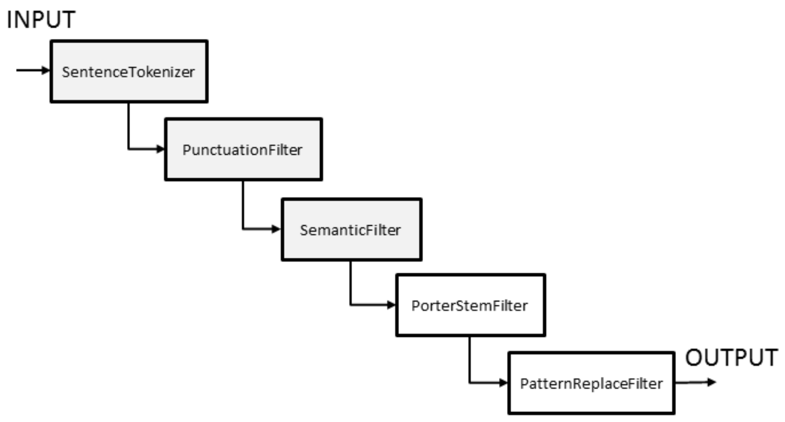


Fig. 1. Lucene analyzer chain for indexing semantic data

3.1 Natural language semantic wildcards and Apache Lucene

Lucene is a Java-based open-source information retrieval software library. Lucene provides indexing and full-text search functionality that can be built in to various software. With Lucene we can index any textual data and store it in a schema-less index. To encode the semantic information in the index the easiest possible solution is to work around the problem, and store the semantic information as Lucene tokens. So we need to write our own *Tokenizer* or *TokenFilter* classes that generate these artificial, additional tokens. The *SemanticFilter* calls Apache OpenNLP parser for the received input sentences and splits the output into tokens. The user's search query is then interpreted as a composition of Lucene *SpanQueries*. Figure 2 depicts the Lucene query for the query **somebody* will feed the *dog**.[1]

[1] In Figure 2 the notation of "*NEAR*" and "*OR*" were used for ease of clarity: they correspond respectively to the *SpanNearQuery* and *SpanOrQuery* queries. For *SpanNearQuery* the tokens matching must be in the order of the subqueries, for *SpanOrQuery* only one token matching is enough for a subquery.

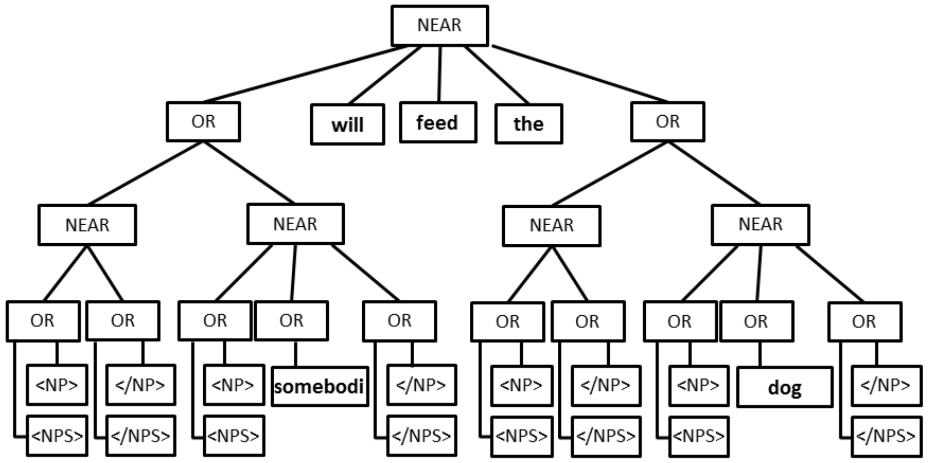


Fig. 2. The Lucene query tree of **somebody* will feed the *dog**

3.2 Natural language wildcards and Prolog language

In this section, we present a solution which makes it possible to construct indices which support the dependency approach. When we introduced the concept of dependency-based matching, it was already mentioned that in this case, dependencies extracted from the processed texts can be represented by directed, labelled graphs. The vertices of such a graph are the words (tokens) of a given sentence S and the edges are labelled with elements of dependency relations (Rel):

$D_S = \{(w_1^1, w_1^2, rel_1), \dots, (w_n^1, w_n^2, rel_n)\}$, where $rel_i \in Rel$ and $w_i^k \in T$.
In the graph there is an edge from w_i^1 to w_i^2 which is labelled by rel_i:

rel_i
w_i^1
w_i^2

In this case, directed graph D_S can be represented by a Prolog program with n rules:

```
rel($rel_1, w_1^1, w_1^2$) .
rel($rel_2, w_2^1, w_2^2$) .
...
rel($rel_n, w_n^1, w_n^2$) .
```

The Prolog representation can entrust the pattern matching to the Prolog runtime environment, as we shall later see. Of course, the above Prolog rule set represents D_S of only one sentence S of a single document (or its only one field). Since we want to index multiple documents by the search engine, and a document (in a particular field) typically contains more than one sentence, we have to make sure that the dependencies of the different sentences do not get mixed up in a document. The efficiency of a search in the Prolog runtime environment can be crucial, as in the end we will use the stored Prolog knowledge base to pass the goal clause

$$:-rel(r_0, u_0, v_0), \dots, rel(r_m, u_m, v_m), \text{where } r_i \in Rel \text{ and } u_j, v_k \in T$$

corresponding to the query $Q = q_0, \dots, q_m$, which can perform the comparison between the dependency graph of the query and of the stored sentences by evaluating the goal clause. The first idea could be that dependencies of all fields (and each block within each field) in all documents are stored in one large Prolog database. For a given field *field* of a given document *doc*, the field's sentences can be represented by the following Prolog code[2]:

```
rel($rel_{1,1}, w^1_{1,1}, w^2_{1,1}, doc, field, 1$).                    }
...                                                                       }  1. sentence
rel($rel_{n_1,1}, w^1_{n_1,1}, w^2_{n_1,1}, doc, field, 1$).              }
rel($rel_{1,2}, w^1_{1,2}, w^2_{1,2}, doc, field, 2$).                    }
...                                                                       }  2. sentence
rel($rel_{n_2,2}, w^1_{n_2,2}, w^2_{n_2,2}, doc, field, 2$).              }
...
rel($rel_{1,s_{doc,field}}, w^1_{1,s_{doc,field}}, w^2_{1,s_{doc,field}}, doc, field, s_{doc,field}$).   }
...                                                                                                      }  Last sentence
rel($rel_{n_{s_{doc,field}},s_{doc,field}}, w^1_{n_{s_{doc,field}},s_{doc,field}},$                     }
    $w^2_{n_{s_{doc,field}},s_{doc,field}}$, $doc$, $field$, $s_{doc,field}$).                          }
```

In this case, an appropriate goal clause for the query can be as follows[3]:

$$\begin{gathered} :-rel(r_0, u_0, v_0, DOC, FIELD, SENTENCE), \dots, \\ rel(r_m, u_m, v_m, DOC, FIELD, SENTENCE) \\ \text{where } r_i \in Rel \text{ and } u_j, v_k \in T. \end{gathered}$$

However, regarding the implementation of Prolog database built in this way, efficiency issues must be taken into account - especially in cases of large Prolog databases. Depending on what kind of Prolog system is used, we can optimize the evaluation time of the program in various ways. For example, if we require the text to not contain special characters or values (e.g., *Rel* elements) we can store Prolog atoms instead of strings. We have tested two Prolog systems: the SICStus Prolog [8] and TuProlog [9]. The performance with Prolog database atoms was always slightly faster than the string representation, but the biggest difference between the two representations was only 0.1203 seconds (in the case of the Bible corpus[4]). This is

2 In the Prolog code $s_{doc,field}$ is the number of sentences in the field *field* of the document *doc*.

3 The following goal clause corresponds to the query just in case if the search term is composed of a natural language sentence, the more complex cases are not discussed here.

4 We made measurements on two text documents which are available free of charge: one is Tractatus Logico-Philosophicus by Ludwig Wittgenstein, and the other one is the English Bible (Old and New Testament).

probably due to both SICStus and TuProlog represent the atoms and strings with similar efficiencies in the background. A greater acceleration was achieved by finding the right order of the terms of the clauses. However, the speed difference is imperceptible in case of small datasets (such as the Tractatus), but it can be seen that the term indexing has a great impact on the performance when dealing with large corpora.

3.3 Term indexing: the optimal term order

In the previous Prolog example, we have presented a format with which the text dependencies can be represented by the facts of the Prolog language. All such facts were of form *rel(*$\mathrm{rel_i}, \mathrm{w_i^1}, \mathrm{w_i^2}, \mathrm{doc}, \mathrm{field}, \mathrm{sentence}$*)*. The Prolog engines usually index the facts by the first term, so in this case by $\mathrm{rel_i}$. Thus, for a given $\mathrm{rel} \in \mathrm{Rel}$ producing the list of matching rules for this term will be very effective, while for the rest of the terms it[5] won't. To find the optimal term order we made some measurements. At first, one might be surprised that some orders are more efficient than others if we restrict ourselves only to query the facts of our Prolog representation and if the representation is supplemented with a few simple rules (see the next section for the rules). If we are just querying the facts and there are no rules, the result is as shown in Figure 3.

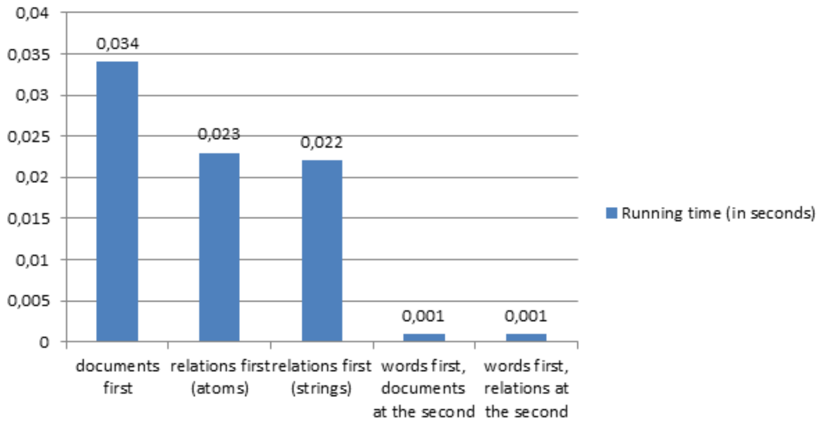


Fig. 3. Effectiveness of Prolog queries for different term orders (using only facts, a lower value means better)

[5] But we could use hash predicates in SICStus Prolog such as *term_hash* and *variant_sha1*.

On the graph the storage strategy *documents first* means the order *(doc, sentence, rel,* $word_1$*,* $word_2$*, field)*, while *relations first* means the order *(rel,* $word_1$*,* $word_2$*, doc, sentence, field)*. With the strategy *relations first (strings)* we considered the same order but the other representation of the relations was used (instead of atoms, string values were represening the relations) - the difference is negligible. *Words first, then documents* means *(*$word_1$*,* $word_2$*, doc, sentence, rel, field)*, and finally *words first, then relations* means the *(*$word_1$*,* $word_2$*, rel, doc, sentence, field)* ordering. The results shown in Figure 4 are the averages of 10 runs, for five different search terms. It's clear that if we work with only facts, then the strategy *documents first* gives the slowest method of the possible permutations[6], and indexing words is the fastest. The reason is that we have a free variable in the argument *document* in the goal clause of the query. However, we have significant changes in the case when we use inference rules in addition to Prolog facts in the search.

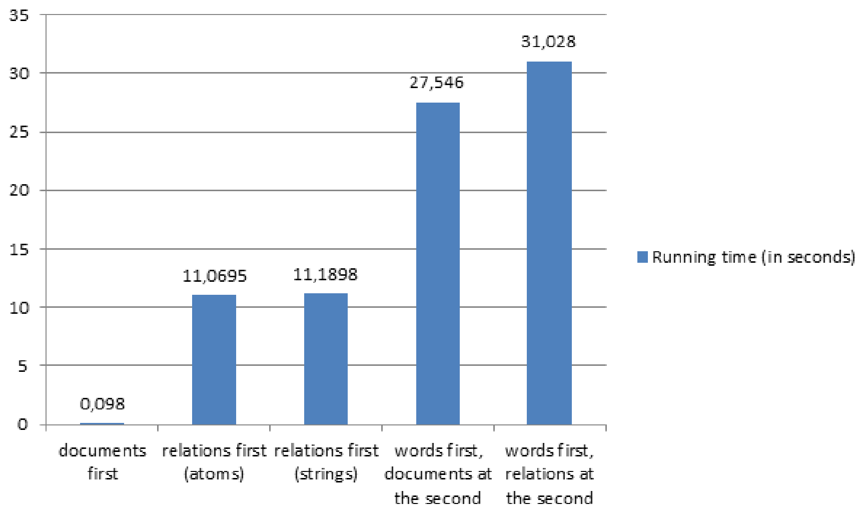


Fig. 4. Effectiveness of Prolog queries for different term orders (using facts and rules)

It's clear that the searches now are much slower than it can be seen on Figure 3 - this is due to the introduction of rules. Depending on what kinds of inference rules we work with, the obtained run times can be different from the above results. In these measurements we used two simple rules, the rules *dobj* and *prep*, as we will see later. It is also shown that the strategies that were effective using only facts for pattern matching, are much slower when rules are also used. So, if we want to store data dependencies derived from the text in only one Prolog knowledge base and inference rules are used, then it is worth using the order *documents first* - of course, all this should only be addressed if we use such Prolog runtime environment that supports

[6] The number of possible permutations is 6! / 2! = 360, or 5! / 2! = 60 if fields are omitted. However, for the most of these permutations we received very similar results. We present only those indexing options which are interesting from the point of our observations.

term indexing. Based on the measurements we can be satisfied with the effectiveness of SICStus Prolog. However, these results are given after a compilation step. The compilation is computationally rather intensive operation, for the Bible corpus it takes an average of 86 seconds (keep in mind that the Prolog representation in this case is a 24-megabyte source file). However, once that's done, then we can run fast and efficient queries on the index. The speed of queries, of course, depends not only on the term order, but the order of the terms (edges of the dependency graph to be matched) in the goal clause of the query.

Evaluation of the goal clause is sequential on its terms, so if the first term in the goal clause is too general (it means that the term matches a number of facts and left sides of rules), then the surplus calculation is accumulated for matching of the complete query, thereby matching the rest of the terms. However, if we are lucky, the first term of the goal clause corresponding to the query is as

```
:-rel(DOC,S,'mahalalel','years',conj),
  rel(DOC,S,'day','that',det).
```

This is a favorable case, because `'mahalalel'` and `'years'` are certainly less frequently together in the English Bible than '`that day`'. Therefore, in addition to the term indexing (or in its absence) also a possibility raised that with some metadata we can further increase efficiency: for example, with automatic reordering of goal clauses (because we know that in our case it will not change the operation of search), or with specifying our own preprocessing algorithms, which in the first step filters the list of the applicable Prolog rules by different, domain-specific meta-information.
As a kind of metadata-based filter, we implemented a simple in-memory inverted index in Java programming language to store the Prolog representation. For seamless integration with the Java platform, we chose TuProlog, which is a Prolog engine implemented in pure Java. The advantage is that it is available for free, and we can use our search engine without installing any Prolog runtime environment. Java integration was also necessary because we use NLP tools which are based on Java.

4 Prolog goal clauses

We have already discussed in detail the generation of Prolog codes from dependency graphs. However, it has not been presented yet, how we can construct goal clauses transferable to the Prolog runtime environment from the user queries. Of course, the first step here is to clean the search terms from the syntax of the semantic wildcards, that is, if the search query is as follows:

```
Russell *is wrong*, because *he did* when *doing something*.
```

Then the next string is extracted from the text:

```
Russell is wrong, because he did when doing something.
```

The extracted string has been stripped from the special wildcard syntax so we can pass it to the dependency parser. Next, the dependency parser from the above string generates the dependency graph of the text, thus we received n dependencies of the form $rel(type, w_i, w_j)$, where type is the type of the dependency, w_i, w_j are the words which are in that dependency relationship. Note that from each of these we can generate a Prolog term in the same form, where we substitute w_i and w_j with free Prolog variables X_i and X_j, if w_i and w_j were a part of a semantic wildcard before stripping the query string. In the previous example the output of the parser on the cleaned search term will be the following dependency graph:

```
nsubj(wrong:3,Russell:1)
cop(wrong:3,is:2)
root(ROOT:0,wrong:3)
mark(did:7,because:5)
nsubj(did:7,he:6)
advcl(wrong:3,did:7)
advmod(doing:9,when:8)
advcl(did:7,doing:9)
dobj(doing:9,something:10).
```

Since the root relationship is only a virtual dependency, we can ignore that. In the remaining dependencies on the next step we replace the word belonging to semantic wildcards with variables. If these steps are carried out then we obtain the next graph[7]:

```
nsubj(WRONG,Russell:1)
cop(WRONG,IS)
mark(DID,because:5)
nsubj(DID,he:6)
advcl(WRONG,DID)
advmod(DOING,when:8)
advcl(DID,DOING)
dobj(DOING,SOMETHING).
```

From which the corresponding Prolog goal clause can be easily prepared:

```
:- rel(nsubj,  WRONG, ”russell”  ),
   rel(cop,    WRONG, IS         ),
   rel(mark,   DID,   ”because”  ),
   rel(nsubj,  DID,   ”he”       ),
   rel(advcl,  WRONG, DID        ),
   rel(advmod, DOING, ”when”     ),
   rel(advcl,  DID,   DOING      ),
   rel(dobj,   DOING, SOMETHING  ).
```

[7] The names of the variables are written in capital letters for the sake of clarity.

But it does not solve all the problems. Indeed, in this case our too simple graph-matching test would not recognize a number of dependencies which are present in the text. Consider the following example:

```
The meaning *should play* a role in syntax.
```

Following the above, from the search expression we would get the following goal clause.

```
:- rel(nsubj,  PLAY,   ”meaning” ),
   rel(aux,    PLAY,   ”should”  ),
   rel(dobj,   PLAY,   ”role”    ),
   rel(prep,   ”role”, ”syntax”  ).
```

Unfortunately the following sentence does not match the above goal clause:

```
In logical syntax the meaning of a sign should never play a
role.
```

Since Prolog representation of the above sentence is the following:

```
rel(amod,"syntax","logical",1).
rel(prep,"play","syntax",1).
rel(det,"meaning","the",1).
rel(nsubj,"play","meaning",1).
rel(det,"sign","a",1).
rel(prep,"meaning","sign",1).
rel(aux,"play","should",1).
rel(neg,"play","never",1).
rel(root,"root","play",1).
rel(det,"role","a",1).
rel(dobj,"play","role",1).
```

The terms filling into the terms of above goal clause are written in bold - the error is in the underlined row. The parser recognizes the prepositional structure, but assigns it to the verb “play”, which is a reasonable choice, but we do know, however, that if there is a direct object of a verb, then the prepositional relation can be extended to this object as well. This is expressed by the following Prolog rule:

```
rel(prep, X, Y, S):- rel(dobj, Z, X, SCORE1),
                     rel(prep, Z, Y, SCORE2),
                     S is SCORE1*SCORE2*0.5.
```

Now we can see how the scoring is done in the Prolog representation: the score of the document (or more properly, the score of a given sentence in the given field of the document) is calculated by Prolog facts and rules. Each application of the above rule reduces the total score results with a factor λ (which has the value 0.5). This prevents the inferred dependencies by the rules to be equal to those documents that can be matched without applying any rules. Another rule, which we found useful is:

```
rel(dobj, X, Y, S):-rel(ccomp, X, Y, SCORE1),
                    S is SCORE1*0.5.
```

Finding the weights λ could be a part of a separate optimization problem in the future. However, we developed an alternative method of scoring, which prevents any infinite cycles. Thus, it is useful for non-tree-based representations of the *Stanford* parser:

```
rel(DOC, SENTENCE, X, Y, dobj, INH, SYNT)
  :- INH > 0.1,
     rel(DOC, SENTENCE, X, Y, ccomp, INH*0.5, SYNT).

rel(DOC, SENTENCE, X, Y, prep, INH, SYNT)
  :- INH > 0.1,
     rel(DOC, SENTENCE, Z, X, dobj, INH*0.5, S1),
     rel(DOC, SENTENCE, Z, Y, prep, INH*0.5, S2),
     min_list([S1, S2], SYNT).
```

The general form of the rules can be seen from the code. Every rule gets an inherited score (INH), and produces a synthesized one (SYNT). If the inherited score falls below a certain threshold (which is determined in 0.1 in the above example), then the dependency derivation tree gets discarded and the evaluation continues. If there are multiple terms on the right side of the rule, then we take the minimum of the synthesized scores, as they are already reduced by the factor of λ. In both cases, the final score will be λ^n, but the value of n is the number of interior nodes of the derivation tree in the first case and is the depth of the tree, in the second case. The latter is typically a smaller number for rules with more right-hand terms. So, in the end we get a more stable and more accurate scoring logic.

5 Experimental results

The effectiveness of information retrieval systems are commonly measured by two metrics: precision and recall. The aim of information retrieval which is to maximize both of them could be difficult, because these two metrics often work against each other - the lower the first one, the higher the second, and vice versa.
Solutions using Lucene are not sufficient neither from precision nor recall point of view (both of them were approximately 0.1) - the root of this problem is the representation: in Lucene, we had to store the retrieved syntactic or semantic information as ordinary Lucene tokens, so the retrieved metadata is stored at the same level or layer as the original text. This is a problem because positionally nearby tokens can fall apart from each other after processing. On the following figure, we can see the resulting TokenStream for the string cat in the hat:

```
<NP> <NP> <NN> cat </NN> </NP> <PP> <IN> in </IN> <NP>
<DT> the </DT> <NN> hat </NN> </NP> </PP> </NP>.
```

In the original text, the distance between the cat and the hat words were only two tokens. After processing, it is increased to 11. And because of the distance is directly proportional to the depth of the parse tree, it is impossible to determine an upper limit on Lucene SpanNearQuery's slop parameters.

Unlike the Lucene-based solution, the implementation in Prolog produced really good results: on the same dataset we can achieve 0.8 precision and recall, not to mention the fact that the TuProlog-based engine was also faster than the Lucene solution, as shown in Figure 5.

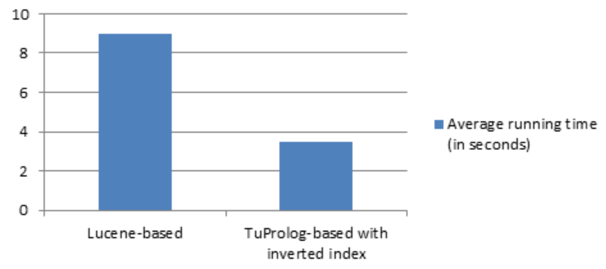


Fig. 5. Average running times of the Lucene-based and the TuProlog-based solution

The reason of the relatively high execution time (as compared to the results of SICStus Prolog) is that the parser model (a file with a several-megabyte size) must be loaded into the memory in each case at the beginning of the program. If we analyze the distribution of the execution time of the TuProlog-based solution, the bottlenecks can be seen in Figure 6.

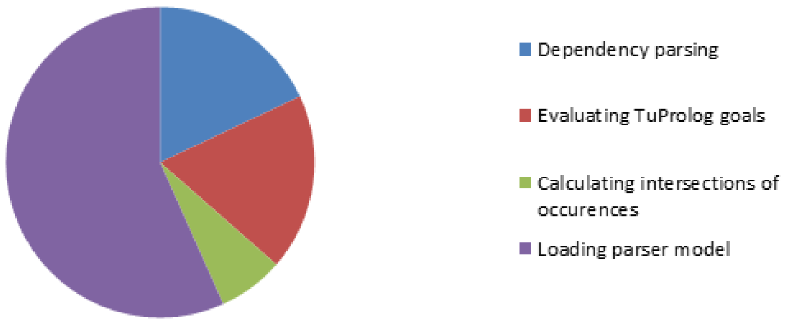


Fig. 6. The average distribution of the execution time of the TuProlog-based solution

6 Conclusions

We have presented two implementation plan for the semantic wildcard search. The implementation has shed some light on practical problems. Namely, it turned out that in the current version of Lucene (4.3, at the time of the writing) is not possible to store complex meta-information in a parallel layer without clumsy workarounds, so this means that implementing a semantic wildcard search engine in Lucene would be inherently sub-optimal and the outcome would be unsatisfactory. However, the next version of Lucene can make a positive difference in this topic: as we can read in [10], the Lucene attribute API already contains an attribute named PositionLength, which is in principle could make Lucene capable to store word lattices. However, the overall infrastructure of the Lucene has not support this attribute yet, however this may change in the future, enabling Lucene to support complex wildcard searches and to be a viable alternative for implementation of a semantic wildcard search engine.

It was surprising to see that the Prolog environments are capable to store large amount of linguistic data (e.g., the Bible corpus), and to be a basis of a full-text search application.

The rearrangement of terms in the auto-generated goal-clauses would be an interesting goal for further development, as well as the optimization of the λ vector or the development of additional document scoring methods or the complete integration with the SICStus Prolog.

7 References

1. Barry Morley, Antoinette Renouf, Andrew Kehoe: Linguistic Research with the XML/RDF aware WebCorp Tool, WWW2003 Conference, Budapest, 2003.
2. KWiCFinder: http://www.kwicfinder.com/KWiCFinder.html, 2013. 06. 08.
3. The Semantic Vectors package: https://code.google.com/p/semanticvectors/, 2013. 06. 08.
4. GloWbE : Corpus of Global, Web-based English:http://corpus2.byu.edu/glowbe/, 2013. 06. 08.
5. Penn Treebank II: http://www.cis.upenn.edu/~treebank/, 2013. 06. 08.
6. Johannes Goller: Parallel Suffix Arrays for Corpus Exploration. (2010).
7. Johannes Goller: Parallel Suffix Arrays for Linguistic Pattern Search - http://www.aclweb.org/anthology-new/R/R11/R11-1068.pdf 2013. 06. 08.
8. SICStus Prolog: http://sicstus.sics.se/, 2013. 06. 08.
9. TuProlog: http://tuprolog.alice.unibo.it , 2013. 06. 08.
10. Michael McCandless: Lucene's TokenStreams are actually graphs, http://blog.mikemccandless.com/2012/04/lucenes-tokenstreams-are-actually.html, 2013. 06. 08.
11. Rada Mihalcea: The semantic wildcard. *Proceedings of the LREC Workshop on Creating and Using Semantics for Information Retrieval and Filtering State of the Art and Future Research*. 2002.
12. Tony Veale: Creative language retrieval: A robust hybrid of information retrieval and linguistic creativity. *Proceedings of ACL*. 2011.

Designing and Implementing Control Flow Graph for Magic 4th Generation Language

Richárd Dévai, Judit Jász, Csaba Nagy, Rudolf Ferenc

Department of Software Engineering
University of Szeged, Hungary
`devai@frontendart.com`, `{jasy|ncsaba|ferenc}@inf.u-szeged.hu`

Abstract. A good compiler which implements many optimizations during its compilation phases must be able to perform several static analysis techniques such as control flow or data flow analysis. Besides compilers, these techniques are common for static analyzers to retrieve information from the code for example code auditing, quality assurance, or testing purposes. Implementing control flow analysis requires handling many special structures of the target language. In our paper we present our experiences in implementing control flow graph (CFG) construction for a special 4th generation language called Magic. During designing and implementing the CFG for this language we identified differences compared to 3rd generation languages because the special programming technique of this language (e.g. data access, parallel task execution, events). Our work was motivated by our industrial partner who needed precise static analysis tools (e.g. for quality assurance or testing purposes) for this language. We believe that our experiences for Magic, as a representative of 4GLs might be generalized for other languages too.

1 Introduction

Control flow analysis is a common technique to determine the control flow of a program via static analysis. The outcome of this analysis is the Control Flow Graph (CFG), which describes the control relations between certain source code elements of the application. The CFG is a directed graph: its nodes are usually basic blocks representing statements of the code that are executed after each other without any jumps. These basic blocks are connected with directed edges representing the jumps in the control flow. CFG is a useful tool for code optimization techniques (e.g. unreachable code elimination, loop optimization, dead code elimination). The first publications of using control flow analysis goes back to the 70s [1] and 80s [4,11,24], but since then most of the compilers have implemented this technique to construct a CFG and implement optimization phases based on it.

Although the basic structure of the CFG is quite common, the methods constructing it for applications are very much language dependent. Identifying control dependencies in special structures of the target language may result special algorithms. Moreover, special program elements or applications may require minor modifications of the structure of the CFG (e.g. nodes like entry nodes).

In our paper we present our experiences in implementing Control Flow Graph construction for a special language called Magic. This language is a so-called 4th generation language [?] because the programmer does not write source code in the traditional way, but he implements the application "at a higher level" with the help of an application development environment (Magic xpa). This special programming technique has many differences compared to 3GLs which are the most common languages today (Java, C, C++, C#, etc.). Because of the philosophy of the Magic language we had to revise traditional concepts like program components, expressions and variables during the design of the CFG.

The main contributions of this paper are (1) a technique to implement a CFG for applications developed in Magic xpa, (2) identified differences of implementing a CFG in a 4GL context compared to 3GLs.

Our work was motivated by our industrial partner who needed a tool set which was able to perform precise static analysis for code auditing and for test case generation purposes. Our experiences for Magic, as a representative of 4GLs could provide a good bases to implement CFG construction for other 4GLs too.

2 Related work

Control flow is a widely used information for example in compiler programs of 3GLs. The method of CFG construction is well defined [18]. We need to discover and identify the statements, and define basic blocks by selection of leader statements. Key parts are to define the structures to handle control passing, and elements for those items of logic which are implicitly influence the behavior of control flow.

Control flow analysis has many uses, such as program transformations or source code optimizations of compilers[1] [12], rule checkers of analyzer tools [6,7,22], security checkers [5], test input generator tools[2] [28], or program slicing [26]. Program dependence analysis approaches are also based upon control dependencies computed by control flow analysis [10].

The implementation of the control flow analysis might differ for different languages. There are many papers published about dealing with higher-order languages (e.g. Scheme), for instance the work of Ashley et al. [2] and the PhD thesis of Ayers [3] both summing up further works too [11,24]. An extensive investigation has been done for functional languages too, which was recently summed up by Midtgaard in a survey [17].

However, CFG solutions for 4GLs are really limited. These work usually tackle the topic from the higher abstraction level of the language. E.g. ABAP, the programming language of SAP is a popular 4GL and there are few published flow analysis techniques which mostly deal with workflow analysis [14,27]. In previous work [20] we implemented a reverse engineering tool set for Magic and we found a real need to adapt some of these techniques to the language. Besides our work, Magic Optimizer[3], as a code auditing tool also shows this requirement.

[1] GCC Internals Online Documentation: http://gcc.gnu.org/onlinedocs/gccint/
[2] Prasoft Products: http://www.parasoft.com/jsp/products.jsp
[3] Magic eDeveloper Tools Homepage: http://www.magic-optimizer.com/

This tool checks for violations of coding rules ("best practices"), and it is able to perform optimization checks and further analyses to give an extended overview of every part of a Magic applications.

3 Specialties of a Magic Application

In the early 80's Magic Software Enterprises[4] introduced a new 4th generation language, called Magic. The main concept was to write an application in higher level meta language, and let an application generator engine create the final application. A Magic application could run on popular operating systems such as DOS and Unix, so applications were easily portable. Magic evolved and new versions of Magic have been released, uniPaaS and lately Magic xpa. New versions support modern technologies such as RIA, SOA and mobile development.

The unique meta model language of Magic contains instructions at a higher level of abstraction, closer to business logic. When one develops an application in Magic, he actually programs the Magic Runtime Application Environment (MRE) using its meta model. This meta model is what really makes Magic a Rapid Application Development and Deployment tool.

Magic comes with many GUI screens and report editors as it was invented to develop business applications for data manipulation and reporting. The most important elements of Magic are the various entity types of business logic, namely the data tables. A table has its columns which are manipulated by a number of programs (consisting of subtasks) binded to forms, menus and help screens. These items may also implement functional logic using logic statements, e.g. for selecting variables (virtual variables or table columns), updating variables, conditional statements.

The main building blocks of a Magic application are defined by repositories. These repositories construct the workspace of a Magic xpa application. For example in the *Data Sources* repository one can define Data Objects. These are essentially the descriptions of the database tables. Using these objects Magic is able to handle several database server types. The logic of an application is implemented in programs stored in the *Programs Repository*. Programs are the core elements of an application. These are executable entities with several sub tasks below them. Programs or tasks interact with the user trough forms to show the results of the implemented logic. Forms are also part of the tasks or programs.

Developers can edit a program with the help of the different views. The main views are the followings:

Data View. Declares which Data Objects are binded to the programs. The binding is in general some variable declaration, where the declaration can be real or virtual. The real declaration connects to a data table column, while the virtual declaration stores some precomputed data.

Logic View. Defines the *Logic Units* of the program. During the execution each task has a predefined evaluation order so-called execution levels. *Logic Units*

[4] http://www.magicsoftware.com

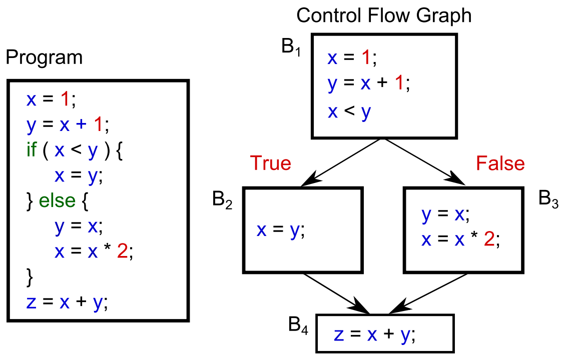


Fig. 1. CFG of a simple conditional.

are that parts of the task which handles the different execution levels. E.g. the *Task Prefix* is the first *Logic Unit* which will be executed to initialize the task. Actually the *Logic Unit* is the place where the developer can write the "code". Here we can define statements to perform calculations, manipulate data, call sub tasks, etc. Statements appear as *Logic Lines* in the *Logic Unit*.

Form View. Defines the properties of a window (e.g. title, size and position). Elements of a window can be typical UI elements such as controls or menus. A window is represented by a Form Entry in Magic xpa. In the Magic xpa development environment we can use many built-in controls or we can also define our custom controls.

As it can be seen now, a Magic 4GL application differs from programs developed in lower level languages. The developers can concentrate to implement the business logic in a predefined layered form, and the rest is handled by the Application Platform.

4 Control flow graph construction

In this section we discuss the main definitions and steps of the control flow creation of 3rd generation languages and introduce the problems of the control flow graph construction of the Magic as a representative of 4GLs.

4.1 Definitions and general steps

The **control flow graph** is a graph representation of computation and control flow in the program, as it is represented by the example of Figure 1. The nodes of a CFG are basic blocks represented by rectangles. Each basic block represents a set of statements which execute after each other sequentially. Branching can only exist at the end of the block, after the execution of its last encapsulated statement.

The first step in the control flow creation is to determine the starting points of the basic blocks [18]. These statements called leaders are the followings:

– the first statement of the program,

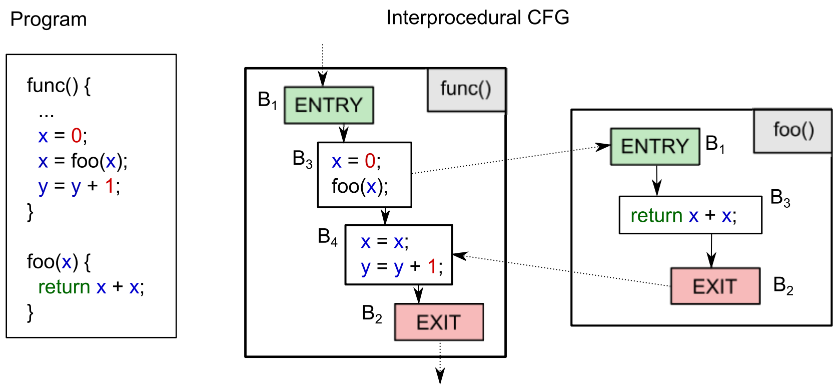


Fig. 2. Example ICFG.

– any statement that is the target of a conditional or unconditional branch statement,
– any statement that immediately follows a conditional or unconditional branch statement,
– any statement that immediately follows method invocation statement[5].

If we know the sequence of the statements in the program and the leaders of the basic blocks, we can determine the blocks by the enumeration of the statements from one leader up to but not including the next leader or the end of the program. Compilers and other source code analyzer tools first build up an intermediate program representation of the source code, called abstract syntax tree (AST), that implicitly describes the sequence of the statements. With the traversal of the AST representation we can determine the sequence of the statements, and if we want to build the control flow with much more granularity, we can determine the evaluation order of the expressions of the statements too. We will discuss finer representations under the examination of Magic expressions and call types in Section 5.

In general, control flow information of methods, procedures or subroutines of the program are represented individually. From technical reasons each of these has two special kinds of basic blocks. The *Entry* block represents the entering of a procedure, while the *Exit* block represents the returning from the called procedure. The potential flows of control among procedures are represented by call edges. The connected control flow graphs of the procedures with the call information give the so-called interprocedural control flow graph (ICFG) of the analyzed program. Figure 2 shows an example of the ICFG, where call edges are represented as an arrow headed dashed line between call site and the *Entry* block of the called procedure. In some cases the detection of procedure boundaries is not an easy task, and the target of a call or branch instruction cannot be determined unambiguously. The earlier situation is commonly appears in binary codes [13], while the later is typical in the presence of function pointers or virtual

[5] Method invocations should not be basic block boundaries in all cases only if we need compute some summarized information at the call sites in our connected application.

function calls at higher level languages. The problems appeared in 4th GLs are discussed in the rest of this section.

4.2 Challenges in Magic

Like compiler programs or other software analyzer tools, our first step is to create an intermediate representation of a Magic applications, called Magic Abstract Syntax Graph (ASG), which is suitable to describe all necessary information for our purpose. Information extracted and stored in the ASG is defined by the Magic Schema [19]. This format allows us to traverse and process every required elements of the Magic application in a well-defined hierarchical graph format through an API to determine the execution order of the Magic statements. ASG contains not only the nodes of the code, but every needed attributes that can affect the control flow. E.g. it contains the propagation information of `Event Handlers`, which can terminate the execution of other event handlers, or the wait attribute of `Raise Event`, which attribute determines the execution point of the given event.

Developing an application in Magic requires a special way of thinking since the programming language is special itself. However this special programming language preserves some main characteristics of procedural languages. Mostly, the main logic of an application can be programmed in a procedural way via control statements in programs and their subtasks. Programs can call each other and they can call their subtasks. Also, tasks can use variables for their computations, and they can have branches within their statements. These structures of the language make it possible to adapt the CFG construction of 3GLs to Magic 4GL. For example for every potential target of call sites of Magic (task, event handler, developer function) we make an intraprocedural control flow graph and we connect these graphs by call edges to get the ICFG. However, there are a number of structures in the language which make harder to construct the CFG of an application. Here we discuss these challenges which we are going to elaborate in later sections.

Tasks architecture is a special event based execution level system. There are different task types for different operations. For example online tasks to interact with user, or batch tasks running without any user interaction. Each task type has its own levels (e.g. task, record) and the developer can operate with these by the so-called *Logic Units*. A user action or a state change in the program can trigger predefined events that are also handled by *Logic Units* of tasks. So statements (*Logic Lines*) of these *Logic Units* get executed if a certain event triggers them. The most challenging thing to construct the CFG of a Magic program is to discover every circumstances that can change the flow of the control between *Logic Units* and between *Logic Lines*. We have to understand and represent the effect of property changes which can influence the behavior of the execution, and represent it in a well describing form.

Raise Event logic lines and **Event logic units** are components of Magic logic to raise an event during the execution of the program and to handle the raised event. A raised event could be handled with special logic units called *Event*

Logic Unit in a special predefined reverse order in tasks. When an event raised, the MRE immediately looks for the last available handler in the given task, and gives the control to the handler. This is the simplest case, the synchronous case. However, we could raise events asynchronously; or set the scope of handlers as they could be handled by parent tasks too, or only by the task which raised it; or every matching handler could terminate the chain of handlers if propagate property is set to no; etc. Describing the proper event handler chains within the CFG requires a complex traverse of logic units in task hierarchy with respect to the influencing attributes. Our model is limited those events which are raised by a code element or a form item.

Data access is supported with a rich toolset in Magic to access databases for efficiency. Magic provides gateway to wide scale of RDBMS systems by handling connection, transactions and generation of SQL queries, beside we could create our own queries. In general we can select from two alternatives to perform our transactions. In Physical mode other DB users see our changes in RDBMS log, and we use the lock system of the DB server. In Deferred mode the Magic xpa is responsible to store our changes and commit them when we have assembled our transaction within a running task. Beside transactional modes we have to select the method of update process for the records we use in the transactions. Different strategies give us opportunity to handle concurrency and integrity on record updates. At the creation of the CFG we have to handle the different event handlers dependent from the selected transaction mode and update strategy.

Parallel task execution makes it possible to execute more programs in parallel. Parallel programs are running in an isolated context where every loaded components of main application are reloaded within the new context. In such a context a parallel program has its own copy of memory tables, own database connections with some limitations (e.g. it cannot store data in main program or communicate directly with other running programs). Tasks can raise asynchronous events in the context of another program to communicate, or they can use shared variables through proper functions in expressions. Parallel processes can run in Single or Multiple instance modes. In Single mode the context is the same for every instance of the task, while Multiple mode uses different context for each task. At the CFG construction we have to simulate all hidden data base copying and the parallel execution of statements.

Forms has many uses during a program execution. In each case we have to build the CFG according the current use of the forms. On the forms the user can manipulate variable data, which appear in the running program as assignment instructions, or the user can affect the running program behavior too.

5 Implementation details

As we seen the process of CFG building is aggregated from several phases. First by the traversal of the ASG we determine the sequence of statements and the evaluation order of expressions. During evaluation we collect information about calls. After we determine basic block leaders and finally we build up the basic blocks for later processes. In our representation each call site will be a block boundary.

To determine the execution order of the contained statements of an analyzed code, we traverse its ASG from the root node step by step on the tree hierarchy and we refine the control flow information among the sub components. In every steps we define the execution order of the composed nodes of an investigated ASG node and we augment the execution sequence with additional expressions or statements, if it is needed. We do this since many semantic elements of a programming language is not appear explicitly in the source code and so in its ASG representation. Due to the hierarchical traversal, the control flow information of descendant nodes are refined after the traversal of their ancestors.

Rectangles of figures of this section represents nodes, or groups of ASG nodes. Parallelograms denote branches where the possible flow of control depends on an attribute of `Logic Units`, `Logic Lines`, controls, variables, etc. Black arrows denote control edges of the CFG, while dashed lines represent call edges among the intraprocedural CFG components. Since in our representation call instructions are basic block boundaries we represent each call with two virtual nodes called `Call Site` and `Return Site`. In some cases we introduce solutions of alternative program versions with the help of one figure. To distinguish the differences of these versions we use black branching points on the paths where the behavior of the different versions are differ.

In the following we discuss cases where we could create general algorithms to process group of nodes with the same base type. Finally we introduce some special solutions where general algorithms are not able to describe precisely the real evaluation order of the analyzed ASG node descendants.

5.1 General algorithms

Tasks in the ASG represent either programs or their sub tasks. The final representation of a `Task` is influenced by the implementations of the contained `Logic Units`, and the used variables, but we have to concentrate only the skeleton of the tasks, since the finer control flows of `Logic Units` are determined in later steps of the traversal.

When we reach a `Task` node in the traversal first we create an intraprocedural CFG context for the `Task` node. Our second step is to collect the ordered sequence of logic units that take part of the execution progress of the task. These nodes are the child nodes of the `Task` node in the ASG. `Task`, `Group`, `Record` are subtypes of the `Logic Unit`, but of course the existence of these elements are only optional in each `Task`. `Prefix` and `Suffix` are sub categories of previous `Logic Unit` subtypes controlled by an attribute. The subtype and the selected attribute value determine the exact execution point and order of these `Logic Units`. So we nominate the distinct `Logic Units` with different types and attributes differently as in Figure 3.

We does not connect every `Logic Unit` subtypes in this step, only the `Task`, `Group` and `Record`. For the `Event` and `Function` subtypes of `Logic Unit` we associate a distinct intraprocedural CFG and handle them separately since this kind of `Logic Units` can be triggered several times from distinct points.

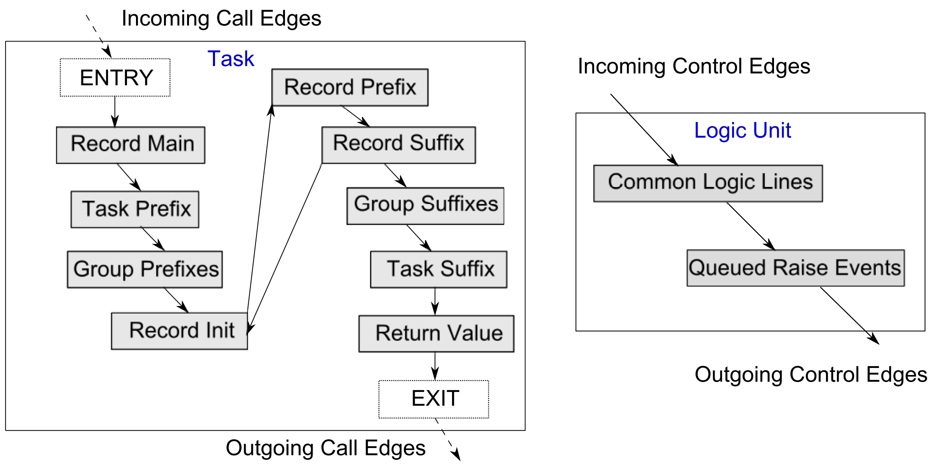


Fig. 3. Evaluated control flow of a `Batch Tasks` and `Logic Units`.

Generated source codes and behaviors of MRE are differ at some point from the structure that we can see in Magic xpa Studio during developing a `Task`, because variable declarations and initializations are also part of the execution of logic, but defined in a separated view as we shown in Section 3. The creations of variables and default value assignments have been done at the start point of a task execution. These commands are gathered by the `Record Main` node.

While the `Task` and `Group` logic units have only two subcategories, the `Prefix` and the `Suffix`, the `Record` logic units logically have three distinct in a loop of control. Each `Record` logic units execution round could have an initialization part what explicitly does not appear in the code. Since it has an important effect to the control flow, we insert a virtual `Record Init` node into the flow of execution. If we does not find any initialization during the investigation of variables in the traversal of the record unit, or the task is not in 'write' mode and the initializations use real variables only we can delete this `Logic Unit` from the CFG at the end of the traversal of the `Task`. In the last step we investigate the return expression node of the `Task`, and if it exits we connect it as the last item before the `Exit` block of the `Task`.

On the left side of Figure 3 we can see the execution order of a `Batch Task` or a `Browse Task`. These tasks contain variables, implement all possible `Logic Unit` types and define a return expression.

Having visited all the contained nodes of the `Task` node, we are able to build up the basic blocks and determine the control and call edges among these elements, since we known the exact execution order of the contained statements, expressions.

Each **Logic Units** consist of `Logic Lines`. Generally `Logic Lines` have two distinct kinds. In the first kind the execution of the logic lines are not dependent from any other factor; we handle them as they can run sequentially in the order of appearance until further checks. We refer these as `Common Logic Lines`. In the

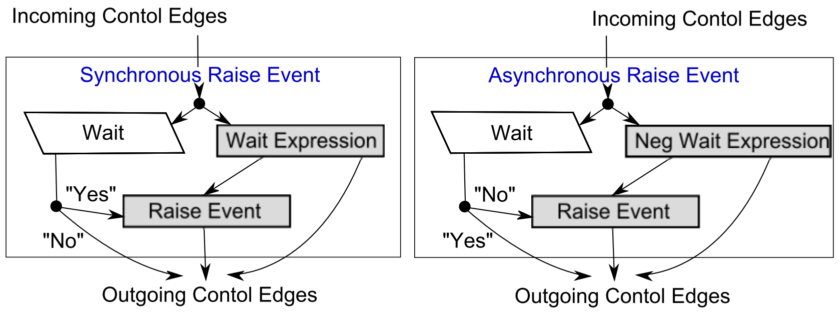


Fig. 4. Control flow of `Raise Events`.

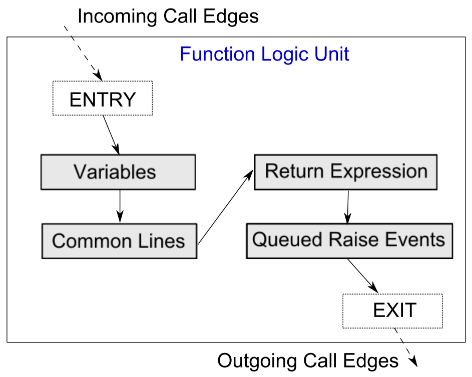


Fig. 5. CFG of Function Logic Unit.

second kind of `Logic Lines` we have to observe the wait attribute. From the so called `Raise Events` nodes we determine the asynchronously executed `Queued Raise Events` according to the Figure 3 if the value of the wait attribute is false. The wait attribute of a `Raise Event` node can be a 'yes' or 'no' constant or a boolean expression. Since the execution places of these lines are dependent from the value of the wait attribute, we have to distinct cases. If this value is logically true, we can speak about synchronous raise events, while in the other case we can speak about asynchronous raise events. In the case where the wait value is given by an expression, we have to explicitly sign the two possible cases in the control flow with additional conditional branches as it is shown by the Figure 4.

The execution of `Logic Lines` are dependent in general from a condition which can allow or decline the execution of the certain line. If the given line get right to run, the flow of control get into the statement, which describe the exact behavior of the logic line. Although this part of the evaluation of the logic lines is general, the behavior of the distinct subtypes of `Logic Lines` can be very different as we can see in the next section.

5.2 Specific algorithms

As it was mentioned in the last subsection the **Function** and **Event Logic Unit** nodes are different from other logic units, but similar to each other. Since the execution of these units are dependent from their context, and their execution

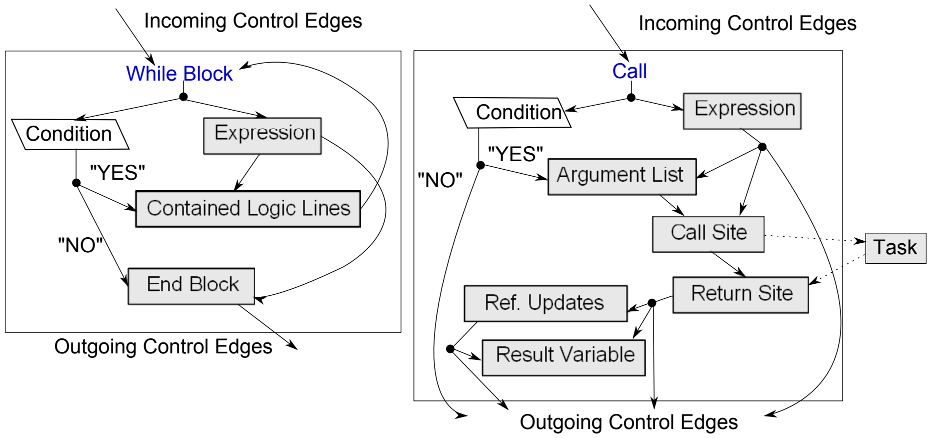


Fig. 6. CFG of a While block and a general Call logic line.

is triggerable by different point from the program, it is better to handle them similar as the `Task Unit`. So we create for these nodes an own intraprocedural CFG representation, which are callable from distinct program points. Next we collect `Logic Lines` which are variable declarations from their contained `Logic Lines`, because they are not necessarily be in order before all other `Logic Lines`, but executed collectively at the beginning of the execution of the `Logic Unit`. Next we have to perform an algorithm like shown for `Logic Units`. The difference between `Function` and `Event Logic Units` is that former could define a `Return Expression` declared by an attribute of `Logic Unit`, what is executed before `Queued Raise Events` as it is shown by Figure 5.

Logic Lines are evaluated through the traversal by specific evaluators. These elements of logic are much more unique from the point of view of the control flow processing than the `Tasks` and `Logic Units` are. We introduce some of these to show the variety and the complexity of their processing.

A `Block` node is implemented by a `Logic Line` pair. A **While Block** with its related `End Block` declare the start and the end of the `Block`. These two encapsulate the body of the `Block`. When we find a `While Block` in the ASG we have to search for its terminating `End Block` node, because they are not connected directly in the ASG. The condition for a `While Block` can be a 'yes' or 'no' constant or an `Expression`. Nesting of `Block` nodes makes harder to carry out this task. Left hand side of Figure 6 shows the evaluation of a while structure. The elaboration of **If Block** is similar to the `While Block`. First we have to find the corresponding `End Block` and `Else Blocks` for each `If Block` node. The multiple selection is implemented by the optional condition argument of an `Else Block` node.

The right of Figure 6 shows a **Call** logic line, which implement a call based on a Magic generated identifier of a program, a sub task or by a public name, etc. A `Call` logic line node has an optional argument list and could receive a return value. The passed-by-reference arguments are updated after the control is give back to the `Return Site`. To implement this behavior in the CFG we have to create update nodes for them. Before the actual call, we insert a `Call Site` node into the CFG, while after the execution of the `Exit Block` of the called CFG we nominate the return with a `Return Site` node.

Select Logic Lines defined on the Data View are separated from the code. The code representation refers to this elements only by identifiers. The semantics of these `Select Logic Lines` should appear by the `Record Main` and `Record Init` nodes during the execution of a given task. The handling of the expressions of `Select Logic Lines` are similar as the normal `Logic Line` types.

All **Expressions** of Magic are arranged into subtypes by categories in our ASG representation. An `Expression` can be unary, binary operations, Function Calls that refer to a built-in function or a `Function Logic Units` and literals. `Literals` can make a reference to an identifier, a resource or a component, or it could contain a constant value. Control flow of a `Function Call` can be built-up as a simpler `Call Logic Line`, the only difference is that its arguments can not passed by reference.

6 Evaluation

Finally we have made a verification of our technique through result validations and performance tests.

Our application has been developed in C++. We have created 105 test cases with Magic xpa Studio and the ASG have produced with the ASG generator application made by Szeged Software Ltd[6].Our work allows C++ applications to consume our CFG algorithm as a library. Validation progress based on the specification of CFG library, the output of the ASG builder in XML format, textual log output, and dump of CFG builder in graphML. Finally we have created a simple batch script to control the test execution progress. First we have created an ASG representation of the analyzed Magic application. The computed ASG is in binary format, but for manual validation we can dump its content in XML too. Having computed the ASG we determine the ICFG of the analyzed program. In the first case we compared the graphML CFG dumps of the individual tasks with the original code, and we verified our computations manually. Finally we rerun the CFG computation without any logging steps to simulate a real-life situation to gather runtime information about our algorithm.

An exported picture from graphML can be seen in Figure 7. The original code contains an infinite `While Block`. This information is shown on the figure too, where basic block with id 4 is unreachable. This information could be easily retrieved by API calls during the traversal of the CFG. Of course in this case this possibly malformed control structure is recognized by Magic xpa Studio too, it warns the programmer about the existence of the infinite loop. The example of the figure contains a call from the body of the `While Block`. This call is also appear in our ICFG. We have compared all the resulting dumps with original source codes manually, and we find each ICFG gives a good description from possible execution pathes of the original codes.

[6] http://www.szegedsw.hu

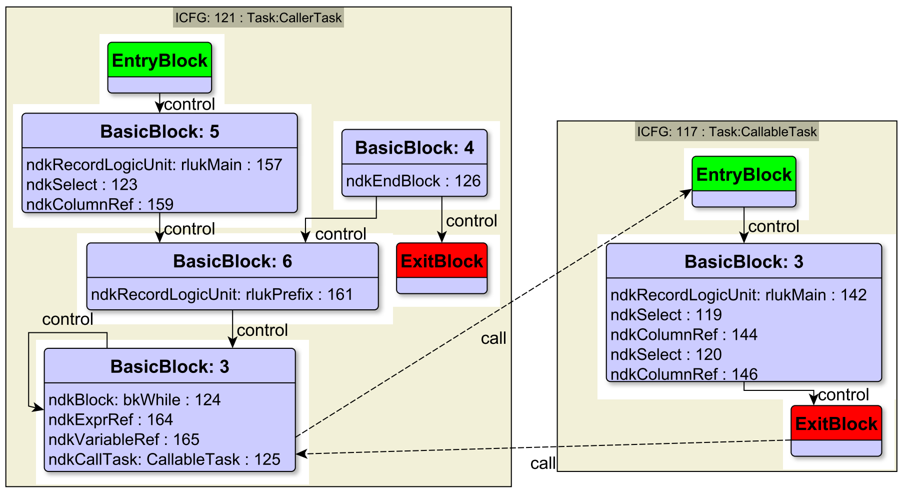


Fig. 7. Visualized ICFG by generated graphML dump.

To verify the usability of our algorithm we ran our implementation on an Intel XENON E5450 @ 3GHz 32 GB Windows Server 2008. Performance results on a medium sized sample project with nearly 200.000 nodes and about 500.000 attributes we get a 0,598 seconds runtime of the ICFG computation. As we can see the ICFG computation is carried out in an affordable time, and so it will be adaptable in any approaches based on this information.

7 Limitations of the approach

Beside the shown advances of our technique there are a few limitations too. Here we describe two main limitations among others.

Our event handling does not handle all the possible specialties of a Magic application. Currently, the implementation is able to follow events that are raised and handled inside the code with a raise event statement or a certain logic unit.Internal events of Magic xpa (such as hotkeys) are not yet supported unless raised by raise event statement.

Our recent CFG model does not support the representation of parallel task executions given by section 4.2. To improve our model, we should investigate previous work about limitations and possible application forms of CFG for parallelism support e.g. [?].

8 Summary and Future Work

In our paper we present an application of CFG concepts for a specific 4th generation language, Magic 4GL. We use a static analysis approach to gain information from generated Magic source code, and build a CFG with fine granularity. We create a reusable library for further use of our model which makes it possible to perform further analyses and process the CFG and ICFG structures which we created. We created a textual and an XML based graphML dump to make it easy to get an overview of the processed information.

Our evaluation shows that the implemented approach is applicable for middle-sized Magic applications. The presented method has an affordable space requirement and it constructs the CFG fast enough to analyze large projects too.

Besides, we show that implementing control flow analysis for a higher-level language, such as Magic, is possible via adapting 3GL techniques, but special structures of the language may result in special methods and special structures in the CFG. For example, the special use of Events enables us to gather more precise information compared to 3GLs where these structures are mostly dynamic.

Conceptually, the presented technique can be applied to other 4GLs too. The core elements of the CFG should be the same in a language independent way, but special constructs of the language should require special solutions. Particularly for other, higher level languages such as 5GLs.

Control flow analysis is just one step for us towards a more complex approach, where we plan to gather information about the available control paths, and generate automatic test cases to support testing and maximize the test coverage of Magic applications [8].

Although we have not yet implemented a specific application which is based on our CFG model, the presented approach and results with our cost measurements are already promising, hence useful for further analyses techniques.

Acknowledgements

This research was supported by the Hungarian national grant GOP-1.1.1-11-2011-0039.

References

1. Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.
2. J. M. Ashley and R. K. Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 20(4):845–868, July 1998.
3. Andrew Edward Ayers. *Abstract analysis and optimization of Scheme*. PhD thesis, Cambridge, MA, USA, 1993. UMI Order No. not available.
4. P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
5. Anupam D., Somesh J., Ninghui L., D. Melski, and T. Reps. Analysis techniques for information security. *Synthesis Lectures on Information Security, Privacy, and Trust*, 2(1):1–164, 2010.
6. Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Fact Extraction and Code Auditing with Columbus and SourceAudit. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, page 513. IEEE Computer Society, September 2004.
7. Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, October 2002.

8. Dániel Fritsi, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. A layout independent GUI test automation tool for applications developed in Magic/uniPaaS. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools (SPLST 2011)*, pages 248–259, 2011.
9. S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *SIGPLAN Not.*, 24(7):28–40, June 1989.
10. Neil D. Jones. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 114–128, London, UK, UK, 1981. Springer-Verlag.
11. K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
12. Ákos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy. Interprocedural static slicing of binary executables. In *Proc. Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 118–127, September 2003.
13. M. Kowalkiewicz, R. Lu, S. Bäuerle, M. Krümpelmann, and S. Lippe. Weak dependencies in business process models. In Witold Abramowicz and Dieter Fensel, editors, *Business Information Systems*, volume 7 of *Lecture Notes in Business Information Processing*, pages 177–188. Springer Berlin Heidelberg, 2008.
14. M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. *SIGARCH Comput. Archit. News*, 20(2):46–57, April 1992.
15. Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, June 2012.
16. Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.
17. Csaba Nagy, László Vidács, Rudolf Ferenc, Tibor Gyimóthy, Ferenc Kocsis, and István Kovács. Complexity measures in 4GL environment. In *Proceedings of the 2011 international conference on Computational science and Its applications - Volume Part V*, pages 293–309. Springer-Verlag, 2011.
18. Csaba Nagy, László Vidács, Rudolf Ferenc, Tibor Gyimóthy, Ferenc Kocsis, and István Kovács. Solutions for reverse engineering 4GL applications, recovering the design of a logistical wholesale system. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 343 –346, 2011.
19. J. Rech and W. Schäfer. Visual support of software engineers during development and maintenance. *SIGSOFT Softw. Eng. Notes*, 32(2):1–3, March 2007.
20. O. Shivers. Control flow analysis in scheme. *SIGPLAN Not.*, 23(7):164–174, June 1988.
21. The Institute of Electrical and Eletronics Engineers. Ieee standard glossary of software engineering terminology. IEEE Standard, September 1990.
22. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.
23. Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. Faster and more focused control-flow analysis for business process models through sese decomposition. In BerndJ. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing – ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 43–55. Springer Berlin Heidelberg, 2007.
24. Visser, W. and Păsăreanu, C. S. and Khurshid, S. Test Input Generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

Runtime Exception Detection in Java Programs Using Symbolic Execution*

István Kádár, Péter Hegedűs, and Rudolf Ferenc

University of Szeged, Department of Software Engineering
Árpád tér 2. H-6720 Szeged, Hungary
{ikadar|hpeter|ferenc}@inf.u-szeged.hu

Abstract. Most of the runtime failures of a software system can be revealed during test execution only, which has a very high cost. In Java programs, runtime failures are manifested as unhandled runtime exceptions.

In this paper we present an approach and tool for detecting runtime exceptions in Java programs without having to execute tests on the software. We use the symbolic execution technique to implement the approach. By executing the methods of the program symbolically we can determine those execution branches that throw exceptions. Our algorithm is able to generate concrete test inputs also that cause the program to fail in runtime.

We used the Symbolic PathFinder extension of the Java PathFinder as the symbolic execution engine. Besides small example codes we evaluated our algorithm on three open source systems: jEdit, ArgoUML, and log4j. We found multiple errors in the *log4j* system that were also reported as real bugs in its bug tracking system.

Keywords: Java Runtime Exception, Symbolic Execution, Rule Checking, Java Virtual Machine

1 Introduction

Nowadays, it is a big challenge of the software engineering to produce great, reliable and robust software systems. About 40% of the total development costs go for testing [1], and the maintenance activities, particularly bug fixing of the system also require a considerable amount of resources [2]. Our purpose is to develop a new method and tool, which supports this phase of the software engineering lifecycle with detecting runtime exceptions in Java programs, and finding dangerous parts in the source code, that could behave as time-bombs during further development. The analysis will be done without executing the program in a real environment.

Runtime exceptions in the Java programming language are the instances of class java.lang.RuntimeException, which represent a sort of runtime error, for example an invalid type cast, an array over indexing, or division by zero. These exceptions are dangerous because they can cause a sudden stop of the program, as they do not have to be handled by the programmer explicitly.

Exploration of these exceptions is done by using a technique called symbolic execution [3]. When a program is executed symbolically, it is not executed on

* This research was supported by the Hungarian national grant GOP-1.1.1-11-2011-0038 and the TÁMOP 4.2.4. A/2-11-1-2012-0001 European grant.

concrete input data but input data is handled as symbolic variables. When the execution reaches a branching condition containing a symbolic variable, the execution continues on both branches. This way, all of the possible branches of the program will be executed in theory. Java PathFinder (JPF) [4] is a software model checker which is developed at NASA Ames Research Center. In fact, Java PathFinder is a Java virtual machine that executes Java bytecode in a special way. Symbolic PathFinder (SPF) [5] is an extension of JPF, which can perform symbolic execution of Java bytecodes. The presented work is based on these tools.

The paper explains how the detection of runtime exceptions of the Java programming language was implemented using Java PathFinder and symbolic execution. Concrete input parameters of the method resulting a runtime exception are also determined. It is also described how the number of execution branches, and the state space have been reduced to achieve a better performance. The implemented tool called *Jpf Checker* has been tested on real life projects, the log4j, ArgoUML, and jEdit open source systems. We found multiple errors in the *log4j* system that were also reported as real bugs in its bug tracking system. The performance of the tool is acceptable since the analysis was finished in a couple of hours even for the biggest system.

The remainder of the paper is organized as follows. We give a brief introduction to symbolic execution in Section 2. After that in Section 3 we present our approach for detecting runtime exceptions. Section 4 discusses the results of the implemented algorithm on different small examples and real life open source projects. Section 5 collects the works that related to ours. Finally, we conclude the paper and present some future work in Section 6.

2 Symbolic Execution

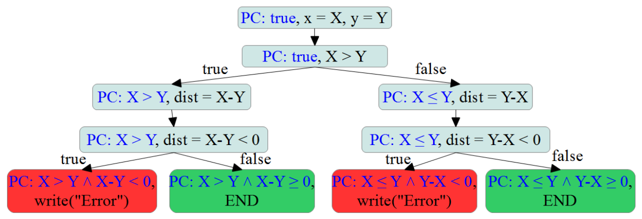
During its execution, every program performs operations on the input data in a defined order. Symbolic execution [3] is based on the idea that the program is operated on symbolic variables instead of specific input data, and the output will be a function of these symbolic variables. A symbolic variable is a set of the possible values of a concrete variable in the program, thus a symbolic state is a set of concrete states. When the execution reaches a selection control structure (e.g. an if statement) where the logical expression contains a symbolic variable, it cannot be evaluated, its value might be also true and false. The execution continues on both branches accordingly. This way we can simulate all the possible execution branches of the program.

During symbolic execution we maintain a so-called *path condition (PC)*. The path condition is a quantifier-free logical formula with the initial value of true, and its variables are the symbolic variables of the program. If the execution reaches a branching condition that depends on one or more symbolic variables, the condition will be appended to the current PC with the logical operator *AND* to indicate the true branch, and the negation of the condition to indicate the false branch. With such an extension of the PC, each execution branch will be linked to a unique formula over the symbolic variables. In addition to maintaining the path condition, symbolic execution engines make use of the so called *constraint solver* programs. Constraint solvers are used to solve the path condition by assigning values to the symbolic variables that satisfy the logical formula. Path condition can be solved at any point of the symbolic execution. Practically, the solutions serve as test inputs that can be used to run the program in such a way that the concrete execution follows the execution path for which the PC was solved.

All of the possible execution paths define a connected and acyclic directed graph called *symbolic execution tree*. Each point of the tree corresponds to a symbolic state of the program. An example is shown in Figure 1.

```
1. int x, y, dist;
2. ...
3. if (x > y) {
4.   dist = x - y;
5. } else {
6.   dist = y - x;
7. }
8. if (dist < 0)
9.   write("Error");
```

(a)



(b)

Fig. 1: (a) Sample code that determines the distance of two integers on the number line (b) Symbolic execution tree of the sample code handling variable x and y symbolically

Figure 1 (a) shows a sample code that determines the distance of two integers x and y. The symbolic execution of this code is illustrated on Figure 1 (b) with the corresponding symbolic execution tree. We handle x and y symbolically, their symbols are X and Y respectively. The initial value of the path condition is true. Reaching the first if statement in line 3, there are two possibilities: the logical expression can be true or false; thus the execution branches and the logical expression and its negation is added to the PC as follows:

$$\mathit{true} \wedge X > Y \Rightarrow X > Y, \quad \text{and} \quad \mathit{true} \wedge \neg(X > Y) \Rightarrow X \leq Y$$

The value of variable *dist* will be a symbolic expression, X-Y on the true branch and Y-X on the false one. As a result of the second if statement (line 8) the execution branches, and the appropriate PCs are appended again. On the true branches we get the following PCs:

$$X > Y \wedge X - Y < 0 \Rightarrow X > Y \wedge X < Y,$$

$$X \leq Y \wedge Y - X < 0 \Rightarrow X \leq Y \wedge X > Y$$

It is clear that these formulas are unsolvable, we cannot specify such X and Y that satisfy the conditions. This means that there are no such x and y inputs with which the program reaches the *write(”Error”)* statement. As long as the PC is unsatisfiable at a state, the sub-tree starting from that state can be pruned, there is no sense to continue the controversial execution.

It is impossible to explore all the symbolic states. It takes unreasonably long time to execute all the possible paths. A solution for this problem can be e.g. to limit the depth of the symbolic execution tree or the number of states which, of course, inhibit to examine all the states. The next subsection describes what are the available techniques in Symbolic PathFinder to address this problem.

2.1 Java PathFinder and Symbolic PathFinder

Java PathFinder (JPF) [4] is a highly customizable execution environment that aims at verifying Java programs. In fact, JPF is nothing more than a Java Virtual Machine which interprets the Java bytecode in a special way to be able to verify certain properties. It is difficult to determine what kind of errors can be found and which properties can be checked by JPF, it depends primarily on its configuration. The system has been designed from the beginning to be easily configurable and extendable. One of its extensions is *Symbolic PathFinder (SPF)* [5] that provides symbolic execution of Java programs by implementing a bytecode instruction set allowing to execute the Java bytecode according to the theory of symbolic execution.

JPF (and SPF) itself is implemented in Java, so it also have to run on a virtual machine, thus JPF is actually a middleware between the standard JVM and the bytecode. The architecture of the system is illustrated on Figure 2.

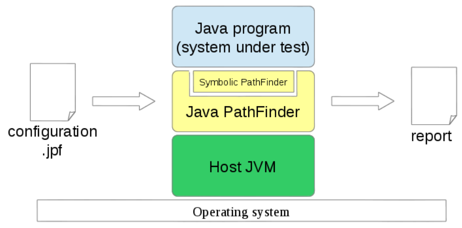


Fig. 2: Java PathFinder as a virtual machine itself runs on a JVM, while performing a verification of a Java program

To start the analysis we have to make a configuration file with .jpf extension in which we specify different options as key-value pairs. The output is a report that contains e.g. the found defects. In addition to the ability of handling logical, integer and floating-point type variables as symbols, SPF can also handle complex types symbolically with the lazy initialization algorithm [6], and allows the symbolic execution of multi-threaded programs too.

SPF supports multiple constraint solvers and defines a general interface to communicate them. *Cvc3* is used to solve linear formulas, *choco* can handle non-linear logical formulas too, while *IASolver* use interval arithmetic techniques to satisfy the path condition. Among the supported constraint solvers, *CORAL* proved to be the most effective in terms of the number of solved constraints and the performance [7].

To reduce the state space of the symbolic execution SPF offers a number of options. We can specify the maximum depth of the symbolic execution tree, and the number of elementary formulas in the path condition can also be limited. Further possibility is that with options *symbolic.minint*, *symbolic.maxint*, *symbolic.minreal*, and *symbolic.maxreal* we can restrict the value ranges of the integer and floating point types. With the proper use of these options the state space and the time required for the analysis can be reduced significantly.

3 Detection of Runtime Exceptions

We developed a tool that is able to automatically detect runtime exceptions in an arbitrary Java program. This section explains in detail how this analysis program, the JPF checker works.

To check the whole program we use symbolic execution, which is performed by Symbolic PathFinder. However, we do not execute the whole program symbolically to discover all of the possible paths, instead we symbolically execute the methods of the program one by one. This results in a significant reduction in the state space of the symbolic execution.

An important question is which variables to be handled symbolically. In general, execution of a method mainly depends on the actual values of its parameters and the referred external variables. Thus, these are the inputs of a method that should be handled symbolically to generally analyze it. Currently, we handle the parameters and data members of the class of the analyzed method symbolically.

Our goal is not only to indicate the runtime exceptions a method can throw (its type and the line causing the exception), but also to determine a parameterization that leads to throwing those exceptions. In addition, we determine this parameterization not only for the analyzed method which is at the bottom of the call stack, but for all the other elements in the call stack (i.e. recursively for all the called methods).

Our work can be divided into two steps:

1. It is necessary to create a runtime environment which is able to iterate through all the methods of a Java program, and start their symbolic execution using Symbolic PathFinder.
2. We need a JPF extension which is built on its listener mechanism, and which is able to indicate potential runtime exceptions and related parameterization while monitoring the execution.

3.1 The Runtime Environment

The concept of the developers of Symbolic PathFinder was to start running the program in normal mode like in a real life environment, than at given points, e.g. at more complex or problematic parts in the program switch to symbolic execution mode [8]. The advantage of this approach is that, since the context is real, it is more likely to find real errors. E.g. the values of the global variables are all set, but if these variables are handled symbolically we can examine cases that never occur during a real run. A disadvantage is that it is hard to explore the problematic points of a program, it requires prior knowledge or preliminary work. Another disadvantage is that you have to run the program manually namely, that the control reach those methods which will be handled symbolic by the SPF.

In contrast, the tool we have developed is able to execute an arbitrary method or all methods of a program symbolically. The advantage of this approach is that the user does not have to perform any manual runs, the entire process can be automated. Additionally, the symbolic state space also remains limited since we do not execute the whole program symbolically, but their parts separately. The approach also makes it possible to analyze libraries that do not have a *main* method such as log4j. One of the major disadvantages is the that we back away from the real execution environment, which may lead to false positive error reports.

For implementing such an execution environment we have to achieve somehow that the control flow reaches the method we want to analyze. However, due to the nature of the virtual machine, JPF requires the entry point of the program, which is the class containing the main method. Therefore, we generate a driver class for each method containing a main method that only passes the control to the method we want to execute symbolically and carries out all the related tasks. Invoking the method is done using the Java Reflection API. We also have to generate a JPF configuration file that specifies, among others, the artificially created entry point and the method we want to handle symbolically. After creating the necessary files, we have to compile the generated Java class and finally, to launch Symbolic PathFinder.

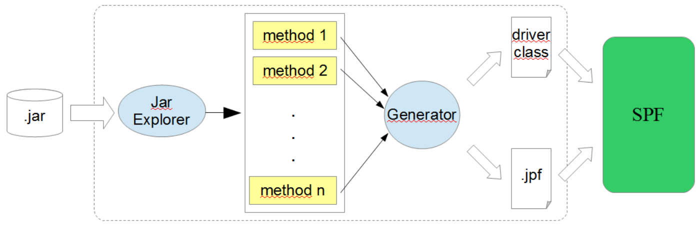


Fig. 3: Architecture of the runtime environment

The architecture of the system is illustrated in Figure 3. The input *jar* file is processed by the *JarExplorer*, which reads all the methods of the classes from the jar file and creates a list from them. The elements of the list is taken by the *Generator* one by one. It generates a driver class and a JPF configuration file for each method. After the generation is complete, we start the symbolic execution.

3.2 Implementing a Listener Class

During functioning, JPF sends notifications about certain events. This is realized with so-called listeners, which are based on the observer design pattern. The registered listener objects are notified about and can react to these events. JPF can send notifications of almost every detail of the program execution. There are low-level events such as execution of a bytecode instruction, as well as high-level events such as starting or finishing the search in the state space. In JPF, basically two listener interfaces exist: the *SearchListener* and *VMListener* interface. While the former includes the events related to the state space search, the latter reports the events of the virtual machine. Because these interfaces are quite large and the specific listener classes often implement both of them, adapter classes are introduced that implement these interfaces with empty method bodies. Therefore, to create our custom listener we derived a class from this adapter and implemented the necessary methods only.

Our algorithm for detecting runtime exceptions is briefly summarized below. By performing symbolic execution of a method all of its paths are executed, including those that throw exceptions. When an exception occurs, namely when the virtual machine executes an ATHROW bytecode instruction, JPF triggers and *excpetionThrown* event. Thus, we implemented the exceptionThrown method in our listener class. The pseudo code of our exceptionThrown implementation is shown in Figure 4.

```
1. exceptionThrown() {
2.   exception = getPendingException();
3.   if (isInstanceOfRuntimeException(exception)) {
4.     pc = getCurrentPc();
5.     solve(pc);
6.     summary = new FoundExceptionSummary();
7.     summary.setExceptionType(exception);
8.     summary.setThrownFrom(exception);
9.     summary.setParameterization(parsePc(pc, analyzedMethod));
10.    invocationChain = buildInvocationChain();
11.    foreach(Method m : invocationChain) {
12.      summary.addStackTraceElement(m, parsePc(pc, m));
13.    }
14.    foundExceptions.add(summary);
15.  }
16.}
```

Fig. 4: Pseudo code of the exceptionThrown event

First, we acquire the thrown Exception object (line 2), then we decide whether it is a runtime exception (i.e. whether it is an instance of the class RuntimeException) (line 3). If it is, we request the path condition related to the actual path and use the constraint solver to find a satisfactory solution (lines 4-5). Lines 6-9 set up a summary report that contains the type of the thrown exception, the line that throws it and a parameterization which causes this exception to be thrown. The parameterization is constructed by the *parsePC()* method, which assigns the satisfactory solutions of the path condition to the method parameters. Lines 10-13 take care of collecting and determining parameterization for the methods in the call stack. If the source code does not specify any constraint for a parameter on the path throwing an exception (i.e. the path condition does not contain the variable), then there is no related solution. This means that it does not matter what the actual value of that parameter is, it does not affect the execution path, the method is going to throw an exception due to the values of other parameters. In such cases parsePc() method assigns the value “any” to these parameters.

It is also possible that a parameter has a concrete value. Figure 5 illustrates such an example. When we start the symbolic execution of method *x()*, its parameter *a* is handled symbolically. As *x()* calls *y()* its parameter *a* is still a symbol, but *b* is a concrete value (42). In a case like this, parsePc() have to get the concrete value from the stack of the actual method.

```
void x(int a) {
  short b = 42;
  y(a, b);
}

void y(int a, short b) {
  ...
  throw new NullPointerException();
  ...
}
```

Fig. 5: An example call with both symbolic and concrete parameters

We note that the presented algorithm reports any runtime exceptions regardless of the fact whether it is caught by the program or not. The reason of this is that we think that relying on runtime exceptions is a bad coding practice and a runtime exception can be dangerous even if it is handled by the program. Nonetheless, it would be easy to modify our algorithm to detect uncaught exceptions only as SPF provides a support for it.

4 Results

The developed tool was tested in a variety of ways. The section describes the results of these test runs. We analyzed manually prepared example codes containing instructions that cause runtime exceptions on purpose; then we performed analysis on different open-source software to show that our tool is able to detect runtime exceptions in real programs, not just in artificially made small examples. The subject systems are the log4j (http://logging.apache.org/log4j/) logging library, the ArgoUML modeling tool (http://argouml.tigris.org/), and the jEdit text editor program (http://www.jedit.org/). We prove the validity of the detected exceptions by the bug reports, found in the bug tracking systems of these projects, that describe program faults caused by those runtime exceptions that are also found by the developed tool.

4.1 A Manually Prepared Example

A small manually prepared example code is shown on Figure 6. The method under test is *callRun()* which calls method *run()* in line 12. Running our algorithm on this code gives two hits: the first is an ArrayIndexOutOfBoundsException, the second is a NullPointerException. The first exception is thrown by method run() at line 24. A parameterization leading to this exception is callRun(7, 11). Method run() will be called only if $x > 6$ (line 10) that is satisfied by 7 and it is called with the concrete value 9 and symbol y. At this point there is no condition for y. Method run() can reach line 24 only if $y > 10$, the indicated value 11 is obtained by satisfying this constraint. Throwing of the ArrayIndexOutOfBoundsException is due to the fact that in line 22 we declare a 5-element array but the following for loop runs from 0 to x. The value of x at this point is 9 which leads to an exception.

The train of thought is similar in case of the second exception. The problem is that variable i created in line 27 initialized only in line 29 to a value different form *null*, but not in the else block, therefore line 33 throws a NullPointerException. This requires that the value of y not to be greater than 10 and not to be less than 5. These restrictions are satisfied by e.g. 5, and value 7 for x is necessary to invoke run(). So the parameterizations are callRun(7, 5) and run(9, 5). The analysis is finished in less than a second.

```
    public class Example5 {
    ...
8.  void callRun(int x, int y) {
9.    Integer i = null;
10.   if (x > 6) {
11.     int b = 9;
12.     run(b, y);
13.     i = Integer.valueOf(b);
14.     System.out.println(i);
15.   } else {
16.     i = Integer.valueOf(3);
17.     System.out.println(i);
18.   }
19. }

20. public void run(int x, int y) {
21.   if (y > 10) {
22.     int[] tomb = new int[5];
23.     for (int i = 0; i < x; i++) {
24.       tomb[i] = i;
25.     }
26.   } else {
27.     Integer i = null;
28.     if (y < 5) {
29.       i = Integer.valueOf(4);
30.       i.floatValue();
31.     } else {
32.       System.out.println(
33.               i.floatValue());
34.     }
35.   }
36. }}
```

Fig. 6: Manually prepared example code with the analysis of method callRun()

4.2 Analysis of Open-source Systems

Analysis of log4j 1.2.15, ArgoUML 0.28 and jEdit 4.4.2 were carried out on a desktop computer with an Intel Core i5-540M 2.53 GHz processor and 8 GB of memory. In all three cases the analysis was done by executing all the methods of the release jar files of the projects symbolically.

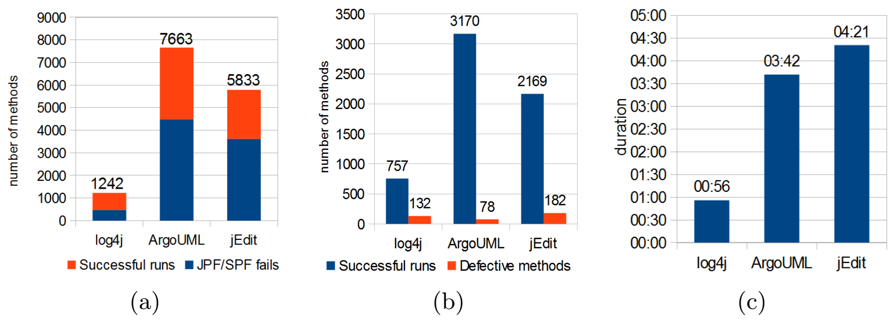


Fig. 7: (a)Number of methods examined in the programs and the number of JPF or SPF faults (b) Number of successfully analyzed methods and the number of defective methods (c) Analysis time

Figure 7 (a) displays the number of methods we analyzed in the different programs. We started analyzing 1242 methods in log4j of which only 757 were successful, in 474 cases the analysis stopped due to the failure of the Java PathFinder (or Symbolic PathFinder). There are a lot of methods in ArgoUML which also could not be analyzed, more than half of the checks ended with failure. In case of jEdit the ratio is very similar. Unfortunately, in general JPF stopped with a variety of error messages.

Despite the frequent failures of JPF, our tool indicated a fairly large number of runtime exceptions in all three programs. Figure 7 (b) shows the number of successfully analyzed methods and the methods with one or more runtime exceptions. The hit rate is the highest for log4j and despite its high number of methods, relatively few exceptions were found in ArgoUML.

The analysis times are shown in Figure 7 (c). Analysis of log4j completed within an hour, while analysis of ArgoUML, that contains more than 7500 methods, took 3 hours and 42 minutes. Although jEdit contains fewer methods than ArgoUML, its full analysis were more time-consuming. The performance of our algorithm is acceptable, especially considering that the analysis was performed on an ordinary desktop PC not on a high-performance server. However, it can be assumed that the analysis time would grow with less failed method analysis.

It is important to note, that not all indicated exceptions are real errors. This is because the analysis were performed in an artificial execution environment which might have introduced false positive hits. When we start the symbolic execution of a method we have no information about the circumstances of the real invocation. All parameters and data members are handled symbolically, that is, it is considered that their value can be anything although it is possible that a particular value of a variable never occurs.

Despite the fact that not all the reported exceptions are real program errors they are definitely representing real risks. During the modification of the source code there are inevitably changes that introduce new errors. These errors often appear in form of runtime exceptions (i.e. in places where our algorithm found possible failures). So the majority of the reported exceptions do not report real errors, but potential sources of danger that should be paid special attention.

4.3 A Real Error

In this subsection a log4j defect is shown which is reported in its bug tracking system, and caused by a runtime exception found also by our tool. The affected bug [1] reports the stoppage of an application using log4j version 1.2.14 caused by a NullPointerException. The reporter got the Exception from line 59 of *Throwable-Information.java* thrown by method *org.apache.log4j.spi.ThrowableInformation.getThrowableStrRep()* as shown in the given stack trace. The code of the method and the problematic line detected by our analysis is shown on Figure 8.

The problem here is that the initialization of the throwable data member of class ThrowableInformation is omitted, its value is null causing a NullPointerException in line 59. This causes that the *log()* method of log4j can also throw an

[1] https://issues.apache.org/bugzilla/show_bug.cgi?id=44038

```
...
public class ThrowableInformation implements java.io.Serializable {
private transient Throwable throwable;
...
 public String[] getThrowableStrRep() {
  if(rep != null) {
   return (String[]) rep.clone();
  } else {
   VectorWriter vw = new VectorWriter();
   throwable.printStackTrace(vw);
   rep = vw.toStringArray();
   return rep;
  }
 }
...
}
```

Fig. 8: Source code of method org.apache.log4j.spi.ThrowableInformation.getThrowableStrRep() included in the bug report

exception which should never happen. Our tool found other errors as well which demonstrate its strength of being capable of detecting real bugs.

5 Related Work

In this section we present works that are related to our research. First, we introduce some well-known symbolic execution engines, then we show the possible applications of the symbolic execution. We also summarize the problems that have been solved successfully by Symbolic PathFinder that we used for implementing our approach. Finally, we present the existing approaches and techniques for runtime exception detection.

The idea of symbolic execution is not new, the first publications and execution engines appeared in the 1970's. One of the earliest work is by King that lays down the fundamentals of symbolic execution [3] and presents the EFFIGY system that is able to execute PL/I programs symbolically. Even though EFFIGY handles only integers symbolically, it is an interactive system with which the user is able to examine the process of symbolic execution by placing breakpoints and saving and restoring states. Another work from the 1970's by Boyer et al. presents a similar system called SELECT [9] that can be used for executing LISP programs symbolically. The users are allowed to define conditions for variables and return values and get back whether these conditions are satisfied or not as an output. The system can be applied for test input generation; in addition, for every path it gives back the path condition over the symbolic variables.

Starting from the last decade the interest about the technique is constantly growing, numerous programs have been developed that aim at dynamic test input generation using symbolic execution. The EXE (EXecution generated Executions) [10] presented by Cadar et al. at the Stanford University is an error checking tool made for generating input data on which the program terminates with failure. The input generation is done by the STP built-in constraint solver that solves the path condition of the path causing the failure. EXE achieved promising results on real life systems. It found errors in the package filter implementations of *BSD* and *Linux*, in the *udhcpd* DHCP server and in different Linux file systems. The runtime detection algorithm presented in this work solves the path condition to generate test input data similarly to EXE. The basic difference is that for running EXE one needs to declare the variables to be handled symbolically while for Jpf Checker there is no need for editing the source code before detection.

The DART [11] (Directed Automata Random Testing) by Godefroid et al. tries to eliminate the shortcomings of the symbolic execution e.g. when it is unable to handle a condition due to its unlinear nature. DART executes the program with random or predefined input data and records the constraints defined by the conditions on the input variables when it reaches a conditional statement. In the next iteration taking into account the recorded constraints it runs the program with input data that causes a different execution branch of the program. The goal is to execute all the reachable branches of the program by generating appropriate input data. The CUTE and jCUTE systems [12] by Sen and Agha extend DART with multithreading and dynamic data structures. The advantage of these tools is that they are capable of handling complex mathematical conditions due to concrete executions. This can be also achieved in Jpf Checker by using the concolic execution of SPF; however, symbolic execution allows a more thorough examination of the source code. Further description and comparison of the above mentioned tools can be found e.g. in the work of Coward [13].

There are also approaches and tools for generating test suites for .NET programs using symbolic execution. Pex [14] is a tool that automatically produces a small test suite with high code coverage for .NET programs using dynamic symbolic execution, similar to path-bounded model-checking. Jamrozik et al. introduce an extension of the previous approach called augmented dynamic symbolic execution [15], which aims to produce representative test sets with DSE by augmenting path conditions with additional conditions that enforce target criteria such as boundary or mutation adequacy, or logical coverage criteria. Experiments with the Apex prototype demonstrate that the resulting test cases can detect up to 30% more seeded defects than those produced with Pex.

Song et al. applied the symbolic execution to the verification of networking protocol implementations [16]. The SymNV tool creates network packages with which a high coverage can be achieved in the source code of the daemon, therefore potential rule violations can be revealed according to the protocol specifications.

The SAFELI tool [17] by Fu and Qian is a SQL injection detection program for analyzing Java web applications. It first instruments the Java bytecode then executes the instrumented code symbolically. When the execution reaches a SQL query the tool prepares a string equation based on the initial content of the web input components and the built-in SQL injection attack patterns. If the equation can be solved the calculated values are used as inputs which the tool verifies by sending a HTML form to the server. According to the response of the server it can decide whether the found input can be a real attack or not.

The main application of the Java PathFinder and its symbolic execution extension is the verification of the internal projects in NASA. Bushnell et al. describes the application of Symbolic PathFinder in TSAFE (Tactical Separation Assisted Flight Environment) [18] that verifies the software components of an air control and collision detection system. The primary target is to generate useful test cases for TSAFE that simulates different wind conditions, radar images, flight schedules, etc.

The detection of design patterns can be performed using dynamic approaches as well as with static program analysis. With the help of a monitoring software the program can be analyzed during manual execution and conclusions about the existence of different patterns can be made based on the execution branches. In his work, von Detten [19] applied symbolic execution with Symbolic PathFinder supplementing manual execution. This way, more execution branches can be examined and the instances found by traditional approaches can be refined.

Ihantola [20] describes an interesting application of JPF in education. He generates test inputs for checking the programs of his students. His approach is that functional test cases based on the specification of the program and their outcome (successful or not) is not enough for educational purposes. He generates test cases for the programs using symbolic execution. This way the students can get feedbacks like "the program works incorrectly if variable a is larger than variable b plus 10".

Sinha et al. deal with localizing Java runtime errors [21]. The introduced approach aims at helping to fix existing errors. They extract the statement that threw the exception from its stack trace and perform a backward dataflow analysis starting from there to localize those statements that might be the root causes of the exception.

The work of Weimer and Necula [22] focuses on proving safe exception handling in safety critical systems. They generate test cases that lead to an exception by violating one of the rules of the language. Unlike Jpf Checker they do not generate test inputs based on symbolic execution but solving a global optimization problem on the control flow graph (CFG) of the program.

The JCrasher tool [23] by Csallner and Smaragdakis takes a set of Java classes as input. After checking the class types it creates a Java program which instantiates the given classes and calls each of their public methods with random parameters. This algorithm might detect failures that cause the termination of the system such as runtime exceptions. The tool is capable of generating JUnit test cases and can be integrated to the Eclipse IDE. Similarly to Jpf Checker JCrasher also creates a driver environment but it can analyze public methods only and instead of symbolic execution it generates random data which is obviously not feasible for examining all possible execution branches.

6 Conclusions and Future Work

The introduced approach for detecting runtime exceptions works well not just on small, manually prepared examples but it is able to find runtime exceptions which are the causes of some documented runtime failures (i.e. there exists an issue for them in the bug tracking system) in real world systems also. However, not all the detected possible runtime exceptions will actually cause a system failure. There might be a large number of exceptions that will never occur running the system in real environment. Nonetheless, the importance of these warnings should not be underrated since they draw attention to those code parts that might turn to real problems after changing the system. Considering these possible problems could help system maintenance and contributes to achieving a better quality software. As we presented in Section 4 the analysis time of real world systems are also acceptable, therefore our approach and tool can be applied in practice.

Unfortunately the Java PathFinder and its Symbolic PathFinder extension – which we used for implementing our approach – contain a lot of bugs. It made the development very troublesome, but the authors at the NASA were really helpful. We contacted them several times and got responses very quickly; they fixed some blocker issues particularly for our request.

The achieved results are very promising and we continue the development of our tool. Our future plan is to eliminate the false positive and those hits that are irrelevant. We would also like to provide more details about the environment of the method in which the runtime exception is detected. The implemented tool gives only the basic information about the reference type parameters whether they are *null* or not, and we cannot tell anything about the values of the member variables of the class playing a role in a runtime exception. These improvements of the algorithm are also in our future plans.

The presented approach is not limited to runtime exception detection. We plan to utilize the potentials of the symbolic execution by implementing other types of error and rule violation checkers. E.g. we can detect some special types of infinite loops, dead or unused code parts, or even SQL injection vulnerabilities.

References

1. Pressman, R.S.: Software Engineering: A Practitioner's Approach. McGraw-Hill Science/Engineering/Math (November 2001)
2. Tassey, G.: The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology (2002)
3. King, J.C.: Symbolic Execution and Program Testing. Communications of the ACM **19**(7) (July 1976) 385–394
4. Java PathFinder Tool-set. `http://babelfish.arc.nasa.gov/trac/jpf`
5. Păsăreanu, C.S., Rungta, N.: Symbolic PathFinder: Symbolic Execution of Java Bytecode. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE '10, New York, NY, USA, ACM (2010) 179–180
6. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized Symbolic Execution for Model Checking and Testing. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'03, Berlin, Heidelberg, Springer-Verlag (2003) 553–568
7. Souza, M., Borges, M., d'Amorim, M., Păsăreanu, C.S.: CORAL: Solving Complex Constraints for Symbolic Pathfinder. In: Proceedings of the Third International Conference on NASA Formal Methods. NFM'11, Berlin, Heidelberg, Springer-Verlag (2011) 359–374
8. Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining Unit-level Symbolic Execution and System-level

Concrete Execution for Testing NASA Software. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. ISSTA '08, New York, NY, USA, ACM (2008) 15–26
9. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT – a Formal System for Testing and Debugging Programs by Symbolic Execution. In: Proceedings of the International Conference on Reliable Software, New York, NY, USA, ACM (1975) 234–245
10. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically Generating Inputs of Death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security. CCS '06, New York, NY, USA, ACM (2006) 322–335
11. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '05, New York, NY, USA, ACM (2005) 213–223
12. Sen, K., Agha, G.: CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In: Proceedings of the 18th International Conference on Computer Aided Verification. CAV'06, Berlin, Springer-Verlag (2006) 419–423
13. Coward, P.D.: Symbolic Execution Systems – a Review. Software Engineering Journal **3**(6) (November 1988) 229–239
14. Tillmann, N., De Halleux, J.: Pex: White Box Test Generation for .NET. In: Proceedings of the 2nd International Conference on Tests and Proofs. TAP'08, Berlin, Heidelberg, Springer-Verlag (2008) 134–153
15. Jamrozik, K., Fraser, G., Tillman, N., Halleux, J.: Generating Test Suites with Augmented Dynamic Symbolic Execution. In: Tests and Proofs. Volume 7942 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2013) 152–167
16. Song, J., Ma, T., Cadar, C., Pietzuch, P.: Rule-Based Verification of Network Protocol Implementations Using Symbolic Execution. In: Proceedings of the 20th IEEE International Conference on Computer Communications and Networks (ICCCN'11). (2011) 1–8
17. Fu, X., Qian, K.: SAFELI: SQL Injection Scanner Using Symbolic Execution. In: Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications. TAV-WEB '08, New York, ACM (2008) 34–39
18. Bushnell, D., Giannakopoulou, D., Mehlitz, P., Paielli, R., Păsăreanu, C.S.: Verification and Validation of Air Traffic Systems: Tactical Separation Assurance. In: Aerospace Conference, 2009 IEEE. (2009) 1–10
19. von Detten, M.: Towards Systematic, Comprehensive Trace Generation for Behavioral Pattern Detection Through Symbolic Execution. In: Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools. PASTE '11, New York, NY, USA, ACM (2011) 17–20
20. Ihantola, P.: Test Data Generation for Programming Exercises with Symbolic Execution in Java PathFinder. In: Proceedings of the 6th Baltic Sea Conference on Computing Education Research. Baltic Sea '06, New York, ACM (2006) 87–94
21. Sinha, S., Shah, H., Görg, C., Jiang, S., Kim, M., Harrold, M.J.: Fault Localization and Repair for Java Runtime Exceptions. In: Proceedings of the 18th International Symposium on Software Testing and Analysis. ISSTA '09, New York, NY, USA, ACM (2009) 153–164
22. Weimer, W., Necula, G.C.: Finding and Preventing Run-time Error Handling Mistakes. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA '04, New York, NY, USA, ACM (2004) 419–431
23. Csallner, C., Smaragdakis, Y.: JCrasher: an Automatic Robustness Tester for Java. Software Practice and Experience **34**(11) (September 2004) 1025–1050

Composable hierarchical synchronization support for REPLICA

Jari-Matti Mäkelä[1], Ville Leppänen[1], and Martti Forsell[2]

[1] University of Turku
Dept. of Information Technology
jmjmak@utu.fi, ville.leppanen@utu.fi
[2] VTT
Oulu, Finland
Martti.Forsell@VTT.Fi

Abstract. Synchronization is a key concept in parallel programming. General purpose languages and architectures often assume a restricted form of synchronicity with the focus on asynchronous execution. The most notable trend during the recent years against asynchrony has been the raise of GPGPU devices with support of tightly synchronous regions in programs. Our REPLICA architecture continues this trend for chip multi-processors with the aim to provide an execution platform for the rich algorithm theory of synchronous shared memory algorithms.
In a simplified multi-threaded computational model with unit time amortized instruction execution cost, step-wise inter-thread synchronicity can be realized. However, the synchronicity does not trivially extend to higher level abstractions. A procedural language Fork introduces means for maintaining hierarchical synchronicity on block basis. The block level synchronicity invariant is maintained with explicit annotations. Control statements for synchronicity mode transitions are explicit.
We focus on a new chip multi-processor architecture REPLICA, employing step-wise synchronization along with fine-grained requirements for special thread group based parallel operations. A new hierarchical, composable, control flow synchronization analysis method is proposed. Our annotations capture the intent better and can be partly inferred. We demonstrate the method's potential with comparisons to existing systems. Our Replica compiler contains an initial implementation.

Keywords: parallel programming, synchronization, threads

1 Introduction

Synchronization is one of the key concepts in parallel programming. In general purpose programming languages and architectures, it is a common approach to assume the synchronicity of some core features, but consider the programming model asynchronous as a whole [1,2]. Whenever synchronous execution is required, it is achieved by applying expensive explicit synchronization on limited regions. This trend has become the norm as architectures and programming environments depend more on out-of-order execution and dynamically scheduled pre-emptive multitasking for speed-ups and as the cost of synchronization has increased with relaxation of synchronization and consistency models [3].

MPI-style and typical OO-language style solutions to specify several concurrently running threads are not very fruitful in the large scale, because by default the threads' execution proceeds asynchronously resulting vagueness in computation's state, and the correctness of multithreaded program is hard to guarantee. Extensive use of locks removes vagueness but leads to poor parallel performance. [4] puts this as "Programming parallel applications with basic concurrency primitives, be it on shared or distributed memory models, breaks all rules of software composition. This leads to non-determinism, unexplained performance issues, debugging and testing mightmares, and does not allow for architectural optimizations. Even specialists struggle to comprehend ..."

The concept of computation's state gets promoted when one aims at having thread-wise synchronously proceeding control flows. The concept of state has been in a central role in achieving correctness in SW engineering of sequential programs as it is heavily related to algorithmic design, specification and testing.

The *REPLICA architecture* [5,6,7] is a very different approach to parallelism compared to the current mainstream. It has an execution mode for maintaining the synchronicity property *implicitly* at the instruction level. This is achieved with synchronization wave technique embedded into the whole architecture. As instructions have a unit time amortized cost in terms of cycles in relation to other threads, it is possible to reason about the time cost of a sequence of instructions assuming no branching happens, but even REPLICA's strong synchronization guarantees do not automatically extend to high level abstractions.

The Replica language [7,8] introduces a way to extend the same style of synchronicity to the level of high level language abstractions, expressions, statements, and declarations using a hierarchical concept of thread groups. The *Fork* [9] language adopts a similar approach for maintaining synchronicity on block and function level, but does it explicitly with annotated regions. Our approach is a novel new synchronization analysis which is similar but orthogonal to the language's type system. The system can be used for checking the program correctness via constraints related to synchronicity conditions, but it also gives rise to a possibility to infer the synchronicity properties, mitigating the need to annotate large bodies of code.

The proposed synchronization analysis uses conservative structural induction to build up a view of the program, starting from basic language features, and extending seamlessly to user defined abstractions. As the analysis is conservative, programs violating the synchronicity assumption will not be accepted, but the synchronicity inference is not optimal and can be improved or manually guided with annotations. While only core language is considered in this paper, the analysis is generic enough to be further extended with the language.

The scope and main contribution of this paper is to present the new synchronization model and the algorithms for using the model and discuss its applicability in practice. The treatment of the system focuses on its expressiveness and comparing the computational capabilities and behavior to existing languages designed for similar architectures such as E [10] and Fork. A full analysis of the model's effect on execution performance requires further tuning of the inference algorithm and is listed as a future work.

The rest of the paper is outlined as follows: Section 2 discusses modeling the synchronicity in a parallel language and considers the feasibility of using them for parallel programming on an architecture such as REPLICA. Section 3 introduces the new synchronization model and gives a description of the semantics for core language constructs. In Section 4 we demonstrate the use of languages to solve a parallel programming kernel using different features of the language. Finally in Section 5 we draw conclusions and discuss future work.

2 Modeling synchronicity

Modern computational platforms employ varying models of synchronicity, memory consistency models and granularity of parallelism. The motivation for such complex core semantics arise from the hardware and programming models and the effort of efficiently translating programs to the machine architecture. Additionally, general purpose programming languages can provide different forms of synchronicity for different kinds of tasks involving concurrency as all tasks do not require or benefit from strictly synchronous execution. To list a few, examples of such tasks are fully independent background tasks with no need for synchronization, asynchronous interaction with I/O devices, exclusive concurrent access to a resource, and propagating the results of a parallel computation between worker threads. Some architectures have special parallel operations [11] that require synchronized state for each thread participating in the operation.

Many attempts to speed up sequential performance by relaxing synchronicity on hardware and operating system level have had a negative impact on the complexity of building well performing and behaving parallel programs [3]. Examples of such issues are the overhead of starting new concurrent tasks and the cost of propagating the result of a computation inside a concurrent program.

Recently a door to significant performance speed-ups opened in the form of general purpose GPUs with a tighter, group-oriented synchronicity and execution model [12]. We argue that more and different forms of parallel computational power can still be harnessed from a similar kind of inherently synchronous model. Our aim is a structured, tighter form of synchronicity, similar to one provided by the Fork language. A practical aspect to this work comes from our implementation and design of a systems programming language Replica employing these techniques on the new architecture prototype called REPLICA.

The semantics of our synchronization model is based on the idea of tagging the control flow and executable language operations with synchronization related conditions and compile-time verification of the condition constraints. The assortment of supported conditions is based on the work of evaluating previous language techniques and computational kernels and extracting their patterns.

2.1 Previous work on strong hierarchical synchronicity

Keller et al. [13] carried out a similar survey on potential languages for a MIMD style, exactly synchronous shared memory architecture and found that there is only little interest in that particular area. Moreover, while traditional research paradigms such as functional languages are gaining traction in practice, we believe that a traditional procedural language with specially crafted extensions still hits the sweet spot in the near future, when it comes to practicality in terms of execution overhead and interacting with hardware on this abstraction level.

Widely known standard solutions for asynchronous shared memory programming (POSIX threads, OpenMP [2]), data-parallel (variants of Fortran) and message passing (MPI [14]) exist, but are all suboptimal for a hardware architecture with strong (lock-step) synchronicity and no special emphasis on massive data parallelism (e.g. vector registers, XMT [15] style on-demand thread creation). Thus, we found the E and Fork languages to be closest to our goals. In the next sections we take a brief look at both languages and their synchronization models.

However, worth noting is that while this style of parallel programming greatly benefits from the hardware feature set of architectures such as REPLICA, we do not believe a strong synchronicity model fits all purposes. One of our goals is to combine together several compatible techniques such as task parallelism [16] and high level parallel skeleton frameworks [17]. From this perspective, the role of a strongly synchronous model is to provide a safer, easier, and more refined model for taking advantage of fine-grained parallelism and the architecture dependent accelerated multi-operations.

2.2 Fork language

Fork assumes a hardware model with lock-step execution semantics and P independent, concurrently running threads, each carrying local data such as their thread id. The execution mode for a group of threads can be asynchronous or synchronous. In the former, all threads work independently, in the latter the threads work in groups. A synchronicity invariant holds for each group. The machine starts with all threads in a single group, but the groups can be recursively split to subgroups, forming a hierarchical tree-like model. Split groups can later be joined together into the parent group. It is also possible to step outside the hierarchy tree and form a new group from any existing set of threads, but care needs to be taken when manually maintaining the synchronicity invariant.

Fork provides a control primitives for synchronicity (`start, seq, farm`, and `join`) and three modes of execution (`async, sync, straight`) for blocks and functions. Fork enforces synchronicity constraints by prohibiting certain calls statically on block basis. Synchronous code cannot call asynchronous functions, asynchronous code cannot call synchronous functions and straight code can only call straight functions. The control primitives extend the rules by allowing asynchronous statements inside synchronous blocks (`farm, seq`), and synchronous statements inside the other two blocks (`start, join`). In addition, heap object allocation is prohibited in asynchronous and `continue` in synchronous mode.

The difference between farm and seq is that farm lets all threads participate in asynchronous execution while in seq only one thread is active. Start switches to a synchronous mode by performing a barrier with all the threads from the group whose member the thread last was. Join is a more flexible way of forming synchronous groups, based on a dynamic condition expressions.

Fork also adds code related to synchronicity in certain cases: short-circuit expressions, conditionals and loops generate subgroups if a condition depends on thread private state (a conservative heuristic is used). Farm, seq, and start add barriers to enforce synchronicity. Statements `break` and `return` also add barriers in the synchronous mode.

The downside of Fork is that mixing asynchronous and synchronous code requires explicit notation also on the calling side. Both modes support different functionality and the transitions between the modes have an overhead. This makes the programming effort of switching between the modes cumbersome. The default mode is asynchronous, but the programmer has to explicitly realize e.g. when a helper function is also usable by synchronous code. Subgroups are also created conservatively, which may affect performance. Instead of prohibiting certain cases of bad behavior, Fork only warns e.g. if farm is used in asynchronous mode. These rationale behind the loopholes is most likely that the compiler's constraints would otherwise also prohibit certain cases of correct, useful code.

2.3 E language

Compared to Fork, E seems like a poor man's version with similar type of goals. It assumes similar hardware model with lock-step execution semantics and threading model. The language is rather built as a macro extension for C and cannot enforce e.g. region based synchronization modes. While its model is easier to implement in terms of compilation techniques, it provides a rich set of intrinsics and constructs for manipulating the machine state – for example, functions for experimental fast versions of parallel operations under special assumptions.

E provides alternative parallel versions of standard C control structures (selection, loops) that either create similar subgroups for diverging control flows like Fork or end with a synchronization or do both. In addition, other hardware provided features such are barriers and multi(prefix)-operations are available via macros. The programmer can easily construct parallel programs using diverse set of hardware features. However, the machine conditions are only enforced in trivial cases where the code used is surrounded with the parallel control structures and a correct version structure is selected.

While E's design is less inspiring as a starting point for compilation algorithms, E's functionality and lightweight style contributed to the list of requirements for refining the Replica language, which is treated in the next section.

2.4 Replica language

The original goal in the REPLICA project was to implement language changes required by the new parallel architecture on top of plain C, extended in Fork or E style. Notable new hardware level requirements were e.g. the step-wise synchronized execution, fast synchronization and parallel aggregate instructions, and the bunch/NUMA hybrid modes for accelerating "legacy" code, which are unfortunately out of the scope of this paper. A low-overhead support for improved parallel library construction in generic or object oriented style was also planned, along with improvements in the analysis of synchronicity, data ownership and new parallel code optimizations.

The complexity of integrating all the features lead to the introduction of Replica language [7,8]. Replica implements a strongly typed, simplified subset of C. It includes the basic data types (integral, composite structs, functions, global and local variables) and imperative operations (`if-else, while, do-while, switch`). The distinction between statements and expressions is more strict. For example, assignments only work as statements. Expresions with immediate side effects are disallowed (e.g. post- and pre-increments). As an exception, function calls still work both as statements and expressions.

In Replica, the control flow operations are extended to automatically support thread subgroup creation when the control flow may diverge. A `split` construct is introduced for explicitly splitting the thread group into subgroups. Replica also provides similar access to thread/group id variables as Fork and E. In addition, a type-class [18] based generics implementation was adopted for implementing low-overhead parallel libraries.

Although Replica is a simplified version of C, in this paper we consider a language subset with decreased redundancy (e.g. only `do-while`, no `while`). The EBNF representation of the relevant grammatical part is given in Figure 1.

3 Control flow and state invariants

The base of our model is the notion of synchronicity. For simplicity, we start building the model from a synchronized subset of language features. We assume core execution semantics from REPLICA, i.e. an amortized unit time instruction execution cost with respect to other threads (distinct from wall-clock time) and a global lockstep synchronization between all threads. Synchronicity is defined pair-wise as an inter-thread relation of having the same program counter value at a given point of time. Thus, at any point of time, threads can be partitioned into groups of one or more according to their synchronicity. If the threads in a group of size n each have a unique id between 0 and $n-1$, the group is enumerated.

Due to the global lock-step synchronization, threads maintain synchronicity between any two points in their execution path unless explicit branching based on a thread-local condition is used. However, in a high level language, it becomes hard to keep track of synchronicity on machine instruction level as the execution costs of operations on a certain abstraction level can be both variable and dynamic. On high level, we statically model the execution with the concept of concurrent control flows, which can be used to piece-wise define sections of code with a certain property with respect to synchronicity.

⟨*declaration*⟩	::=	‘`fun`’ ⟨*type*⟩ ⟨*variable*⟩ ‘`(`’ [⟨*type*⟩ ⟨*variable*⟩ ‘`,`’ ⟨*type*⟩ ⟨*variable*⟩] ‘`)`’ ‘`{`’ ⟨*statement*⟩ ‘`}`’
⟨*statement*⟩	::=	[⟨*annotation*⟩] (⟨*seq*⟩ | ⟨*funcall*⟩ | ⟨*if-else*⟩ | ⟨*do*⟩ | ⟨*assignment*⟩ | ⟨*return*⟩)
⟨*annotation*⟩	::=	‘`@{`’ ⟨*annsign*⟩ ⟨*variable*⟩ ⟨*annsign*⟩ ‘`}`’
⟨*annsign*⟩	::=	‘`+`’ | ‘`-`’ | ⟨*empty*⟩
⟨*seq*⟩	::=	⟨*statement*⟩ ⟨*statement*⟩
⟨*funcall*⟩	::=	⟨*funexpr*⟩ ‘`;`’
⟨*if-else*⟩	::=	‘`if`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ⟨*statement*⟩ ‘`else`’ ⟨*statement*⟩
⟨*do*⟩	::=	‘`do`’ ⟨*statement*⟩ ‘`while`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ‘`;`’
⟨*assignment*⟩	::=	⟨*expression*⟩ ‘`=`’ ⟨*expression*⟩ ‘`;`’
⟨*return*⟩	::=	‘`return`’ ⟨*expression*⟩ ‘`;`’
⟨*expression*⟩	::=	⟨*primitive-literal*⟩ | ⟨*reference*⟩ | ⟨*conditional*⟩ | ⟨*short-circuit*⟩ | ⟨*composite*⟩ | ⟨*funexpr*⟩
⟨*reference*⟩	::=	⟨*variable*⟩
⟨*conditional*⟩	::=	⟨*expression*⟩ ‘`?`’ ⟨*expression*⟩ ‘`:`’ ⟨*expression*⟩
⟨*composite*⟩	::=	(‘`!`’ | ‘`*`’ | ‘`&`’) ⟨*exression*⟩ | ⟨*binary*⟩
⟨*binary*⟩	::=	⟨*expression*⟩ (‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’) ⟨*expression*⟩
⟨*short-circuit*⟩	::=	⟨*expression*⟩ (‘`&&`’ | ‘`||`’) ⟨*expression*⟩
⟨*funexpr*⟩	::=	⟨*expression*⟩ ‘`(`’ [⟨*expression*⟩ ‘`,`’ ⟨*expression*⟩] ‘`)`’

Fig. 1. Basic Replica language grammar

In an imperative language, the execution is controlled with three basic rules: sequence, repetition, and selection. In C [19] and its derivatives, these map to the implicit top-down execution semantics of statements, call-by-value expression evaluation order, and explicit control constructs (`do, while, goto, if`, and `switch`). Functions aggregate and encapsulate statements, raising the level of abstraction in a structured, hierarchical manner.

3.1 Modeling the flow condition invariant

The program's possible execution paths form a graph, where the set of vertices is a union of expressions, statements and declarations derived from the program's abstract syntax tree, the edges are determined from the evaluation order and the function calling sequence between the nodes. We associate with each edge a set of control flow conditions (e.g. $\langle F, F' \rangle$ or $\langle F_1, F_2 \rangle$), and with each vertex a pair of sets of flow conditions representing the constraints for flow conditions before and after the node's evaluation. The model also comes with a list of inference rules similar to type rules that determine the relations between flow conditions internal to a node (e.g. subexpressions) and the external pre- and postconditions. The model supports arbitrary amount of conditions, but our preliminary version is focused on the REPLICA architecture with the following list of conditions:

- C_S = The flow consists of threads with the same program counter value.
- C_G = After last group creation, no branching occurred / branches converged.
- C_T = The flow has exclusive ownership of a fast synchronization token.
- C_1 = The control flow consists of at most a single thread.
- C_L = Threads in the flow are located on the same physical processor.
- C_P = The control flow executes code that depends on thread-private state.

C_G represents a possibly asynchronous, enumerated thread group, C_S a synchronous thread group (not necessarily enumerated), and $\{C_G, C_S\}$ a synchronous enumerated group. C_T models REPLICA's support for a limited number of concurrent fast parallel multiprefix-style operations, each associated with an id value on the machine level. C_T is needed as the REPLICA architecture has special hardware for these operations – but only a limited number of such operations can be issued at the same time. C_1 annotates operations that are inherently sequential, C_L enables using coalesced active memory operations and processor local features such as local storage, C_P models expressions that may diverge execution when encountering a conditional. While the conditions are related to REPLICA, a similar set of conditions can be extracted from other parallel computational models.

3.2 Checking of the condition constraints

As the flow constraints are defined as inference rules in formal logic, a standard type checker can be adapted to automatically check for program correctness with respect to synchronicity. Analogous to type checking, the rules also give rise to similar potential for synchronicity and flow condition inference, for which a full algorithm is out of the scope of this paper on this model, but we outline the mechanism at the end of the section.

The rules for the core features of Replica are given next. Equations 1…5 represent shorthand functions for testing conditions (*pr, sync, tok*) related to node *node*, for removing synchronicity (*async*) and for extracting the C_P condition from a list of nodes (*prs*). The rest of the rules for each category of language constructs are described in Sections 3.2.1…3.2.4.

$$pr(node) = C_P \in F_2 \mid node : \langle F_1, F_2 \rangle \tag{1}$$

$$tok(node) = C_T \in F_2 \mid node : \langle F_1, F_2 \rangle \tag{2}$$

$$sync(node) = C_S \in F_2 \mid node : \langle F_1, F_2 \rangle \tag{3}$$

$$async(F) = F \setminus \{C_S\} \tag{4}$$

$$prs(node_1, \ldots, node_n) = \bigcup_{i=1}^{n} F_{e_i} \cap \{C_P\} \mid \forall i \in 1 \ldots n.\ node_i : \langle F_i, F_{e_i} \rangle \tag{5}$$

3.2.1 Condition inference rules of user-defined annotations

The condition rules bear resemblance to ordinary type inference rules. Instead of a type, the nodes are ascribed with a pair of sets of conditions associated with the control flow graph vertex, i.e. the syntax tree node. For each condition inference rule (or a set of rules), an analogous grammar construct exists (see Figure 1).

The first two rules (Equations 6, 7) depict the user defined annotations that can be associated with any node (the analogous grammar rule in Figure 1 is <*annotation*>). `@{+X}` requires that the condition is set, `@{-X}` is means the opposite, that `X` is not set. `@{X+}` means that the condition is set after the node, `@{X-}` means again the opposite, that `X` is set off. `X` can be any variable name, for example any of the condition names mentioned above (C_S, C_G, C_T, etc.).

It is always considered safe to add pre-conditions with `@{+X}` and to remove post-conditions with `@{X-}` as they will only disable (correct) code from compiling. The opposite could potentially lead to code that misbehaves e.g. if the synchronicity property is broken. Special intrinsic functions may use the other two annotations.

$$\text{F-PRE}\frac{s : \langle F, F' \rangle \qquad \forall X}{(@\{+X\}\ s) : \langle F \cup \{C_X\}, F' \rangle \quad (@\{-X\}\ s) : \langle F \setminus \{C_X\}, F' \rangle} \tag{6}$$

$$\text{F-POST}\frac{s : \langle F, F' \rangle \qquad \forall X}{(@\{X+\}\ s) : \langle F, F' \cup \{C_X\} \rangle \quad (@\{X-\}\ s) : \langle F, F' \setminus \{C_X\} \rangle} \tag{7}$$

3.2.2 Condition inference rules for statements

The second category has a list of basic statements. Other statements (`for, if` without `else, switch, split`) can be composed from these by rewrite. For simplicity, a set of other Replica features (`break, continue, goto, label, sequential, numa`, blocks) have been left out from this version, but the behavior described in [13] can be adopted by expanding the set of conditions. To simplify the analysis, we assume functions to be non-recursive, first-order and have a single explicit exit point (`return`) – multiple exit points can be reduced by similar rewrite rules.

The basic rule for statements (F-STMT) defines that an unannotated statement must preserve all flow conditions except synchronicity. The sequence rule (F-SEQ) states that the next statement may only require a subset of conditions offered by the preceding statement. The rules related to conditionals (F-IF) and loops (F-DO) state that the C_T token (see Section 3.1) cannot duplicate when the flow diverges and also that C_G will not propagate to diverging branches. F-ASSIGN obeys the same kind of sequence logic as F-SEQ and F-RET does not change the return value's (e) flow conditions.

$$\text{F-STMT}\frac{s : \langle F, F' \rangle \qquad async(F) = async(F`)}{\top} \tag{8}$$

$$\text{F-SEQ}\frac{s : \langle F_1, F_2 \rangle \qquad s_2 : \langle F_3, F_4 \rangle \qquad F_2 \supseteq F_3}{s; s_2 : \langle F_1, F_4 \rangle} \tag{9}$$

$$\text{F-FUN}\ \frac{f(p_1,\ldots,p_n) : \langle F, F' \rangle}{f(p_1,\ldots,p_n) : \langle F, F' \rangle} \quad (10)$$

$$\text{F-SYNC}^*\ \frac{}{\textbf{sync} : \langle F, F \cup \{C_S\} \rangle} \quad (11)$$

$$\text{F-GROUP}^*\ \frac{}{\textbf{group} : \langle F \cup \{C_S\}, F \cup \{C_S, C_G\} \rangle} \quad (12)$$

$$\text{F-IF}\ \frac{\begin{array}{c} e : \langle F_1, F_c \rangle \qquad s_1 : \langle F_{e_1}, F_{ee_1} \rangle \qquad s_2 : \langle F_{e_2}, F_{ee_2} \rangle \qquad F_c \supseteq F_{e_1} \cup F_{e_2} \\ pr(e) \Rightarrow F_2 = async(F_1) \qquad tok(e) \Rightarrow \neg tok(s_1) \land \neg tok(s_2) \\ pr(e) \Rightarrow C_G \notin F_{e_1} \cup F_{e_2} \qquad \neg pr(e) \Rightarrow F_2 = F_{ee_1} \cap F_{ee_2} \cup (F_1 \cap \{C_T\}) \end{array}}{\textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2 : \langle F_1, F_2 \rangle} \quad (13)$$

$$\text{F-DO}\ \frac{\begin{array}{c} s : \langle F_1, F_l \rangle \qquad e : \langle F_e, F_{ee} \rangle \qquad F_{ee} \supseteq F_1 \qquad \neg pr(e) \Rightarrow F_2 = F_{ee} \\ pr(e) \Rightarrow F_2 = async(F_1) \land \neg tok(s) \land \neg tok(e) \land C_G \notin F_1 \qquad F_l \supseteq F_e \end{array}}{\textbf{do}\ s\ \textbf{while}\ (e) : \langle F_1, F_2 \rangle} \quad (14)$$

$$\text{F-ASSIGN}\ \frac{e : \langle F_1, F_2 \rangle \qquad e_2 : \langle F_3, F_4 \rangle \qquad F_2 \supseteq F_3}{e_2 = e : \langle F_1, F_4 \rangle} \quad (15)$$

$$\text{F-RET}\ \frac{e : \langle F, F' \rangle}{\textbf{return}\ e : \langle F, F' \rangle} \quad (16)$$

3.2.3 Condition inference rules for expressions The next list defines all language expressions. Rules for primitives (T-PRIMITIVE) and shared references (T-REF) are trivial. References to private variables (T-REF-PRIV) spawn the C_P condition. Ternary (T-COND) and short-circuit (T-SHORT) operations make similar assumptions as conditional statements (see T-IF in Equation 13), but also propagate the thread-private state C_P like many expressions. Function call checks arguments' conditions according to the evaluation order. The rest of the built-in expressions can be fit to use the same composite rule, which checks the condition compatibility in the evaluation order of subexpression arguments.

$$\text{F-PRIMITIVE / F-REF}\ \frac{}{e : \langle F, F \rangle} \quad (17)$$

$$\text{F-REF-PRIV}\ \frac{}{e : \langle F, F \cup C_P \rangle} \quad (18)$$

$$\text{F-COND}\ \frac{\begin{array}{c} c : \langle F_1, F_c \rangle \qquad e_1 : \langle F_{e_1}, F_{ee_1} \rangle \qquad e_2 : \langle F_{e_2}, F_{ee_2} \rangle \\ F_c \setminus \{C_P\} \supseteq F_{e_1} \cup F_{e_2} \qquad tok(c) \Rightarrow \neg tok(s_1) \land \neg tok(s_2) \\ pr(c) \Rightarrow F_2 = async(F_1) \cup F_p \land C_G \notin F_{e_1} \cup F_{e_2} \\ \neg pr(c) \Rightarrow F_2 = F_{ee_1} \cap F_{ee_2} \cup (F_1 \cap \{C_T\}) \cup F_p \qquad F_p = prs(c, e_1, e_2) \end{array}}{(c\ ?\ e_1\ :\ e_2) : \langle F_1, F_2 \rangle} \quad (19)$$

$$\text{F-SHORT}\frac{\begin{array}{c} e_1 : \langle F_1, F_2 \rangle \qquad e_2 : \langle F_3, F_4 \rangle \qquad F_2 \supseteq F_3 \qquad tok(e_1) \Rightarrow \neg tok(e_2) \\ pr(e_1) \Rightarrow F_5 = async(F_1) \cup F_p \wedge C_G \notin F_3 \\ \neg pr(e_1) \Rightarrow F_5 = F_{ee_1} \cap F_{ee_2} \cup (F_1 \cap \{C_T\}) \cup F_p \qquad F_p = prs(e_1, e_2) \end{array}}{(e_1 \;\&\&\; e_2) : \langle F_1, F_5 \rangle \quad (e_1 \mid\mid e_2) : \langle F_1, F_5 \rangle} \tag{20}$$

$$\text{F-COMPOSITE}\frac{\forall i.c_i : \langle F_{c_i}, F_{ce_i} \rangle \qquad \forall i > 1.F_{ce_{i-1}} \supseteq F_{c_i}}{< operator > (c_1, \ldots, c_n) : \langle F_{c_1}, F_{ce_n} \cup prs(c_1, \ldots, c_n) \rangle} \tag{21}$$

$$\text{F-FUN-E}\frac{f : \langle F_f, F_{fe} \rangle \qquad \forall i.p_i : \langle F_{p_i}, F_{pe_i} \rangle \qquad \forall i > 1.F_{pe_{i-1}} \supseteq F_{p_i} \qquad F_{pe_n} \supseteq F_f}{f(p_1, \ldots, p_n) : \langle F_{p_1}, F_{fe} \rangle} \tag{22}$$

3.2.4 Condition inference rules for function declarations Finally the rule for function declarations mainly propagates the result from the function body, but can be used as a point for attaching annotations on the callee's side is given in Equation 23.

$$\text{F-FUNDECL}\frac{s : \langle F, F' \rangle}{fun < type >< name > (< p_1 >, \ldots, < p_n >) \; \{s\} : \langle F, F' \rangle} \tag{23}$$

Since the rules have been described, we discuss the condition checking algorithm, which is quite simple. We start from all top-level declarations and recursively apply the rules until we obtain a pair of sets of conditions for the top-level declarations. If there is no matching rule for a certain node, we can still do a pattern match based on the syntax tree node type and display the offending rule and related flow conditions. Once the top-level declarations are associated with a pair, we check to see if the pre-condition of the *main* function is compatible with $\{C_S\}$, the initial state of the machine.

3.2.5 On condition inference algorithm To get an idea of the condition inference algorithm, we describe a preliminary version used in our current compiler. Instead of starting with all top-level declarations we start with the *main* function and pass around the current flow conditions calculated from the previous flow state. The rules are built in such a way that in a correct program, we can progress to all subnodes in a certain order without backtracking. If there is no way to proceed, we have three implicit conversion rules for switching between the states: in the inference equations, the rules marked with a star, F-SYNC (Equation 11), F-GROUP (Equation 12), and the third rule is a sequence of rules F-SYNC & F-GROUP. There is also a plan to add a fourth conversion to a single threaded context with the code `split { group(1) { ...} } )`. If none of the conversions can be applied, we issue an error as in the previous algorithm. The purpose of the implicit conversion rules is to achieve the same implicit barriers and group creation as in E and Fork when mixing asynchronous and synchronous code.

This section gave an overview of the language from this point of view. However, lots of related details had to be left out of the scope of this paper. For example, the conditions C_T and C_L are closely tied to the runtime task system and no other code should activate the conditions. Enabling condition C_1 requires more static analysis and in the preliminary compiler it is tied to the split construct not covered here. While designing the system, it became apparent that the previous set of control constructs might be too limited if we want to support more fine-grained control operations such as if-then-else with C_T used by a single branch and the subsequent code. In case more conditions are added to the system, there also needs to be a more structured way of categorizing the conditions into classes to make sure the existing rules are applicable to new conditions. For example, some conditions can be multiplied as branching occurs, others pick at most one branch.

4 Programming a computational kernel

In Figure 2 we give a brief example of the synchronicity model with a computational kernel utilizing both synchronous and asynchronous regions of code and also multi-operations. The simple example divides the control flow into three parts, threads $0 \dots 4$ get a new reverse numbering $10 \dots 6$. The first two threads execute an asynchronous region, continue with a parallel multi-operation. The result of the branch is printed once. The next three threads enter the second branch, print a single result of 0. Finally the whole program terminates.

To give a better idea of the coding style employed by each of the languages, the code listings begin with definitions for certain library routines used in the example. In practical parallel programming, the focus is less on generic library code and more in the computational kernel since the library code is part of the standard toolchain distribution and less prone to change, if at all. While the definitions in the example written in E look like normal C, both Replica and Fork examples are annotated with constraints.

In Replica code we use the constraints C_T, C_S, and C_1 introduced in Section 3.1. Since the language compilers may expect ASCII input, we denote the subscript with a leading underscore. The *fast_multi* function expects the fast synchronization token and will not operate in this example when it is not being called from a task parallel worker function initiated by the task parallel runtime system. Since the *printf* function writes the output to standard output or screen, it expects a single thread of control – otherwise the output could be messed up badly. The function *disrupt* is a function that makes the execution asynchronous.

In Fork, each type of region with differing synchronicity needs to be manually annotated with a block level annotation (`start, farm, seq`) while in Replica

```
kernel.replica:

@{+C_S} @{+C_T}
@{C_S+} @{C_T+} void
  fast_multi(int,int*,int);
@{+C_1} @{C_1+} void
  printf(string, int);
@{C_S-} void
  disrupt();
void multi(int,int*,int);

void main() {
  int i = $$;
  if ($$ < 5) {
   $ = 10 - $;
   if (i < 2) {
    int tmp = 0;
    disrupt();
    // sync; implicitly added

    // fast_multi(MADD,
    //   &tmp, $); not allowed
    multi(MADD, &tmp, $);
    printf("#1, sum %d", tmp);
   } else {
    printf("#2, sum %d", 0);
   }
}}}
```

```
kernel.e:

// calls can be unsafe
#define fast_multi(
  int,int*,int) ...

void printf(string, int);
void disrupt();
#define multi(
  int,int*,int) ...
```

```
kernel.fork:

sync void mpadd(int *,int);
async void disrupt();

void main() {
 start {
  int i = $$;
  if ($$ < 5) {
   $ = 10 - $;
   if (i < 2) {
    int tmp = 0;
    farm disrupt();
    mpadd(&tmp, $);
    seq printf(
    "#1, sum %d", tmp);
   } else {
    seq printf(
    "#2, sum %d", 0);
   }
  }
 }
}
```

```
void main() {
  int i = $$;
  if ($$ < 5) {
   $ = 10 - $;
   if (i < 2) {
    int tmp = 0;
    disrupt();
    sync;
    // allowed (but illegal)
    fast_multi(MADD, &tmp, $);
    multi(MADD, &tmp, $);
    printf("#1, sum %d", tmp);
   } else {
    printf("#2, sum %d", 0);
   }
}}}
```

Fig. 2. Examples of kernels using diverse forms synchronicity.

the region type is checked and inferred from the library function signatures. In E checking is omitted and even illegal fast multi-operations can be called without a special synchronicity token. Subgroup creation is carried out manually by renumbering the threads, but the implicit subgroup creation semantics leave untested in this example. In addition, performance considerations related to such implicit boilerplate code generation are unrealistic with simple examples demonstrating the language semantics and require a thorough analysis in the context of computational kernels once the Replica's inference system is optimized for code efficiency.

In this example, Replica's control flow model checks equal amount or more conditions than the two existing languages without opening new possibilities for evident synchronicity errors. It also enables moving verbose annotations from user code to the library, which supports our goals of designing a more expressive, generic and easy to use language for parallel programming. A thorough analysis with a larger set of real world benchmarks is still needed to validate our claims in a practical setting and clearly defined semantics for flow conditions in cases like recursion is still needed for parallel algorithms such as quick sort.

5 Conclusions and future work

In this paper, we presented a new framework for composable, hierarchical, synchronization support on REPLICA and other architectures with a similar kind of execution model. The preliminary framework and its implementation in our Replica language compiler consists of a definition of synchronicity using parallel control flows, a partially formal description of the rules for correctness, a simple checking algorithm, and ideas for doing preliminary condition inference.

We demonstrated the resulting language by comparing example code against equal, ported implementations in previous languages E and Fork. The example mainly focused on demonstrating the new programming model with each system and, for now, omitted necessary performance evaluation needed when dealing with practical HPC kernels. In the next version of the system, strong emphasis will be put on practical implementation aspects.

While the preliminary work on the model can be already used to check programs, as future work, we suggest extending the rules to cover the whole language and (task) parallel runtime system, studying and extending the inference algorithm to work in a wider set of cases, opening possibilities for program performance tuning. The set of useful parallel control abstractions may also be far from complete. Examples of possibly useful fine-grained abstractions were given.

Acknowledgment

This work was funded by VTT.

References

1. Garcia, F., Fernandez, J.: Posix thread libraries. Linux Journal **2000**(70es) (2000) 36
2. Dagum, L., Menon, R.: OpenMP: An Industry Standard API for Shared-Memory Programming. Computational Science & Engineering, IEEE **5**(1) (1998) 46–55
3. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. computer **29**(12) (1996) 66–76
4. Duranton, M., Black-Schaffer, D., Yehia, S., De Bosschere, K.: Computing systems: Reseach challenges ahead – the hipeac vision 2011/2012 (2012)
5. Forsell, M.: A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads. Int. Journal of Networking and Computing **1**(1) (2011) 21–35
6. Forsell, M.: TOTAL ECLIPSE – An Efficient Architectural Realization of The Parallel Random Access Machine. Parallel and Distributed Comput., Ed. A. Ros, IN-TECH, Wien (2010) 39–64
7. Mäkelä, J.M., Hansson, E., Forsell, M., Kessler, C., Leppänen, V.: Design Principles of the Programming Language Replica for Hybrid PRAM-NUMA Many-Core Architectures. In: Proceedings of 4th Swedish Workshop on Multi-Core Computing, Linköping University (2011) 136
8. Mäkelä, J.M., Hansson, E., Åkeson, D., Forsell, M., Kessler, C., Leppänen, V.: Design of the Language Replica for Hybrid PRAM-NUMA Many-Core Architectures. In Werner, B., ed.: 10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2012, IEEE (2012) 697–704
9. Kessler, C., Seidl, H.: The Fork95 Parallel Programming Language: Design, Implementation, Application. Int. Journal of parallel programming (1997)
10. Forsell, M.: E – A Language for Thread-Level Parallel Programming on Synchronous Shared Memory NOCs. WSEAS Trans. on Computers **3**(3) (jul 2004) 807–812
11. Forsell, M., Roivainen, J.: Supporting Ordered Multiprefix Operations in Emulated Shared Memory CMPs. In: Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11). Las Vegas, USA. (2011)
12. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable Parallel Programming with CUDA. Queue **6**(2) (2008) 40–53
13. Keller, J., Kessler, C., Träff, J.: Practical PRAM Programming. Wiley (2001)
14. Forum, M.P.: MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA (1994)
15. Vishkin, U., Dascal, S., Berkovich, E., Nuzman, J.: Explicit Multi-threaded (XMT) Bridging Models for Instruction Parallelism. In: Proc. 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA). (1998) 140–151
16. Kessler, C., Hansson, E.: Flexible Scheduling and Thread Allocation for Synchronous Parallel Tasks. In: Proc. of 10th Workshop on Parallel Systems and Algorithms (PASA'12). (2012)
17. Darlington, J., Field, A., Harrison, P., Kelly, P., Sharp, D., Wu, Q., While, R.: Parallel Programming Using Skeleton Functions. In: PARLE'93 Parallel Architectures and Languages Europe, Springer (1993) 146–160
18. Wadler, P., Blott, S.: How To Make Ad-Hoc Polymorphism Less Ad Hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '89, New York, NY, USA, ACM (1989) 60–76
19. Kernighan, B., Ritchie, D.: The C Programming Language",(\ ANSI C"). Prentice Hall,(ISBN: 0-13-110362-8) (1988)

Checking visual data flow programs with finite process models

Jyrki Nummenmaa[1], Maija Marttila-Kontio[2], and Timo Nummenmaa[1]

[1] School of Information Sciences, University of Tampere, Finland
[2] School of Computing, University of Eastern Finland, Finland
jyrki.nummenmaa@uta.fi, maija.marttila@uef.fi, timo.nummenmaa@uta.fi

Abstract. A visual data flow language (VDFL) allows graphical presentation of a computer program in the form of a directed graph, where data tokens travel through the arcs of the graph, and the vertices present e.g. the input token streams, calculations, comparisons, and conditionals. Amongst their benefits, VDFLs allow parallel computing and they are presumed to improve the quality of programming due to their intuitive readability. Thus, they are also suitable for computing education. However, the token-based computational model allowing parallel processing may make the programs more complicated than what they look. We propose a method for checking properties of VDFL programs using finite state processes (FSPs) using a commonly available labelled transition system analyser (LTSA) tool. The method can also be used to study different VDFL programming constructs for development or re-design of VDFLs. For our method, we have implemented a compiler that compiles a textual representation of a VDFL into FSPs.

1 Introduction

In computing studies, programs are often visualized with flow-charts or similar, demonstrating the program flow and giving a visual representation for the basic concepts of programming, such as choice, iteration, and input introduction. Visual data flow languages (VDFLs) are special visual programming languages [7]. The programs are presented in the form of directed graphs. The vertices present such functionalities as e.g. calculations, comparisons, and conditionals. The data flow in the form of data tokens in the arcs from functionalities to other functionalities.

The visual nature of the VDFL programs makes them intuitive and is supposed to increase their understandability [4] [7] [13]. This is an important factor, when the quality of software is a crucial factor. Furthermore, a system based on the data flow execution paradigm has advantages such as easier program verification, better modularity and extendability of hardware, reduced protection problems and superior confinement of software errors [1].

The computational model, based on the function application, allows parallel and concurrent execution strategies for the programs [2] [11]. Even though this is an important and desired feature, at the same time this increases the complexity of the program, as it may be hard for the programmer to understand and anticipate the different execution sequences allowed by the VDFL program.

A formal method is a set of tools and notations (with a formal semantics) used to specify unambiguously the requirements of a computer system, supporting the proof of properties of that specification and proofs of correctness of an eventual implementation with respect to that specification [8]. Formal methods are commonly used to study the properties of computational systems allowing concurrent or parallel processing. However, to our knowledge there has been little interest to apply formal methods to study the properties of VDLF programs. The work of Marttila-Kontio et al. [10] established a mapping between VDFL programs and actions systems. This mapping provides a possible path for reasoning on VDFL programs.

When applying formal methods, a formal specification can be created. The formal specification should precisely state what the piece of software being specified is supposed to do [6]. In this work, we propose to use finite state processes as formal specifications to represent and analyse the processing of VDFL programs. In our work, we limit our attention to a basic VDFL language, which allows the definition of basic constructs: inputs, outputs, conditionalities and basic computations, and iteration, which comes by the cyclic structure of the programs. Even though VDFL programs are visual by nature, for the analysis of the programs, only non-visual information is needed. We have implemented a simple compiler, which compiles a textual representation of a VDFL program into finite state processes (FSPs). The FSPs can then be automatically analysed by a labelled transition system analyser (LTSA) tool for certain safety properties, such as deadlocks, and certain reachability properties, such as terminal sets. It is also possible to state explicit progress properties about the FSP model, and to check those using the LTSA tool. The book by Magee and Kramer [9] explains the usage of the tool and formalism we use for the analysis and modeling of the FSPs.

There is some previous work in the area, where formal approach has been used to study visual programing or modeling languages. E.g. Zhang et al. [14] have studied the visual language semantics specification using grammatical treatment of the programming language structures, and Gostagliola et al. [3] have studied the use of grammars in the development of visual modeling languages. Even though we utilize grammars and parsing in our work, our goal is different: We aim for a formal model that can be used as an input for a tool that can automatically check certain formal properties of the program.

The content of the rest of the paper is as follows. Section 2 presents our VDFL model. Section 3 describes how FSPs are compiled out of the VDFL model. Section 4 discusses the applicability of our work in the context of a commercial VDFL system, LabView [13]. Section 5 explains how our approach can be utilized in the context of visual language development. Section 6 contains concluding remarks.

2 The basic VDFL model

Our version is quite a minimal version of a visual data flow language, however, it contains the necessary basic building blocks that allow to specify basic VDFL programs. We will introduce the elements of the VDFL and their grammatical representation, which is used in the process of the compilation. For readability, our syntax is not as concise as would be possible. The operations are selected from the set introduced by Davis and Keller [5].

A VDFL program can be represented as a directed graph, where arcs denote the data channels (or data wires) by which data tokens flow along the direction of the arcs, and the nodes represent inputs, outputs, and functions of the program. The tokens contain values, which are used in the computations. In our example, we limit ourselves to integer and Boolean values, even though in many practical applications other data types, simple and structured, would be used. To feed the intuition of the reader, we give a visual example of a simple VDFL program in Figure 1.

In that program, X and Y feed input tokens to the system. The tokens go to a "less than" comparison, the result of which is taken to the Selector node. If the result of the comparison is True, then the Y valued token, fed to the T-marked entrance to the Selector, will be passed to the Result, and otherwise the X valued token from the F-marked entrance will be passed to the Result. So, the value of Result will be Y, if X is smaller than Y, and X otherwise.

Let us now consider our modelling primitives for a basic VDFL. The primitives are given with grammatical representations, as our system will utilize textual input, which could be exported from a graphical programming environment.

The arcs are defined by a simple statement

```
"channel" Ident
```

where `Ident` denotes an identfying name. These names are then used to refer to the arcs.

The source token streams are defined by

```
"source" Ident "to" [Ident]
```

where the `Ident` gives a name to the source (completely documentary and not used in the computation) and the `[Ident]` specifies a list of names of the arcs to which the source stream feeds the tokens. This way, it is possible to send the values to several places, as will be done also with several other structures.

The desired result from the computation is specified as

```
"result" ":" Ident
```

where the Ident specifies the arc that is feeding the result (output) tokens.

In addition to variables, whose values are not supposed to be known by the time the program is specified, it is possible to introduce constants:

```
"constant" Integer "to" [Ident]
```

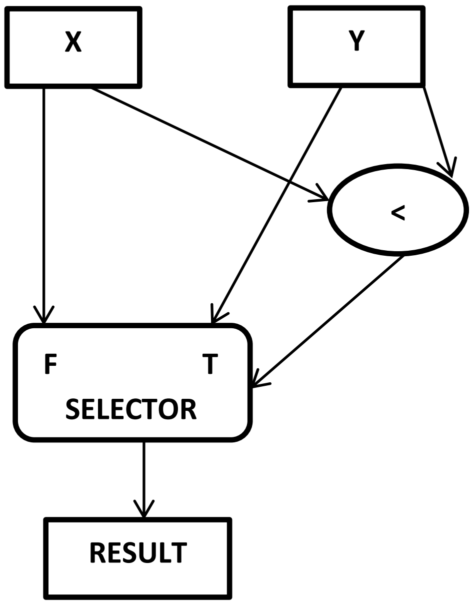


Fig. 1. A simple VDFL program

where `Integer` specifies the constant value, and `[Ident]` specifies the arcs to which the stream of constant tokens are fed.

Binary arithmetics are specified with statements of the form:

```
"bin_arith" Ident BOper Ident "to" [Ident]
```

where `BOper` specifies the binary operator, such as `*` or `+`, and the first and second `Ident` specify the input arcs, and the token resulting from the arithmetic operation is fed into the arcs specified by `[Ident]`.

In the similar way, comparisons are specified with statements of the form:

```
"compare" Ident COper Ident "to" [Ident]
```

where `COper` specifies the comparison operator, one of `<`, `<=`, `=`, `>=`, or `>`, and the first and second `Ident` specify the left and right input arcs, respectively, and the token resulting from the arithmetic operation is fed into the channel specified by the third `Ident`. It should be noted that in the visual representation, the input arcs are read from left to right, and this needs to be converted into text respectively.

According to the standard computational model, once there are input tokens in the incoming arcs, the function of an edge can be performed and the result can be passed on. In this work, we assume that only one token fits into one arc, that is, if the outgoing arc is still occupied by a previously produced token, a function is not going to output more tokens to the arc. The same assumption was made by Davis and Keller. As for the source/input tokens, we just assume that when there is space in the arc and input is available, a token will be positioned in the arc. As for the result/output, we assume that the user or the process using the program will need to explicitly consume the result/output tokens before a next result is produced.

The structures introduced this far are just data flow based counterparts of standard computational operations. However, VDFL programming includes some features that are specific to the computational model. The *selector* has two standard incoming arcs (of course in our case limited to integers and Booleans), one input arc that brings in Boolean (selector) tokens, and a set of outgoing arcs. The two standard inputs are labeled by True and False. If the selector value is True, then a token from the True-labelled input stream is passed to the outgoing arcs, and if the selector value is False, then a token from the False-labelled input stream is passed to the outgoing arcs. The selector is specified as follows, where it should be evident how True-labelled and False-labelled arcs map to the *Ident* elements.

```
"selector" "if" Ident "then" Ident "else" Ident "to" [Ident]
```

The selector is a bit more complicated structure for the computational model. First of all, it is in principle capable of executing even if it does not have tokens in all input arcs, that is, if the selector input is False, then the True input is not needed, and vice versa. Also, it could only consume a token from one input, thus leaving one token in its arc. Both of these assumptions imply some complications, so we abandon them and in our work the assumption is that the selector only executes, when all inputs are present, and they are all consumed.

Let us now reconsider the program of Figure 1. There are two sources, X and Y. Due to the computational model, once X and Y produce tokens, they will go to the Selector and the comparison (smaller than) node. However, the selector can only execute once the comparison is done. If X is smaller than Y, the result will be Y, and otherwise X. Notably, the figure contains graphical layout information about the parameters, the smaller than comparison is read from left to right, and the True and False incoming arcs in the Selector are placed at the T and F symbols. The graphical layout information is, of course, coded in the textual representation that we use. This needs some bookkeeping in the VDFL development environment. The program, expressed in textual form, is given below. The first line has a program name, which would also be extracted from the VDFL environment.

```
program comp :
channel ch1
channel ch2
channel ch3
channel ch4
channel ch5
channel ch6
source X to ch1, ch2
source Y to ch3, ch4
compare ch1 < ch3 to ch5
selector if ch5 then ch4 else ch2 to ch6
result : ch6
```

3 Compilation of VDFL programs into FSPs

This section explains the FSP structures used to represent the VDFL programs, introduces the implementation technology used in VDFL compilation, and discusses the choices done in the implementation.

Before discussing how we have mapped the VDFL programs into FSPs, we give a very short introduction to FSPs. FSPs are composed of actions using sequentiality (`->`), choice (`|`), guards (`when (boolean_cond)`), the if-else structure (`if (boolean_cond) then else`), parallel composition (`||`) and other operations, not used here. The process names start with uppercase and the action names with lowercase. The action names can be indexed, e.g. with values that are related to the action. When we use integer indexing and a new value is introduced, a range for the possible values needs to be given. The generated code defines a range `MaxInt = 0..1` which, of course, can be changed by the user. This way, we can define e.g. the process `CHANNEL = (in_chan1[i:MaxInt]-> out_chan2[i] -> CHANNEL)` to say that the channel can produce an alternation of taking in a token with index from the given range and once i is thus fixed, it will produce an out action that has the now fixed index, and after that it can start again and a new index can be chosen.

Parallel composition is used to combine processes, e.g. we combine process `CONSTANT = (in_chan1[1] -> CONSTANT)` with `CHANNEL`, given above:

```
||CHANNEL_WITH_CONSTANT = (CONSTANT || CHANNEL).
```

and now the common actions can only be executed when both `CHANNEL` and `CONSTANT` can execute them. This generally limits the possible execution traces, and may lead to deadlock ie. a situation where there are no actions availabe for execution. In this case, the process `CONSTANT` is limited to execute `in_chan_1[1]` only when `CHANNEL` is also ready to execute it. Notably, since this is the only value available, the parameter in `CHANNEL = (in_chan1[i:MaxInt]-> out_chan2[i] -> CHANNEL)` is always bound to 1.

Even though the VDFL programs we used as examples seem limited for their data types, only including an integer and Boolean datatype, in practice general integers lead into state-space explosion, and their values need to be strongly limited. In this work, we will only include binary integer values in the models. Even though slightly bigger values would in practice be feasible, this choice will simplify our models and processing, still giving a possibility to analyze certain part of the behavior of the programs.

This way, a channel / arc of the program is implemented as a process, where Ident stands for the channel identifying name.

```
CHANNEL_Ident = (in_Ident[i:MaxInt] -> out_Ident[i]
 -> CHANNEL_Ident).
```

With this process definition, the channel admits a token with a value from range MaxInt and then the same token value needs to go out of the channel before the next token can enter. This is exactly how the arcs should work under the assumption that the arc can hold at most one value at all times.

The source simply needs to input a token into a list channels. This is achieved by:

```
SOURCE_Ident = (value_Ident[i:MaxInt] -> in_Ident1[i]
  ->... -> in_Identk[i] -> SOURCE_Ident).
```

In this FSP `Ident` is the identifying name of the channel into which the source outputs tokens, and `Ident1, ..., Identk` are the identifying names of the channels to which the token will be put. This way the action of the source synchronizes with the related channels actions to take in tokens. CONSTANT is just like SOURCE apart from the index value for the in-action is fixed. Even though this means that the channels need to take in the values in the order specified by the identifiers, this makes no difference in practice.

Comparisons take two input values and produce an output value, either 0 or 1, representing False and True. Here, we use the simple `if-then-else` structure of the FSPs that allows to define conditional behavior. Since the assumption is that both input values need to exist, it does not matter in which order the comparison takes them in.

```
COMP_Ident1_Ident2 = (out_Ident1[i:MaxInt] -> out_Ident2[j:MaxInt]
 -> if (i COper j) then (in_Ident3[1] -> COMP_Ident1_Ident2)
     else (in_Ident3[0] -> COMP_Ident1_Ident2)).
```

Binary arithmetic is even simpler than comparison, it takes the two input value tokens and passes on the result of the arithmetic expression, which is easy to generate.

```
BOPER_Ident1_Ident2 = (out_Ident1[i:MaxInt] -> out_Ident2[j:MaxInt]
          -> in_Ident3[i BOper j] -> BOper_Ident1_Ident2).
```

Finally, the selector process takes in the Boolean (0 or 1) token and the tokens from which to select, and then puts the appropriate token in the output channel:

```
SELECTOR_Ident1 = (out_Ident1[i:0..1] -> out_Ident2[j:MaxInt]
  -> out_Ident3[k:MaxInt] ->
    if (i==1)  then (in_Ident4[j] -> SELECTOR_Ident1)
                  else (in_Ident4[k] -> SELECTOR_Ident1).
```

Our compiler implementation is based on the use of the BNF Converter (BNFC) [12], a compiler construction tool that is given a labelled BNF grammar, produces various useful artefacts. Our compiler is implemented using the Haskell programming language, so we use the following files, generated by BNFC.

– A parser generator file for Happy. The file can directly be used to generate a Happy parser. Happy is a parser generator that comes as a part of the Haskell Platform environment.
– A lexer generator file for Alex . The file can directly be used to generate an Alex lexer. Alex is a lexical analyser generator that comes as a part of the Haskell Platform environment.
– A test program to parse source language inputs and to pretty-print out the parse tree. Since the actual compilation needed is very simple, we have managed to create a compiler just by modifying these source files to eliminate some debug output and to generate the necessary code.

Since we are basically developing a method that would use output from a VDFL development environment, the error management is not a central issue in the compiler. We may assume that these environments print out the program information in the correct form. The top level of the labelled BNF grammar is largely introduced in the grammar snippets we have given. The whole labelled BNF grammar is given below.

```
Prog. Program  ::= "program" Ident ":" [Stm] ;
Chan. Stm  ::= "channel" Ident ;
Sour. Stm  ::= "source" Ident "to" [Ident] ;
Cons. Stm   ::= "constant" Integer "to" [Ident] ;
BAri. Stm  ::= "bin_arith" Ident BOper Ident "to" [Ident]  ;
UAri. Stm  ::= "un_arith" UOper Ident "to" [Ident]  ;
Comp. Stm  ::= "compare" Ident COper Ident "to" [Ident]  ;
Sel. Stm   ::= "selector" "if" Ident "then" Ident
                          "else" Ident "to" [Ident] ;
Dist. Stm  ::= "distributor" "if" Ident "then" Ident "to" Ident
                                            "else" "to" [Ident] ;
Res. Stm  ::= "result" ":" Ident ;
separator Stm "" ;
separator Ident "," ;
EEq.     COPer ::= "==" ;
ENeq.    COper ::= "!="  ;
ELeq.    COper ::= "<="  ;
ELt.     COper ::= "<"  ;
EGeq.    COper ::= ">="  ;
```

```
  EGe.     COper ::= ">"  ;
  EAdd.    BOper ::= "+"  ;
  ESub.    BOper ::= "-"  ;
  EMul.    BOper ::= "*"  ;
  EDiv.    BOper ::= "/"  ;
```

Thus, the program of Figure 1 compiles into the following code, where the compilation rules are given above, apart from the fact that we need a final definition that composes everything into the common model.

```
range MaxInt = 0..1
CHANNEL_ch1 = (in_ch1[i:MaxInt] -> out_ch1[i] -> CHANNEL_ch1).
CHANNEL_ch2 = (in_ch2[i:MaxInt] -> out_ch2[i] -> CHANNEL_ch2).
CHANNEL_ch3 = (in_ch3[i:MaxInt] -> out_ch3[i] -> CHANNEL_ch3).
CHANNEL_ch4 = (in_ch4[i:MaxInt] -> out_ch4[i] -> CHANNEL_ch4).
CHANNEL_ch5 = (in_ch5[i:MaxInt] -> out_ch5[i] -> CHANNEL_ch5).
CHANNEL_ch6 = (in_ch6[i:MaxInt] -> out_ch6[i] -> CHANNEL_ch6).
SOURCE_X = (value_X[i:MaxInt] ->  in_ch1[i] ->  in_ch2[i] ->
                    SOURCE_X).
SOURCE_Y = (value_Y[i:MaxInt] ->  in_ch3[i] ->  in_ch4[i]  ->
                    SOURCE_Y).
COMP_ch1_ch3 = (out_ch1[i:MaxInt] -> out_ch3[j:MaxInt] ->
 if (i<j) then ( in_ch5[1] ->  COMP_ch1_ch3)
 else ( in_ch5[0] ->  COMP_ch1_ch3)).
SELECTOR_ch5 = ( out_ch5[i:0..1] -> out_ch4[j:MaxInt] ->
  out_ch2[k:MaxInt] -> if (i==1)
  then ( in_ch6[j] ->  SELECTOR_ch5)
  else ( in_ch6[k] ->  SELECTOR_ch5)).
RESULT_ch6 = (out_ch6[i:MaxInt] -> RESULT_ch6).
||SYSTEM = (CHANNEL_ch1 || CHANNEL_ch2 || CHANNEL_ch3
  || CHANNEL_ch4 || CHANNEL_ch5 || CHANNEL_ch6
  || SOURCE_X || SOURCE_Y || COMP_ch1_ch3
  || SELECTOR_ch5 || RESULT_ch6).
```

This specification model can now be pasted into the LTSA tool, and the tool can be used to check for error / undefined states, and deadlocks. There are none. Also, the tool can be used to check if there are terminal sets, that is, if the execution will eventually cycle in just a subset of states. In this case, also this does not happen.

The LTSA tool can also be used to execute the model step-by-step, thereby generating an action trace. The following is an example trace from an execution with the LTSA tool.

```
 value_X.1 -> value_Y.0 -> in_ch1.1 -> in_ch2.1 -> in_ch3.0 ->
   in_ch4.0 -> out_ch1.1 -> out_ch3.0 -> in_ch5.0 -> out_ch5.0 ->
   out_ch4.0 -> out_ch2.1 -> in_ch6.1 -> out_ch6.1
```

4 LabView

LabView [13] is a commercial VDFL system. To show how our work maps to a real commercial system, we will discuss some LabView code samples. It should be noted that LabView has additional features and not only the ones presented until now, so naturally covering all of them would need additional structures to our grammar and compilation.

Four separate LabVIEW codes are illustrated in Figure 2. The first code represents a simple summation operation. When the program executes, the Add node waits until it has received values from the data channels attached to it. The data channel between the control A and the indicator B is populated by a token right after the user has given an input value to the control A. The lower data channel is immediately populated by the constant value (4) when the program executes. The indicator B represents a user interface VI for presenting the result of the summation operation to the user. Colours in LabVIEW program represent different data types. Integer values are represented by blue colours, whereas green colour is for Boolean values. It would be straightforward to represent this code sample with our grammar.

The second code contains a basic For loop. The constant value (3) is attached to the For loop's input terminal N which represents the number of iterations. Inside the For loop is the Round indicator presenting the iteration round (0,1, and finally 2) on the user interface. The code sample is implementable using our grammar, but a direct For loop structure provides convenience to the programmer. Using the grammar we have used above in our paper, it would be necessary to add 1 to the counter on every round and to compare it to the number of rounds, to know when the result of the iteration should be passed further on.

In the third code, the Select node waits until the integer controls C and D, and the Boolean control "True or False?" have got new data given by a user. If the user enters True to the control, the select node passes the value from C to the indicator E. If False is entered, the value from D is passed to the indicator E. Both data channels become empty after the Select node executes. This is the same as the Selector discussed in this paper.

The fourth code represent the case structure. Here, the case structure is dependent on Boolean values, but other data types can be attached to the "?"-terminal as well. The case structure only executes once the user enters a value to the Multiply? control. The multiplication inside the True case is performed immediately when the Multiply? control has True value. In False case, the program stops. The False case is not visible in the code because LabVIEW shows only one case at a time. Implementation of this type of conditionality would need more building blocks in our grammar.

The basic primitives we have used for VDFLs match reasonably well with practical examples, however, it is easy to see that for the programmer's convenience, certain higher-level structures would be useful (even if they can decomposed into our initial modeling primitives). The examples also demonstrate that in LabView the user may give input as direct manipulation. This is another feature not covered by our model, however the Source definitions we have used

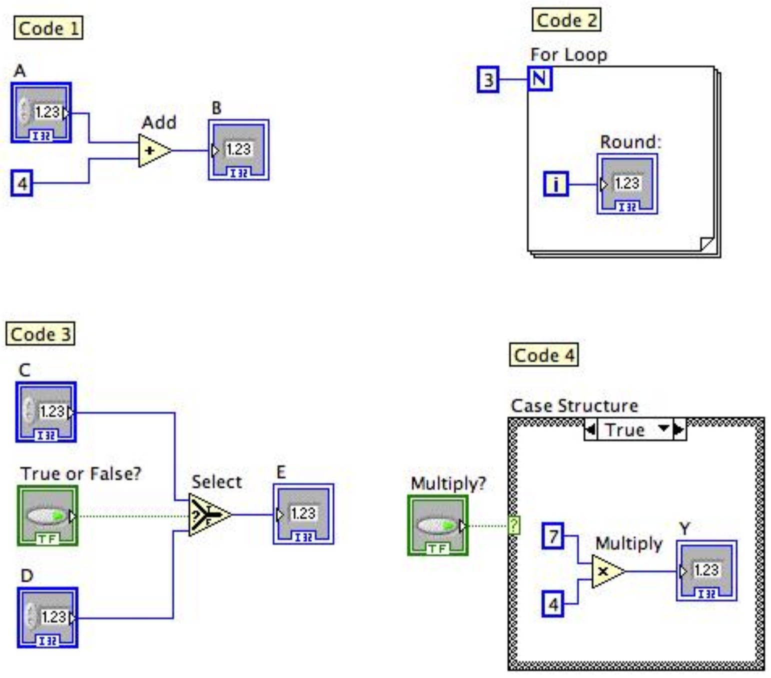


Fig. 2. LabView program examples

are not dependent on how the input values enter the computation system, be it through e.g. some device giving physical readings, or an end-user feeding the data through a user interface.

5 Language development

In this section we discuss how our work can be applied in the development and specification of a VDFL. The idea is simple: Once a new language feature is designed, it can be implemented as a part of the grammar of expressions, and example programs can be tried out to test for possible unexpected and unwanted phenomena and side effects. We exemplify this by extending the language presented this far. However, we point out that all steps we have produced are fairly straighforward for other, similar languages, and the same approach could be used with some changes to the textual representation of the language and related changes to the compilation.

As an example, let us consider adding the *Distributor* structure [5] to our grammar. The distributor has one standard incoming arc (of course in our case limited to integers and Booleans), one input arc that brings in Boolean tokens, and two outgoing arcs, labelled as True and False. If the selector value is True, then a token from the standard input stream is passed to the outgoing True arc, and if the selector value is False, then a token from the standard input stream is passed to the outgoing False arc. The distributor is specified as follows, where, again, it should be evident how True-labelled and False-labelled arcs map to the *Ident* elements. In practice, there could be many outgoing True and False arcs, but this is enough for our purposes now.

```
"distributor" "if" Ident1 "then" Ident2 "to" [Ident3]
                                "else" "to" [Ident4]
```

The distributor would be represented as follows in the FSP model, in the case that outgoing arc lists have just one arc each. The extension to a list is straightforward with a sequence of actions.

```
DISTRIBUTOR_Ident1 = (out_Ident1[i:0..1] -> out_Ident2[j:MaxInt]
-> if (i==1)  then (in_Ident3[j] -> DISTRIBUTOR_Ident1)
                  else (in_Ident4[k] -> DISTRIBUTOR_Ident1).
```

The reader might guess, by now, that the distributor may be problematic as an operation, because it does not put data into all of the channels. Below is a sample program that uses the distributor.

```
program dist :
channel ch1
channel ch2
channel ch3
channel ch4
channel ch5
channel ch6
channel ch7
channel ch8
source X to ch1, ch2
source Y to ch3, ch4
compare ch1 < ch3 to ch5
distributor if ch5 then ch2 to ch6 else to ch7
compare ch4 < ch7 to ch8
result : ch8
```

The compiled FSP model is as follows.

```
range MaxInt = 0..1
CHANNEL_ch1 = (in_ch1[i:MaxInt] -> out_ch1[i] -> CHANNEL_ch1).
CHANNEL_ch2 = (in_ch2[i:MaxInt] -> out_ch2[i] -> CHANNEL_ch2).
CHANNEL_ch3 = (in_ch3[i:MaxInt] -> out_ch3[i] -> CHANNEL_ch3).
CHANNEL_ch4 = (in_ch4[i:MaxInt] -> out_ch4[i] -> CHANNEL_ch4).
CHANNEL_ch5 = (in_ch5[i:MaxInt] -> out_ch5[i] -> CHANNEL_ch5).
CHANNEL_ch6 = (in_ch6[i:MaxInt] -> out_ch6[i] -> CHANNEL_ch6).
CHANNEL_ch7 = (in_ch7[i:MaxInt] -> out_ch7[i] -> CHANNEL_ch7).
CHANNEL_ch8 = (in_ch8[i:MaxInt] -> out_ch8[i] -> CHANNEL_ch8).
SOURCE_X = (value_X[i:MaxInt] ->  in_ch1[i] ->  in_ch2[i] ->
     SOURCE_X).
SOURCE_Y = (value_Y[i:MaxInt] ->  in_ch3[i] ->  in_ch4[i] ->
     SOURCE_Y).
COMP_ch1_ch3 = (out_ch1[i:MaxInt] -> out_ch3[j:MaxInt] ->
     if (i<j) then ( in_ch5[1] ->  COMP_ch1_ch3)
              else ( in_ch5[0] ->  COMP_ch1_ch3)).
DISTRIBUTOR_ch5 = ( out_ch5[i:0..1] -> out_ch2[j:MaxInt] ->
                  if (i==1) then ( in_ch6[j] ->  DISTRIBUTOR_ch5)
                  else ( in_ch7[j] ->  DISTRIBUTOR_ch5)).
COMP_ch4_ch7 = (out_ch4[i:MaxInt] -> out_ch7[j:MaxInt] ->
         if (i<j) then ( in_ch8[1] ->  COMP_ch4_ch7)
         else ( in_ch8[0] ->  COMP_ch4_ch7)).
RESULT_ch8 = (out_ch8[i:MaxInt] -> RESULT_ch8).
||SYSTEM = (CHANNEL_ch1 || CHANNEL_ch2 || CHANNEL_ch3
  || CHANNEL_ch4 || CHANNEL_ch5  || CHANNEL_ch6 || CHANNEL_ch7
  || CHANNEL_ch8 || SOURCE_X  || SOURCE_Y || COMP_ch1_ch3
  || DISTRIBUTOR_ch5 || COMP_ch4_ch7 || RESULT_ch8).
```

When tested with the LTSA tool, the tool identifies a potential deadlock and shows a trace to the deadlock. The trace has over 40 actions. The experiment is easy to repeat. This cannot be seen as firm evidence against the distributor structure, particularly as our program was a bit carelessly programmed, but it does show that our approach can be used to pin-point risky situations, where either one needs to be careful with the language structures, the programs need to be analyzed carefully before execution, or the language constructs need re-consideration.

6 Conclusions

We propose a method to automatically analyze VDFL programs using finite state processes. For this, we have implemented a simple compiler that compiles our example VDFL programs from textual representation into FSPs, in a form that can readily be read into an analyzer program. Our method is, in principle, aimed for a toolset for VDFL program development. However, the method can also be used in new language design. By first implementing the new language features in our system, programs utilizing the new features can be analyzed.

Due to potential state-space explotion, one needs to be careful about the integer value ranges used in the analysis models. In practice, they usually need to be very small.

References

1. T. Agerwala and Arvind. Data flow systems: Guest editors' introduction. *Computer*, 15(2):10–13, February 1982.
2. John Backus. Acm turing award lectures. chapter Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. ACM, New York, NY, USA, 2007.
3. Gennaro Costagliola, Vincenzo Deufemia, and Giuseppe Polese. A framework for modeling and implementing visual notations with applications to software engineering. *ACM Trans. Softw. Eng. Methodol.*, 13(4):431–487, October 2004.
4. Lorrie Cranor and Ajay Apte. Programs worth one thousand words: visual languages bring programming to the masses. *Crossroads*, 1(2):16–18, December 1994.
5. A. L. Davis and R. M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, February 1982.
6. Antoni Diller. *Z - An introduction to formal methods.* John Wiley & Sons, 1990.
7. Daniel D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3:69–101, 1992.
8. M.G. Hinchey and J.P. Bowen. *Applications of formal methods.* Prentice-Hall international series in computer science. Prentice Hall, 1995.
9. Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs.* John Wiley & Sons, Inc., New York, NY, USA, 2006.
10. M. Marttila-Kontio, M. Ronkko, and P. Toivanen. Visual data flow languages with action systems. In *Computer Science and Information Technology, 2009. IMCSIT '09. International Multiconference on*, pages 589–594, 2009.
11. Walid A. Najjar, Edward A. Lee, and Guang R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25:1907–1929, 1999.
12. Aarne Ranta. *Implementing Programming Languages - An Introduction to Compilers and Interpreters.* Texts in Computing. College Publications, 2012.
13. Kirsten N. Whitley, Laura R. Novick, and Doug Fisher. Evidence in favor of visual representation for the dataflow paradigm: An experiment testing labview's comprehensibility. *Int. J. Hum.-Comput. Stud.*, 64(4):281–303, April 2006.
14. Ke-Bing Zhang, Mehmet A. Orgun, and Kang Zhang. Visual language semantics specification in the vispro system. In *Selected papers from the 2002 Pan-Sydney workshop on Visualisation - Volume 22*, VIP '02, pages 121–127, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

Efficient Saturation-based Bounded Model Checking of Asynchronous Systems

Dániel Darvas[1], András Vörös[1], and Tamás Bartha[2]

[1] Dept. of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
`vori@mit.bme.hu`
[2] Computer and Automation Research Institute
MTA SZTAKI,
Budapest, Hungary

Abstract. Formal verification is becoming a fundamental step in assuring the correctness of safety-critical systems. Since these systems are often asynchronous and even distributed, their verification necessitates methods that can deal with huge or even infinite state spaces. Model checking is one of the current techniques to analyse the behaviour of systems, as part of the verification process. The so-called *saturation* algorithm has an efficient iteration strategy combined with symbolic data structures, providing a powerful state space generation and model checking solution for asynchronous systems. In this paper we present the first approach to integrate two advanced saturation algorithms — namely *bounded saturation* and *constrained saturation-based structural model checking*— in order to improve on previous methods. *Bounded saturation* utilizes the efficiency of saturation in bounded state space exploration. *Constrained saturation* is an efficient structural model checking algorithm. Our measurements confirm that the new approach does not only offer a solution to deal with even infinite state spaces, but in many cases it even outperforms the original methods.

1 Introduction

Assuring the quality of safety critical, embedded systems is a challenging task. Advances in technology are making it even more difficult: components are becoming more complex, and systems have more components that interact using complicated communication and synchronisation mechanisms. Due to this complexity it is impossible to make claims about the correctness of these systems without the help of *formal methods*. On the other hand, exactly this complexity raised the need for highly efficient formal verification algorithms.

Formal verification usually starts with the creation of a formal model of the studied system. Then the behaviour of the formal model is analysed to prove its adequacy. One of the most prevalent analysis techniques is *model checking* [4], an automatic technique to check whether the model (and thus the modelled system) satisfies its specification. The specification is typically expressed in *temporal* *logic*. *Computation Tree Logic* (CTL) is a popular temporal logic language due to the efficient and relatively simple analysis algorithms supporting it.

Model checking traverses the state space of the model being analysed. Safety critical systems are often asynchronous, even distributed, so the composite state space of their asynchronous subsystems can be as large as the Cartesian product of the local components' state spaces, i.e., the state space of the whole system explodes. *Symbolic methods* [4] are advanced techniques to handle huge state spaces of synchronous systems. Instead of storing states explicitly, symbolic techniques rely on an encoded representation of the state space such as *decision diagrams*. These are compact graph representations of discrete functions. Ordinary symbolic methods, however, usually perform poorly for asynchronous systems.

Saturation [1] is considered as one of the most effective state space generation and model checking algorithms for asynchronous systems It combines the efficiency of symbolic methods with a special iteration strategy. Saturation-based state space exploration computes the set of reachable states. The so-called *saturation-based structural model checking* algorithm can analyse temporal logic properties. Nowadays, the so-called *constrained saturation-based structural model checking* algorithm is one of the most efficient algorithms for model checking [12].

However, many complex models still have a state space, which is either too large to be represented even symbolically, or it is infinite. In these cases *bounded model checking* can be a solution, as it explores and examines the prescribed properties on a bounded part of the state space. *Bounded saturation-based state space exploration* was introduced in [11], where the authors described a new saturation algorithm that explores the state space only to some bounded depth.

1.1 Motivation

Former approaches solved only one of the problems: they could either be used for structural model checking over the entire state space; or they could traverse the state space up to a given bound, but without being able to check complex properties on it. In this paper we introduce a new saturation-based bounded model checking algorithm that integrates both approaches. Our algorithm incrementally explores the state space and performs structural model checking on the uncovered bounded part. To our best knowledge, this is the first attempt to combine bounded saturation-based state space exploration with constrained saturation-based CTL model checking, in order to gain the advantages of both techniques.

Furthermore, bounded model checkers usually do not support full CTL. Even though there were theoretical results in this area, former bounded model checking approaches did not work well with CTL due to its branching characteristics. Our work is a step towards efficient bounded CTL model checking with many directions to be explored in the future.

This paper extends our former work [8] described in 3.1 with an efficient iteration strategy (namely constrained saturation) to traverse the bounded state space. This is the first time where the efficiency of constrained saturation based state space traversal is utilized for bounded model checking.

The structure of our paper is as follows: section 2 introduces the background and prerequisites of our work. Section 3 gives an overview of the advanced saturation-based algorithms our work relies on. Section 4 describes the new bounded CTL model checking algorithm and its details. Section 5 presents our measurements results. At the end our conclusions and ideas for future work complete the paper.

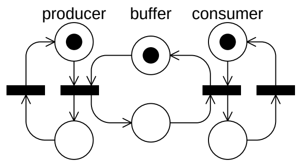
2 Background

In this section we outline the theoretical background of our work. First, we describe the underlying data structures of our algorithms for storing the state space during model checking: *Multiple-valued Decision Diagrams* (MDDs) and *Edge-valued Decision Diagrams* (EDDs). EDDs extend MDDs with extra information: in addition to storing the state space they also provide the distance information for bounded state space generation. Finally, we summarize the saturation-based state space exploration algorithm and the model checking background.

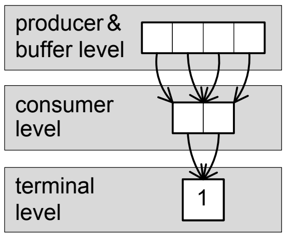
2.1 Decision Diagrams

This section is based on [10]. Decision diagrams are used in symbolic model checking for efficiently storing the state space and the possible state changes of the models. A *Multiple-valued Decision Diagram* (MDD) is a directed acyclic graph, representing a function f consisting of K variables: $f : \{0, 1, \ldots\}^K \to \{0, 1\}$. An MDD has a node set containing two types of nodes: non-terminal and two terminal nodes (terminal 0 and terminal 1). The nodes are ordered into $K+1$ levels. A non-terminal node is labelled by a variable index $1 \leq k \leq K$, which indicates to which level the node belongs (which variable it represents), and has n_k (domain size of the variable, in binary case $n_k = 2$) arcs pointing to nodes in level $k - 1$. A terminal node is labelled by the variable index 0. Duplicate nodes are not allowed, so if two nodes have identical successors in level k, they are also identical. These rules ensure that MDDs are canonical and compact representation of a given function or set. The evaluation of the function is the top-down traversal of the MDD through the variable assignments represented by the arcs between nodes.

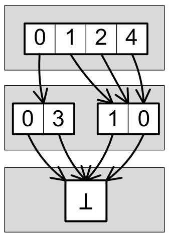
Figure 1(a) depicts a simple example Petri net [7] model of a producer-consumer system. The producer creates items and places them in the buffer, from where the consumer consumes them. For synchronizing purposes the buffer's capacity is one, so the producer has to wait till the consumer takes away the item from the buffer. This Petri net model has a finite state space containing 8 states. Figure 1(b) depicts an MDD used for storing the encoded state space of the example Petri net. Each edge encodes a possible local state [1], and the possible (global) states are the paths from the root node to the terminal *one* node. (The model has to be decomposed to be able to represent its state space using decision diagrams efficiently. This decomposition will be discussed in Section 2.3.)



(a) The Petri net of producer-consumer model



(b) State space representation with MDD



(c) State space and state distance representation with EDD

Fig. 1. Producer-consumer example

An *Edge-valued Decision Diagram* (EDD) is an extended MDD that can represent the following function: $f : \{0, 1, \ldots\}^K \rightarrow \mathbb{N} \cup \{\infty\}$. Figure 1(c) depicts an EDD storing the encoded state space enriched with the distance information (computed from the initial state). The differences between an MDD and an EDD are the following:

– Every p node is visualized as a rectangle with k slots, where k is the number of children (domain of the variable).
– On the terminal level there is only one terminal node, named $\perp$. This is equivalent to the terminal *one* node in an MDD.
– Every edge has a weight and a target node. The ith edge starts from the ith slot of the p node, and the value $p[i].value$ (the weight of the edge) is written to that slot. We write $\langle n, w\rangle$ if the edge has weight $w \in \mathbb{N} \cup \{\infty\}$ and has target node n. In addition, we write $p[i] = \langle n, w\rangle$ if the ith edge of the node p is $\langle n, w\rangle$ and $p[i].value \equiv w, p[i].node \equiv n$.
– If $p[i].value = \infty$, then $p[i].node = \perp$. This is equivalent to an edge in an MDD which goes to the terminal zero node. Usually the zero valued dangling edges and the ∞ valued edges are not shown.
– Every non-terminal node has an outgoing edge with weight 0.

In the example of Figure 1(c) let the node on the left side of the *consumer level* be x. This x node has two children: $x[0] = \langle\perp, 0\rangle$ and $x[1] = \langle\perp, 3\rangle$.

2.2 Model Checking and Bounded Model Checking

Given a formal model, *model checking* [4] is an automatic technique to decide whether the model satisfies the specification. Formally: let M be a Kripke structure (i.e., the model in the form of a labelled state-transition graph). Let f be a formula of temporal logic (i.e., the specification). The goal of model checking is to find all states s of M that $M, s \models f$.

Bounded model checking decides whether the model satisfies the specification in a predefined number of steps, i.e., the depth of the state space traversal.

Formally: let M be a Kripke structure, and f be a formula of temporal logic. The bounded model checking problem for the k-bounded state space is to find all states s of M such that $M, s \vDash_k f$. Among others, bounded model checking is useful when the full state space is not needed to decide on a property. This is e.g. the case for *shallow bugs* that can be found in a bounded state space quickly.

Structural model checking uses a set operations to evaluate temporal logic specifications by computing fixed-points in the state space. *CTL* (Computation Tree Logic) [4] is widely used temporal logic specifications formalism, as it has expressive syntax, and structural model checking yields efficient algorithms to analyse CTL specifications. CTL expressions contain state variables, Boolean operators, and *temporal operators*. Temporal operators occur in pairs in CTL: the path quantifier, either A (on all paths) or E (there exists a path), is followed by the tense operator, one of X (next), F (future, or finally), G (globally), and U (until). However, only three: EX, EU, EG of the 8 possible pairings need to be implemented due to duality [4]. The remaining five can be expressed with the help of the former three in the following way: $\mathsf{AX}p \equiv \neg\mathsf{EX}\neg p$, $\mathsf{AG}p \equiv \neg\mathsf{EF}\neg p$, $\mathsf{AF}p \equiv \neg\mathsf{EG}\neg p$, $\mathsf{A}[p\mathsf{U}q] \equiv \neg\mathsf{E}[\neg q\ \mathsf{U}(\neg p \wedge \neg q)] \wedge \neg\mathsf{EG}\neg q$, $\mathsf{EF}p \equiv \mathsf{E}[true\ \mathsf{U}\ p]$.

2.3 Saturation

Saturation is a *symbolic algorithm* for state space generation and model checking. *Decomposition* serves as the prerequisite for the symbolic encoding: the algorithm maps the state variables of the chosen high-level formalism into symbolic variables of the decision diagram. The global state of the model can be represented as the composition of the local states of components: $s_g = (s_1, s_2, \ldots, s_n)$, where n is the number of components. See Figure 1(b) for a possible decomposition and the corresponding MDD representation of the example model in Figure 1(a). Furthermore, decomposition helps the algorithm to efficiently exploit *locality*, which is inherent in asynchronous systems. Locality ensures that a transition usually affects only some components or some parts of the submodels. The algorithm does not create a large, monolithic next state function representation. Instead it divides the global next state function $\mathcal{N}$ into smaller parts, according to the high-level model. Formally: $\mathcal{N} = \bigcup_{e\in\mathcal{E}} \mathcal{N}_e$, where $\mathcal{E}$ is the set of events in the high level model. The granularity of the decomposition, i.e. the next state relations represented by $\mathcal{N}_e$ can be chosen arbitrarily [3].

Saturation uses *symbolic encoding of the next state function.* In our work we use the symbolic next state representation from [3]. This approach partitions disjunctively the global next state function according to the high level model events in the system: $\mathcal{N} = \bigcup_{e\in\mathcal{E}} \mathcal{N}_e$. Logically, if $\mathcal{N}$ is represented by the relation between state variables (in the decision diagram representation) $\boldsymbol{x}, \boldsymbol{x'}$ with $\mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x'})$, then the global relation can be expressed by the symbolic next state relations of the events: $\mathcal{R}(\boldsymbol{x}, \boldsymbol{x'}) = \bigvee_{e\in\mathcal{E}} \mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x'})$. This way the algorithm can use smaller next state representations. However, in many cases the computation of the local $\mathcal{N}_e$ functions is still expensive. The algorithm handles this problem by conjunctive partitioning according to the enabling and updating functions (denoted by $\mathcal{N}^{enable}$ and $\mathcal{N}^{update}$) [3]: $\mathcal{N}_e = \bigcap_{\forall i}(\mathcal{N}_{e,i}^{enable} \bigcap \mathcal{N}_{e,i}^{update})$, which can be symbolically computed by the following equation: $\mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x}') = \bigwedge_{\forall i}(\mathcal{R}_{e,i}^{enable}(\boldsymbol{x}, \boldsymbol{x}') \bigwedge \mathcal{R}_{e,i}^{update}(\boldsymbol{x}, \boldsymbol{x}'))$. Applying $\mathcal{N}_e$ to a given set of states represented by *states* results in $\mathcal{N}_e(\mathit{states}) = \mathit{RelProd}(\mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x}'), \mathit{states})$, where *RelProd* is the well-known relational product function [3]. The smaller the partitions we create, the less computation they need. The limit for the size of the partitioning comes from the used high level modelling formalism.

Saturation uses a *special iteration strategy*, which is efficient for asynchronous systems. The construction of the MDD representation of the state space starts by building the MDD representing the initial state. Then the algorithm saturates every node in a bottom-up manner, by applying saturation recursively, if new states are discovered. Saturation iterates through the MDD nodes and generates the whole state space representation using a node-to-node transitive closure. In this way saturation avoids the peak size of the MDD to be much larger than the final size, which is a critical problem in traditional approaches. The result is the state space representation encoded by MDD.

Saturation-based Structural Model Checking. Saturation-based structural CTL model checking was first presented in [2], where the authors introduced how the least fixed point operators can be computed with the help of saturation. CTL model checking explores the state space in a backward manner. It constructs the inverse representation $\mathcal{N}^{-1}$ and computes the inverse next state, greatest and least fixed points of the operators. The semantics of the three implemented CTL operators [4] is:

- **EX**: $i^0 \vDash \mathsf{EX}\, p$ iff $\exists i^1 \in \mathcal{N}(i^0)$ s.t. $i^1 \vDash p$. This means that EX corresponds to the function $\mathcal{N}^{-1}$, applying one step backward through the next state relation.
- **EG**: $i^0 \vDash \mathsf{EG}\, p$ iff $i^0 \vDash p$ and $\forall n > 0, \exists i^n \in \mathcal{N}(i^{n-1})$ s.t. $i^n \vDash p$ so that there is a strongly connected component containing states satisfying p. This computation needs a greatest fixed point computation, so that saturation cannot be applied directly to it. Computing the fixed point, however, benefits from the locality accompanying the decomposition.
- **EU**: $i^0 \vDash \mathsf{E}[p \,\mathsf{U}\, q]$ iff $i^0 \vDash p$ and $\exists n > 0, \exists i^1 \in \mathcal{N}(i^0), \ldots, \exists i^n \in \mathcal{N}(i^{n-1})$ s.t. $i^n \vDash q$ and $i^m \vDash p$ for all $m < n$ (or $i^0 \vDash q$). The states satisfying this property are computed with the following least fixed-point: $\mathbf{lfp}\ Z[q \vee (p \wedge \mathsf{EX}\, Z)]$ Informally: we search for a state q reached through only states satisfying p.

3 Bounded and Constrained Saturation

In this section we give an overview of the two saturation-based advanced algorithms that form important parts of our new approach. *Bounded saturation* is used for state space exploration. *Constrained saturation* is used to restrict structural model checking to the bounded state space. The integration of constrained saturation with the bounded saturation-based state space generation lead to the first saturation-based bounded model checking algorithm, which exploits the efficiency of structural model checking for bounded state spaces.

3.1 Bounded Saturation

It is difficult to exploit the efficiency of saturation for bounded state space exploration, because saturation uses an irregular recursive iteration order, which is totally different from traditional breadth-first traversal. Consequently, bounding the recursive exploration steps of saturation does not necessarily guarantee this bound to be global for the state space representation.

There are different solutions for the above problem in the literature, both for globally and locally bounded saturation-based state space generation. In our work we chose one that has already proved its efficiency [11]. Although MDDs provide a highly compact solution for state space representation, bounded saturation needs additional distance information during the traversal. For this reason, [11] uses Edge-valued Decision Diagrams (EDDs) instead of MDDs, and —in addition to the state space— it also encodes the minimal distance of each state from the initial state(s) into the EDD. The algorithm first iterates through the state space until a given bound is reached, which is represented by an edge in the EDD. After that it cuts the parts that are beyond the depth of the traversal from the EDD, thereby computing the reachability set below the bound.

In our previous work [10] we extended the algorithm [11] with on-the-fly updates [1] and an additional caching mechanism.

3.2 Constrained Saturation

In [12] the authors introduced an advanced saturation-based iteration strategy for the purpose of structural model checking. The algorithm, called *constrained saturation*, computes the least fixed point of the reachability relation that satisfies a given constraint.

The main novelty of the new algorithm is the slightly different iteration style. Instead of combining saturation with breadth-first traversal, it uses a pre-checking phase. The algorithm builds on the following observation [12]: in order to do the symbolic step $\mathcal{N}_e$ from the set of state *states* to a set of states satisfying the constraint $\mathcal{C}$, we have to compute $\mathcal{N}_e(\mathit{states}) \cap \mathcal{C}$. This contains an expensive intersection operation after each step. Using the following observation: $\mathcal{N}_e(\mathit{states}) \cap \mathcal{C} = \mathit{RelProd}(\mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x}'), \mathit{states}) \cap \mathcal{C} = \mathit{RelProd}(\mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x}') \wedge \boldsymbol{x}' \cap \mathcal{C} \neq 0, \mathit{states})$ the algorithm can use pre-checking phase and it avoids the computation-intensive intersection operation after the symbolic state space step, instead it simply skips those steps which would go out of the constraint [12].

Algorithms 1 and 2 formalize the operation of the constrained saturation algorithm. The lines starting with $*$ are the additions to traditional saturation. In Algorithm 1 it is easy to see that the *ConsSaturate*(c,s) computes $\mathit{RelProd}(\mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x}') \cap \mathcal{C}, \mathit{states})$ without using the expensive symbolic intersection operation. Research showed [12] that *ConsSaturate* is faster than traditional saturation when there is a constraint on the possible states. This is the situation e.g. in the case of the EU CTL operator.

Algorithm 1: ConsSaturate

```
     input   : $c, s$ : node
     // $c$: constraint,
     // $s$: node to be saturated
     output : node
   1 $l \leftarrow s.level$; $r \leftarrow \mathcal{N}_l^{-1}$;
   2 $t \leftarrow NewNode(l)$;
   3 foreach $i \in \mathcal{S}_l : s[i] \neq \mathbf{0}$ do
 * 4     if $c[i] \neq \mathbf{0}$ then
   5         $t[i] \leftarrow ConsSaturate(c[i], s[i])$;
   6     else
 * 7         $t[i] \leftarrow s[i]$;
   8 repeat
   9     foreach $i, i' \in \mathcal{S}_l : r[i][i'] \neq \mathbf{0}$ do
 *10         if $c[i'] \neq \mathbf{0}$ then
  11             $u \leftarrow RelProd(c[i'], t[i], r[i][i'])$;
  12             $t[i'] \leftarrow Union(t[i'], u)$;
  13 until $t$ unchanged;
  14 $t \leftarrow CheckIn(l, t)$;
  15 return $t$;
```

Algorithm 2: RelProd

```
     input   : $c, s, r$ : node
     // $c$: constraint,
     // $s$: node to be saturated,
     // $r$: next state function
     output : node
   1 if $s = \mathbf{1} \wedge r = \mathbf{1}$ then return $\mathbf{1}$;
   2 ;
   3 $l \leftarrow s.level$; $t \leftarrow \mathbf{0}$;
   4 foreach $i, i' \in \mathcal{S}_l : r[i][i'] \neq \mathbf{0}$ do
 * 5     if $c[i'] \neq \mathbf{0}$ then
   6         $u \leftarrow RelProd(c[i'], t[i], r[i][i'])$;
   7         if $u \neq \mathbf{0}$ then
   8             if $t = \mathbf{0}$ then
                     $t \leftarrow NewNode(l)$;
   9             ;
  10             $t[i'] \leftarrow Union(t[i'], u)$;
  11 $t \leftarrow CheckIn(l, t)$;
  12 $t \leftarrow ConsSaturate(c, t)$;
  13 return $t$;
```

4 Efficient Saturation-based Bounded Model Checking

In this section we present our new, saturation-based bounded model checking algorithm. In order to have an efficient model checking procedure that produces the model checking result from the specification and the formal model, the following ingredients are needed:

– an efficient state space exploration method,
– an efficient model checking algorithm,
– a powerful search strategy,
– a mechanism to decide on the specification.

We use bounded saturation to efficiently explore the bounded state space and produce a symbolic representation [8]. In this section we introduce a new approach for model checking: we employ constrained saturation-based model checking to provide full CTL model checking on this state space. The motivation of the new approach is that this way we can constrain the CTL model checking algorithm to traverse only the bounded state space which is not the situation for traditional CTL model checking algorithms (for example presented in [8]).

4.1 Constrained Saturation using the Bounded State Space

Many model checking tools limit the specification syntax to a subset of the CTL temporal language, in order to simplify the analysis task and boost performance.

We want to support the full CTL semantics in model checking, and thus we must use backward traversal. This is our main reason for choosing the traditional, fixed-point–based algorithms; as the semantics of forward and backward CTL model checking are different (and incomparable) [5].

The naive approach to combine bounded exploration and structural model checking would be to apply the fixed point computations from the bounded state space on the complete lattice. However, the efficiency of this naive approach would converge to traditional fixed point computations. It could be improved by constructing the intersection of the result from the fixed point iterations with the bounded state space representation, practically restricting each iteration of the fixed point computation to the bounded subspace. All the same, the improvement still suffers from poor performance due to the extensive use of the costly intersection operation.

Our aim is to utilize the saturation approach also during model checking, and to exploit the constrained saturation iteration strategy to provide an efficient bounded model checking algorithm. The main idea is that the symbolically encoded explored bounded state space can serve as the constraint in the constrained saturation algorithm. This way we can expeditiously bound the least fixed point computations. Below we define how the constrained saturation decides on the following CTL operators (where **lfp** denotes the least fixed-point, and *bss* denotes the bounded state space as stored by the MDD):

- **EF**: $M, s \models_k \mathsf{EF}p$ iff $s_0 \subseteq \mathbf{lfp}\ Z[(p \wedge bss) \vee (bss \wedge \mathsf{EX}\, Z)] = \mathit{ConsSaturation}(bss, p \cap bss)$. This way we can directly exploit the constrained saturation algorithm to produce the least fixed point in the given bounded state space *bss*. The result can be utilised by other, both least and greatest fixed point operators.
- **EU**: $M, s \models_k \mathsf{E}[p\mathsf{U}q]$ iff $s_0 \subseteq \mathbf{lfp}\ Z[(q \wedge bss) \vee (bss \wedge p \wedge \mathsf{EX}\, Z)] = \mathit{ConsSaturation}(bss \cap q, bss \cap p)$. This is similar to using the constrained saturation algorithm in traditional saturation-based model checking [12], but within a bounded setting. This result can also be nested into both least and greatest fixed point operators.

As greatest fixed point computations (EG) and simple next state operators (EX) does not require such restrictions in the exploration, we apply traditional fixed point algorithms for them. Although operator EF is just a special case of operator EU, for performance reasons it is worth to be implemented separately.

4.2 Search Strategies

The choice of the search strategy followed during bounded model checking has a significant impact on performance. In this section we evaluate the possible search strategy alternatives. With regard to bounded state space generation, we can have two approaches:

- Given a fixed bound b, we explore the b bounded state space and evaluate the specification on it. We call it the *fixed bound strategy.*

– Given an initial bound *init* and increment value *inc*, we start exploring the state space to the given bound *init*. The model checking algorithm then decides whether it can stop, or it has to increase the bound by *inc*. The procedure stops when it runs out of resources, or the model checking question is answered. We call it the *incremental strategy.*

Traditional bounded model checking uses the increasing depth incremental strategy, typically looking one step further in the state space in a breadth-first manner. Applying this strategy in saturation would lead to lose the efficiency of the special iteration order of saturation. Our experience shows that it is better to let saturation increase the depth by at least 5–10 steps. Finding a good trade-off in choosing the iteration depth is important. A one-step iteration results in the loss of efficiency during saturation. On the other hand, a too large increase of iteration depth results in the loss of efficiency during bounded model checking. We have developed two different incremental search strategies:

– The *restarting strategy* starts again the iteration from the initial state after each iteration, and uses the increased bound in the exploration.
– The *continuing strategy* reuses the formerly explored bounded state space as the set of initial states in the next iteration, and extends it using the bounded saturation algorithm to represent the state space of the increased bound.

The restarting strategy was straightforward to implement, since it simply uses the bounded saturation algorithm. For the continuing strategy we had to modify the bottom-up building strategy of the saturation algorithm. For this purpose, we needed to extend the algorithm to be able to handle even huge initial state sets. This extension contained the modification of the *truncating operations*, the *caching mechanisms* in order to preserve correctness, and the *construction of the decision diagram representation* to be able to handle huge initial set of states. The continuing strategy uses the formerly built data structures which can be more efficient than building every data structure from scratch at each iteration.

4.3 Decision Mechanism

It is also important to be able to decide if the specification is satisfied. Bounded model checking is a semi-decision procedure, therefore it can be used to ensure the following behavioural properties of the specification:

– *Invariant and safety*: proving these properties needs the full state space to be explored, or bounded model checking can give a short counterexample (witness), if it exists.
– *Liveness*: bounded model checking can find a short witness to these properties, or the full state space has to be explored to refute them.
– Other properties, such as combination of safety and liveness properties: 3-valued logic can be used for decision.

Invariant and safety properties are usually proved (in symbolic model checking) by finding inductive invariants without exploring the full state space. This approach cannot be used directly for liveness properties.

Finding Inductive Proof against Liveness Properties. EDD-based state space representation helps us to tell more about liveness properties. Refuting liveness properties may come from the fact that: (1) the algorithm has to explore more from the state space to find a witness, (2) the liveness property does not hold, and there exists a counterexample in the bounded state space. Our approach can handle these differences. This is in contrast to traditional bounded model checking approaches, since they have to encode the difference of the two cases into the SAT formula directly, which is inefficient.

If a liveness property $\mathsf{EG}\ p$ does not hold in the bounded state space *bss*, we can decide whether to investigate the state space further, or to conclude that it will never hold. Let $p_{d=bound}$ be the set of states, where p is true and their distance from the initial state is $d = bound$. $p_{d=bound}$ is encoded in the EDD, we need to traverse the EDD once to get this state set. It can be computed efficiently from the symbolic encoding. Let $result = \mathbf{lfp}\ Z[p_{d=bound} \vee (p \wedge \mathsf{EX}\, Z)] = ConsSaturate(p, p_{d=bound})$, then $s_0 \wedge result = false \Rightarrow \mathsf{EG}\, p = false$ holds.

4.4 Summary of Our Contributions

In this section we described the first efficient saturation-based bounded model checking algorithm, which combines the efficiency of constrained saturation and bounded state space exploration. It has the following properties:

- $\forall f(Z)$: $\mathbf{fp}\ f(Z) \subseteq bss$, for all fixed point the bounded saturation algorithm is bounded by the state space, even for the least fixed point computations.
- It is efficient from the model checking point of view as the algorithm traverses the bounded state space with the saturation iteration strategy.
- With the creative use of constrained saturation it avoids to examine states outside of the discovered bounded state space in the model checking phase.
- It avoids expensive intersection operators during the state traversal of least fixed point operators.

5 Evaluation

We have performed measurements in order to confirm that the presented novel constrained saturation-based bounded model checking algorithm performs better than former approaches. This section summarizes our measurement results.

Our aim was to examine the efficiency of our new algorithm and compare it to a classical saturation-based structural model checking algorithm. We have also examined how saturation-based bounded state space traversal can make CTL-based model checking more scalable. For this purpose we have developed an experimental implementation of our algorithm using the C# programming language. We have also implemented the algorithm taken from [12] as the reference for comparison, which we denoted in the measurements as "Unbounded". For the measurements we used a desktop PC (Intel Q8400 2.66 GHz CPU, 4 GB memory with Windows 7 x64 and .NET 4.0 framework).

The models we used for the evaluation are widely known in the model checking community. We took the models of *Tower of Hanoi* from [10]. The state space of the Tower of Hanoi models scales from 531 441 up to $3,5 \cdot 10^9$ states. The saturation algorithm does not perform well for this model, as it does not correspond to an asynchronous system. These measurements demonstrate that our bounded model checking algorithm can analyse even those models, which are not well suited for saturation. The *Slotted Ring* (SR) is the model of a communication protocol [1], [9]. The size of the state space of the SR–100 model is about 10^{100} states. The *Flexible Manufacturing System* (FMS–N) is a model of production systems [1]. The parameter N refers to the complexity of the model checking problem. For $N = 20$ the state space of the FMS model has 10^{20} states.

Both the initial bound and the increment distance are changeable parameters, thus our algorithm can be fine tuned by the user. If the properties to prove are expected to be "shallow", then the algorithm can be set to work optimally for smaller distances. On the other hand, when the properties to prove are "deeper", then both the initial bound and the increment distance can be set bigger to find a proof in fewer iterations. A priori knowledge about the expected behaviour of the properties can significantly reduce the computational time.

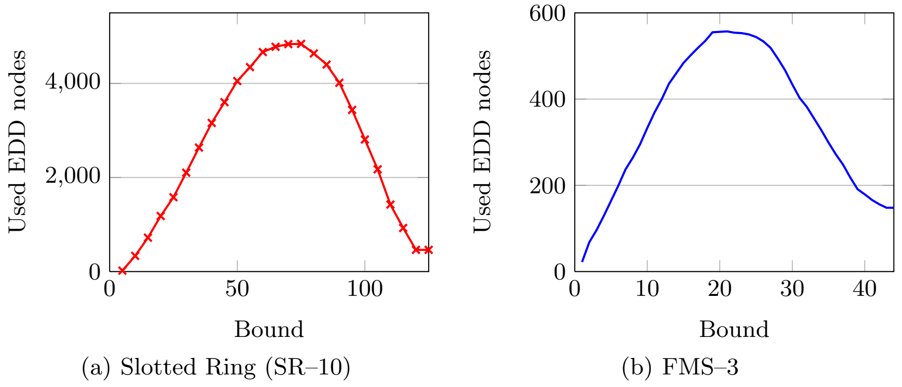


Fig. 2. Size of state space representation (EDD) at each iteration

Table 1 lists our run time measurements for simple reachability properties of the structural model checking (Unbounded), and our bounded model checking approach (Bounded, incremental, restarting strategy). Saturation-based model checking is extremely efficient for asynchronous systems, and the modified iteration strategy requires more computational resources, so one would expect that for these models the traditional approach is better. In the case of Slotted Ring (SR–N, where N is the number of components) models, the analysed property was the following: $\mathsf{E}(B_1 \neq 1 \vee F_1 \neq 1 \;\mathsf{U}\; G_2 = 1 \wedge A_2 = 1)$. The advantage of

Table 1. Comparing run times of model checking for different asynchronous models

Model	**Unbounded**	**Bounded, incremental, restarting strategy**
SR–100	> 1800 s	15.99 s
SR–200	> 1800 s	38.12 s
SR–300	> 1800 s	49.82 s
RR–100	0.24 s	0.27 s
RR–200	0.47 s	0.05 s
RR–1000	2.61 s	0.28 s
RR–10 000	32.54 s	3.39 s
DPhil–10	0.05 s	0.04 s
DPhil–100	0.40 s	0.53 s
DPhil–1000	5.26 s	5.14 s
DPhil–3000	16.19 s	19.52 s
DPhil–10 000	79.64 s	323.26 s

Table 2. Tower of Hanoi model checking run time results

Model	**Unbounded**	**Bounded, incremental, restarting strategy**	**Bounded, incremental, continuing strategy**	**Bounded, fixed bound**
Hanoi–12	39.2 s	6.45 s	2.15 s	1.62 s
Hanoi–14	> 1800 s	6.85 s	2.38 s	1.76 s
Hanoi–16	> 1800 s	10.09 s	2.72 s	1.92 s
Hanoi–18	> 1800 s	10.80 s	3.09 s	2.04 s
Hanoi–20	> 1800 s	11.26 s	3.12 s	2.64 s

Table 3. Comparing strategies for complex properties

Model	**Unbounded**	**Bounded, incremental, restarting strategy**	**Bounded, incremental, continuing strategy**	**Bounded, fixed bound**
FMS–25	1.70 s	1.01 s	1.14 s	0.39 s
FMS–50	9.58 s	2.37 s	3.00 s	1.03 s
FMS–100	82.39 s	4.88 s	6.55 s	1.93 s
FMS–1000	> 1800 s	5.58 s	6.49 s	1.93 s
FMS–10 000	> 1800 s	5.60 s	7.16 s	1.91 s
FMS–1 000 000	> 1800 s	5.68 s	7.11 s	1.95 s

bounded model checking is revealed by the model, as traditional model checking runs out of resources even for such a simple property.

We have also examined Round-Robin models (RR–N, where N is the number of components), which are quite efficiently handled by the traditional saturation based model checking approach. We chose the following property to be checked: $\mathsf{E}(pload_1 = 0 \mathsf{U}\ psend_0 = 1)$. This property is shallow, so the advantage of our bounded model checking approach is well reflected in the results.

The model of the Dining Philosophers (DPhil–N, where N is the number of philosophers) revealed that for those models, where the saturation algorithm answers the model checking question (in this case: $\mathsf{E}(\neg eating_2\ \mathsf{U}\ eating_1)$) extremely fast, bounded model checking is slower. The reason for this is that the overhead of bounded model checking simply does not pay off.

In Table 2 and Table 3 we compare the different approaches for complex properties. Table 2 contains the measurements of the Tower of Hanoi models. We have examined a combined safety-liveness property ($\mathsf{EG}(\mathsf{EF}(B_{\downarrow 8} > 0))$, where $B_{\downarrow 8} > 0$ denotes the placement of the 8th disk to the 2nd rod). The traditional structural model checking approach (Unbounded) runs out of resources early. Knowing the exact bound can help the algorithm to answer the model checking question as fast as possible (Bounded, fixed bound). Comparing the two different bounded model checking strategies, the continuing strategy has advantage as it uses up the formerly computed results during the model checking.

In Table 3 the run time results for the property $\mathsf{EG}(\mathsf{E}(M1 > 0\ \mathsf{U}\ (P1s = P2s = P3s = 8)))$ of the model FMS are depicted. This property is also a combined safety-liveness property that represents the existence of a circle in a certain set of states satisfying some safety requirements (based on [2]). The structural model checking algorithm time-outs for big parameters. By setting an adequate bound, the bounded model checking approach answers the model checking question very fast (Bounded, fixed bound). When we compare the two bounded model checking strategies, the result is surprising: the restarting strategy solves the model checking problem for every parameter *faster* than the continuing strategy. We investigated the reason for this. It can be seen in Figure 2 that for asynchronous systems (like FMS) the state space representation grows steeply up to a given value, but after that it starts decreasing (resembling a bell curve). The continuing strategy uses these intermediate state space representations as the initial state, which is a large computational overhead compared to starting the iteration from the initial state. By beginning model checking from scratch (i.e., using the restarting strategy) we can exploit the efficiency of saturation for building the state space representation. By starting to modify an intermediate representation (i.e., using the continuing strategy) the algorithm has to do more computations, especially if the intermediate representation is larger than the final one.

6 Conclusion and future work

We have presented in this paper an advanced bounded model checking approach based on the saturation algorithm. Our work exploits the efficiency of saturation and enables us to verify complex, or even infinite-state models. Our approach also extends the set of asynchronous systems that can be analysed with the help of symbolic methods. We have proved the efficiency of the new approach with measurements.

We intend to develop our solution further. We will investigate the use of forward model checking [6] instead of the classical backward fixed point computation, as we believe this can further improve the performance of our algorithm. We also plan to use the constrained saturation algorithm in a different way, in order to avoid redundant computations more efficiently.

References

1. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound. In: TACAS 2003. pp. 379–393. Springer (2003)
2. Ciardo, G., Siminiceanu, R.: Structural symbolic CTL model checking of asynchronous systems. In: Computer Aided Verification (CAV'03), LNCS 2725. pp. 40–53. Springer-Verlag (2003)
3. Ciardo, G., Yu, A.: Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. Correct Hardware Design and Verification Methods 3725, 146–161 (2005)
4. Clarke, E., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
5. Henzinger, T., Kupferman, O., Qadeer, S.: From pre-historic to post-modern symbolic model checking. In: Computer Aided Verification. pp. 195–206 (1998)
6. Iwashita, H., Nakata, T.: Forward model checking techniques oriented to buggy designs. ICCAD-97 pp. 400–404 (1997)
7. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
8. Penjam, J. (ed.): Proc. of the 12th Symposium on Programming Languages and Software Tools, SPLST'11. Tallinn, Estonia (2011)
9. Vörös, A., Bartha, T., Darvas, D., Szabó, T., Jámbor, A., Horváth, Á.: Parallel saturation based model checking. In: ISPDC11. IEEE Computer Society (2011)
10. Vörös, A., Darvas, D., Bartha, T.: Bounded Saturation Based CTL Model Checking. In: Penjam [8], pp. 149–160
11. Yu, A., Ciardo, G., Lüttgen, G.: Decision-diagram-based techniques for bounded reachability checking of asynchronous systems. Int. J. Softw. Tools Technol. Transf. 11, 117–131 (2009)
12. Zhao, Y., Ciardo, G.: Symbolic CTL model checking of asynchronous systems using constrained saturation. pp. 368–381. ATVA '09, Springer-Verlag, Berlin, Heidelberg

Acknowledgement

This work was partially supported by the ARTEMIS JU and the Hungarian National Development Agency (NFÜ) in framework of the R3-COP project. The authors would like to thank Prof. Gianfranco Ciardo for his valuable advice and suggestions.

Extensions to the CEGAR Approach on Petri Nets[⋆]

Ákos Hajdu[1], András Vörös[1], Tamás Bartha[2], and Zoltán Mártonka[1]

[1] Dept. of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
`vori@mit.bme.hu`
[2] Computer and Automation Research Institute
MTA SZTAKI,
Budapest, Hungary

Abstract. Formal verification is becoming more prevalent and often compulsory in the safety-critical system and software development processes. Reachability analysis can provide information about safety and invariant properties of the developed system. However, checking the reachability is a computationally hard problem, especially in the case of asynchronous or infinite state systems. Petri nets are widely used for the modeling and verification of such systems. In this paper we examine a recently published approach for the reachability checking of Petri net markings. We give proofs concerning the completeness and the correctness properties of the algorithm, and we introduce algorithmic improvements. We also extend the algorithm to handle new classes of problems: submarking coverability and reachability of Petri nets with inhibitor arcs.

1 Introduction

The development of complex, distributed systems, and safety-critical systems in particular, require mathematically precise verification techniques in order to prove the suitability and faultlessness of the design. Formal modeling and analysis methods provide such tools. However, one of the major drawbacks of formal methods is their computation and memory-intensive nature: even for relatively simple distributed, asynchronous systems the state space and the set of possible behaviors can become unmanageably large and complex, or even infinite.

This problem also appears in one of the most popular modeling formalisms, Petri nets. Petri nets have a simple structure, which makes it possible to use strong structural analysis techniques based on the so-called *state equation.* As structural analysis is independent of the initial state, it can handle even infinite state problems. Unfortunately, its pertinence to practical problems, such

[⋆] This work was partially supported by the European Union and the European Social Fund through the project FuturICT.hu (grant no. TAMOP-4.2.2.C-11/1/KONV-2012-0013) of VIKING Zrt Balatonfured.

as reachability analysis, has been limited. Recently, a new algorithm [12] using Counter-Example Guided Abstraction Refinement (CEGAR) extended the applicability of state equation based reachability analysis.

Our paper improves this new algorithm in several important ways. The authors of the original CEGAR algorithm have not published proofs for the completeness of their algorithm and the correctness of a heuristic used in the algorithm. In this paper we analyze the correctness and completeness of their work as well as our extensions. We prove the lack of correctness in certain situations by a counterexample, and provide corrections to overcome this problem. We also prove that the algorithm is incomplete, due to its iteration strategy. We describe algorithmic improvements that extend the set of decidable problems, and that effectively reduce the search space. We extend the applicability of the approach even further: we provide solutions to handle Petri nets with inhibitor arcs, and the so-called *submarking coverability problem.* At the end of our paper we demonstrate the efficiency of our improvements by measurements.

2 Background

In this section we introduce the background of our work. First, we present Petri nets (Section 2.1) as the modeling formalism used in our work. Section 2.2 introduces the counterexample guided abstraction refinement method and its application for the Petri net reachability problem.

2.1 Petri nets

Petri nets are graphical models for concurrent and asynchronous systems, providing both structural and dynamical analysis. A discrete ordinary Petri net is a tuple $PN = (P, T, E, W)$, where P is the set of *places*, T is the set of *transitions*, with $P \neq T \neq \emptyset$ and $P \cap T = \emptyset$, $E \subseteq (P \times T) \cup (T \times P)$ is the set of *arcs* and $W : E \rightarrow \mathbb{Z}^+$ is the weight function assigning weights $w^-(p_j, t_i)$ to the edge $(p_j, t_i) \in E$ and $w^+(p_j, t_i)$ to the edge $(t_i, p_j) \in E$ [9].

A *marking* of a Petri net is a mapping $m : P \rightarrow \mathbb{N}$. A place p contains k tokens in a marking m if $m(p) = k$. The initial marking is denoted by m_0.

Dynamic behavior. A transition $t_i \in T$ is *enabled* in a marking m, if $m(p_j) \geq w^-(p_j, t_i)$ holds for each $p_j \in P$ with $(p_j, t_i) \in E$. An enabled transition t_i can *fire*, consuming $w^-(p_j, t_i)$ tokens from places $p_j \in P$ if $(p_j, t_i) \in E$ and producing $w^+(p_j, t_i)$ tokens on places $p_j \in P$ if $(t_i, p_j) \in E$. The firing of a transition t_i in a marking m is denoted by $m[t_i\rangle m'$ where m' is the marking after firing t_i.

A word $\sigma \in T^*$ is a *firing sequence*. A firing sequence is *realizable* in a marking m and leads to m', $m[\sigma\rangle m'$, if either $m = m'$ and σ is an empty word, or there exists a $w \in T^*$ realizable firing sequence, a $t_i \in T$, and an m'' such that $m[w\rangle m''[t_i\rangle m'$. The *Parikh image* of a firing sequence σ is a vector $\wp(\sigma) : T \rightarrow \mathbb{N}$, where $\wp(\sigma)(t_i)$ is the number of the occurrences of t_i in σ.

Petri nets can be extended with *inhibitor arcs* to become a tuple $PN_I = (PN, I)$, where $I \subseteq (P \times T)$ is the set of inhibitor arcs. There is an extra condition for a transition $t_i \in T$ with inhibitor arcs to be enabled: for each $p_j \in P$, if $(p_j, t_i) \in I$, then $m(p_j) = 0$ must hold. A Petri net extended with inhibitor arcs is *Turing complete*.

Reachability problem. A marking m' is *reachable* from m if there exists a realizable firing sequence $\sigma \in T^*$, for which $m[\sigma\rangle m'$ holds. The set of all reachable markings from the initial marking m_0 of a Petri net PN is denoted by $R(PN, m_0)$. The aim of the *reachability problem* is to check if $m' \in R(PN, m_0)$ holds for a given marking m'.

We define a *predicate* as a linear inequality on markings of the form $Am \geq b$, where A is a matrix and b is a vector of coefficients [6]. The aim of the *submarking coverability problem* is to find a reachable marking $m' \in R(PN, m_0)$ for which a given predicate $Am' \geq b$ holds.

The reachability problem is decidable [8], but it is at least EXPSPACE-hard [7]. Using inhibitor arcs, the reachability problem in general is undecidable [3].

State equation. The *incidence matrix* of a Petri net is a matrix $C_{|P| \times |T|}$, where $C(i, j) = w^+(p_i, t_j) - w^-(p_i, t_j)$. Let m and m' be markings of the Petri net, then the *state equation* takes the form $m + Cx = m'$. Any vector $x \in \mathbb{N}^{|T|}$ fulfilling the state equation is called a *solution*. Note that for any realizable firing sequence σ leading from m to m', the Parikh image of the firing sequence fulfills the equation $m + C\wp(\sigma) = m'$. On the other hand, not all solutions of the state equation are Parikh images of a realizable firing sequence. Therefore, the existence of a solution for the state equation is a necessary but not sufficient criterion for the reachability. A solution x is called *realizable* if there exists a realizable firing sequence σ, with $\wp(\sigma) = x$.

T-invariants. A vector $x \in \mathbb{N}^{|T|}$ is called a *T-invariant* if $Cx = 0$ holds. A realizable T-invariant represents the possibility of a cyclic behavior in the modeled system, since its complete occurrence does not change the marking. However, during firing the transitions of the T-invariant, some intermediate markings can be interesting for us later.

Solution space. Each solution x of the state equation $m + Cx = m'$, can be written as the sum of a *base vector* and the linear combination of T-invariants [12], which can formally be written as $x = b + \sum_i n_i y_i$, where b is the base vector and n_i is the coefficient of the T-invariant y_i.

2.2 The CEGAR approach

The counterexample guided abstraction refinement (CEGAR) is a general approach for analyzing systems with large or infinite state space. The CEGAR method works on an abstraction of the original model, which has fewer restrictions. During the iteration steps, the CEGAR method refines the abstraction using the information from the explored part of the state space. When applying CEGAR on the Petri net reachability problem [12], the initial abstraction is the state equation. Solving the state equation is an integer linear programming problem [5], for which the ILP solver tool can yield one solution, minimizing a target function of the variables. Since the algorithm seeks the shortest firing sequences leading to the target marking, it minimizes the function $f(x) = \sum_{t \in T} x(t)$. When solving the ILP problem, the following situations are possible:

– If the state equation is infeasible, the necessary criterion does not hold, thus the target marking is not reachable.
– If the state equation has a realizable solution, the target marking is reachable.
– If the state equation has an unrealizable solution, it is a counterexample and the abstraction has to be refined.

The purpose of the abstraction refinement is to exclude counterexamples from the solution space, without losing any realizable solution. For this purpose, the CEGAR approach uses linear inequalities over transitions, called *constraints.*

Constraints. Two types of constraints were defined by Wimmel and Wolf [12]:

– *Jump constraints* have the form $|t_i| < n$, where $n \in \mathbb{N}$, $t_i \in T$ and $|t_i|$ represents the firing count of the transition t_i. Jump constraints can be used to switch between base vectors, exploiting their pairwise incomparability.
– *Increment constraints* have the form $\sum_{i=1}^{k} n_i |t_i| \geq n$, where $n_i \in \mathbb{Z}$, $n \in \mathbb{N}$, and $t_i \in T$. Increment constraints can be used to reach non-base solutions.

Partial solutions. For a given Petri net $PN = (P, T, E, W)$ and a reachability problem $m' \in R(PN, m_0)$, a *partial solution* is a tuple $(\mathcal{C}, x, \sigma, r)$, where:

– $\mathcal{C}$ is the set of jump and increment constraints, together with the state equation they define the ILP problem
– x is the minimal solution satisfying the state equation and the constraints in $\mathcal{C}$,
– $\sigma \in T^*$ is a maximal realizable firing sequence, with $\wp(\sigma) \leq x$, i.e., each transition can fire as many times as it is included in the solution vector x,
– $r = x - \wp(\sigma)$ is the remainder vector.

Generating partial solutions. Partial solutions can be produced from a solution vector x (and a constraint set $\mathcal{C}$) by firing as many transitions as possible. For this purpose, the algorithm uses a "brute force" method. The algorithm builds a tree with markings as nodes and occurrences of transitions as edges. The root of the tree is the initial marking m_0, and there is an edge labeled by t between nodes m_1 and m_2 if $m_1[t\rangle m_2$ holds. On each path leading from the root of the tree to a leaf, each transition t_i can occur at most $x(t_i)$ times. Each path to a leaf represents a maximal firing sequence, thus a new partial solution. Even though the tree can be traversed only storing one path in the memory at a time using depth first search, the size of the tree can grow exponentially. Some optimizations are presented later in this section to reduce the size of the tree.

A partial solution is called a *full solution* if $r = 0$ holds, thus, $\wp(\sigma) = x$, which means that σ realizes the solution vector x. For each realizable solution x of the solution space there exists a full solution [12]. This full solution can be reached by continuously expanding the minimal solution of the state equation with constraints.

Consider now a partial solution $ps = (\mathcal{C}, x, \sigma, r)$ which is not a full solution, i.e., $r \neq 0$. This means that some transitions could not fire enough times. There are three possible situations in this case:

1. x may be realizable by another firing sequence σ', thus a full solution $ps' = (\mathcal{C}, x, \sigma', r)$ exists.
2. By adding jump constraints, greater, but pairwise incomparable solutions can be obtained.
3. For transitions $t \in T$ with $r(t) > 0$ increment constraints can be added to increase the token count on the input places of t, while the final marking m' must be unchanged. This can be achieved by adding new T-invariants to the solution. These T-invariants can "borrow" tokens for transitions in the remainder vector.

Generating jump constraints. Each base vector of the solution space can be reached by continuously adding jump constraints to the minimal solution [12]. In order to reach non-base solutions, increment constraints are needed, but they might conflict with previous jump constraints. Jump constraints are only needed to obtain a different base solution vector. However, after the computation of the base solution, jump constraints can be transformed into equivalent increment constraints ([12]).

Generating increment constraints. Let $ps = (\mathcal{C}, x, \sigma, r)$ be a partial solution with $r > 0$. This means that some transitions (in r) could not fire enough times. The algorithm uses a heuristic to find the places and number of tokens needed to enable these transitions. If a set of places actually needs n ($n > 0$) tokens, the heuristic estimates a number from 1 to n. If the estimate is too low, this method can be applied again, converging to the actual number of required tokens. The heuristic consists of the following three steps:

1. First, the algorithm builds a dependency graph [10] to get the transitions and places that are of interest. These are transitions that could not fire, and places which disable these transitions. Each source SCC[3] of the dependency graph has to be investigated, because it cannot get tokens from another components. Therefore, an increment constraint is needed.

[3] Strongly connected component

2. The second step is to calculate the minimal number of missing tokens for each source SCC. There are two sets of transitions, $T_i \subseteq T$ and $X_i \subseteq T$. If one transition in T_i becomes fireable, it may enable all the other transitions of the SCC, while transitions in X_i cannot activate each other, therefore their token shortage must be fulfilled at once.
3. The third step is to construct an increment constraint c for each source SCC from the information about the places and their token requirements. These constraints will force transitions (with $r(t) = 0$) to produce tokens in the given places. Since the final marking is left unchanged, a T-invariant is added to the solution vector.

When applying the new constraint c, three situations are possible depending on the T-invariants in the Petri net:

– If the state equation and the set of constraints become infeasible, this partial solution cannot be extended to a full solution, therefore it can be skipped.
– If the ILP solver can produce a solution $x + y$ (with y being a T-invariant), new partial solutions can be found. If none of them help getting closer to the full solution, the algorithm can get into an infinite loop, but no full solution is lost. A method to avoid this non-termination phenomenon will be discussed below.
– If there is a new partial solution ps' where some transitions in the remainder vector could fire, this method can be continued.

Theorem 1. *(Reachability of solutions) [12] If the reachability problem has a solution, a realizable solution of the state equation can be reached by continuously adding constraints, transforming jumps before increments.*

Optimizations. Wimmel and Wolf [12] presented also some methods for optimization. The following are important for our work:

– **Stubborn set** The stubborn set method [10] investigates conflicts, concurrency and dependencies between transitions, and reduces the search space by filtering the transitions: stubborn set method usually leads to a search tree with lower degree.
– **Subtree omission** When a transition has to fire more than once ($x(t) > 1$), the stubborn set method does not provide efficient reduction. The same marking is often reached by firing sequences which only differ in the order of transitions. During the abstraction refinement, only the final marking of the firing sequence is important. If a marking m' is reached by firing the same transitions as in a previous path, but in a different order, the subtree after m' was already processed. Therefore, it is no longer of interest.
– **Filtering T-invariants** After adding a T-invariant y to the partial solution $ps = (\mathcal{C}, x, \sigma, r)$, all the transitions of y may fire without enabling any transition in r, yielding a partial solution $ps' = (\mathcal{C}', x + y, \sigma', r)$. The final marking and remainder vector of ps' is the same as in ps, therefore the same T-invariant y is added to the solution vector again, which can prevent the

algorithm from terminating. However, during firing the transitions of y, the algorithm could get closer to enabling a transition in r. These intermediate markings should be detected, and be used as new partial solutions.

3 Theoretical results

In this section we present our theoretical results with regard to the correctness and completeness of the original algorithm.

3.1 Correctness

Although Theorem 1 states that a realizable solution can be reached using constraints, we found out that in some special cases the heuristic used for generating increment constraints can overestimate the required number of tokens for proving reachability. We prove the incorrectness by a counterexample, for which the original algorithm [12] gives an incorrect answer.

Consider the Petri net in Figure 1 with the reachability problem $(0,1,0,0,1,0,0,2) \in R(PN,(1,0,0,0,0,0,0,2))$, i.e., we want to move the token from p_0 to p_1 and p_4. The example was constructed so that the target marking is reachable by the firing sequence $\sigma_m = (t_1,t_2,t_0,t_5,t_6,t_3,t_7,t_4)$, realizing the solution vector $x_m = (1,1,1,1,1,1,1,1)$.

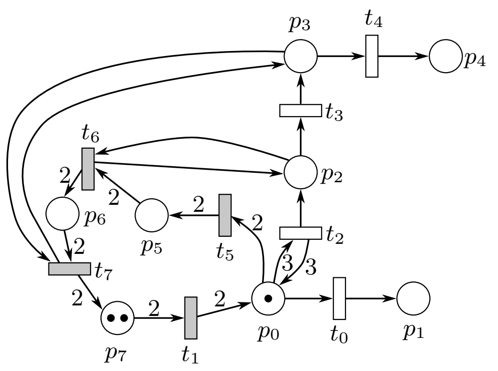


Fig. 1. Counterexample for correctness.

The CEGAR algorithm does the following steps. First, it finds the minimal solution vector $x = (1,0,1,1,1,0,0,0)$, i.e., it tries to fire the transitions t_0,t_2,t_3,t_4. From these transitions only t_0 is enabled, therefore the only partial solution is $ps = (\emptyset, x, \sigma = (t_0), r = (0,0,1,1,1,0,0,0))$. At this point the algorithm looks for an increment constraint. The dependency graph contains transitions t_2,t_3,t_4 (since they could not fire) and places p_0,p_2,p_3 (because they disable the previous transitions). The only source SCC is the set containing one place p_0 with zero tokens (because t_0 has consumed one token from there). The algorithm estimates that three tokens are needed in place p_0, where only transition t_1 can produce tokens. Therefore, the T-invariant t_1, t_5, t_6, t_7 is added twice to the solution vector. This invariant is constructed so that for each of its firing, a token has to be produced in places p_2, p_3, p_4, which token can no longer be removed. In the target marking only one token can be present on these places, therefore the algorithm cannot find the solution for the reachability problem.

Notice that the problem is the over-estimation of tokens required at p_0. Without forcing t_0 to fire, the algorithm could get a better estimation. This would imply that the invariant t_1, t_5, t_6, t_7 is added only once to the solution vector, producing the realizable solution x_m. The problem is that the algorithm always tries to find maximal firing sequences, though some transitions would not be practical to fire (t_0 in the example above). Due to this, the estimated number of tokens needed in the final marking of the firing sequence may not be correct.

Solution. Our improved algorithm counts the maximal number of tokens in each place during the firing sequence of the partial solutions into a vector m_{max}. If the final marking is not the maximal regarding a SCC, the algorithm might have over-estimated the required number of tokens. This can be detected by ordering the intermediate markings. Formally: an over-estimation can occur if a place p exists in a SCC, for which $m_{max}(p) > m'(p)$ holds, where m' is the final marking of the firing sequence.

3.2 Completeness

To our best knowledge, the completeness of the algorithm has neither been proved nor disproved yet. When we examined the iteration strategy of the abstraction loop, we found a whole subclass of nets, which cannot be solved with this strategy. As an example, consider the Petri net in Figure 2 with the reachability problem $(1, 1, 0, 0) \in R(PN, (0, 1, 0, 0))$, i.e., we want to produce a token in p_0. We constructed the net so that the firing sequence $\sigma = (t_1, t_4, t_2, t_3, t_3, t_0, t_1, t_2, t_5)$ solves the problem. The main concept of this example is that we lend an extra token on p_1 indirectly using the T-invariant t_4, t_5.

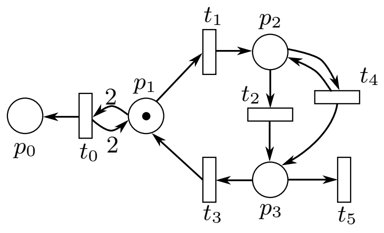


Fig. 2. Counterexample of completeness.

When applying the algorithm on this problem, the minimal solution vector is $x_0 = (1,0,0,0,0,0)$, i.e., firing t_0. Since t_0 is not enabled, the only partial solution is $ps_0 = (\emptyset, x_0, \sigma_0 = (), r_0 = (1,0,0,0,0,0))$. The algorithm finds that an additional token is required in p_1, and only t_3 can satisfy this need. With an increment constraint $c_1 : |t_3| \geq 1$, the T-invariant t_1, t_2, t_3 is added to the new solution vector $x_1 = (1,1,1,1,0,0)$, giving us one partial solution $ps_1 = (c_1, x_1, \sigma_1 = (t_1, t_2, t_3), r_1 = r_0)$. Firing the T-invariant t_1, t_2, t_3 does not help getting closer to enabling t_0, since no extra token can be "borrowed" from the previous T-invariant. The iteration strategy of the original algorithm does not recognize the fact that an extra token could be produced in p_3 (using t_4) and then moved in p_1, therefore it can not decide reachability.

4 Algorithmic contributions

In this section we present our algorithmic contributions. In Section 4.1 we show some classes of problems for which the original algorithm cannot decide reachability, and our improved algorithm solves these problems. In Section 4.2 we present two extensions of the algorithm, solving submarking coverability problems and handling Petri nets with inhibitor arcs.

4.1 Improvements

In the previous section we proved that the algorithm is not complete, but during our work we found some opportunities to extend the set of decidable problems. Moreover, we developed a new termination criterion which we prove to be correct, i.e., no realizable solution is lost using this criterion.

Total ordering of intermediate markings. When a partial solution $ps = (\mathcal{C}, x, \sigma, r)$ is skipped using the T-invariant filtering optimization, the original algorithm checks if it was closer to firing a transition t in the remainder during the firing sequence σ. This is done by "counting the minimal number of missing tokens for firing t in the intermediate markings occurring"[12]. We found out that this criterion is not general enough: in some cases the total number of missing tokens may not be less, but they are missing from different places, where additional tokens can be produced. In our new approach, we use the following definition:

Definition 1. *An intermediate marking m_i is considered better than the final marking m', if there is a transition $t \in T, r(t) > 0$ and place p with $(p,t) \in E$ for which the following criterion holds:*

$$m'(p) < w^-(p,t) \quad \wedge \quad m_i(p) > m'(p). \tag{1}$$

The left inequality in the expression means that in the final marking t is disabled by the insufficient amount of tokens in p. This condition is important, because we do not want to have more tokens on places, that already have enough to enable t. The right inequality means that p has more tokens in the intermediate marking m_i compared to the final marking m'.

Theorem 2. *Definition 1 is a total ordering of the intermediate markings occurring in the firing sequence of a partial solution.*

Proof. We first show that Definition 1 includes the original ordering of the intermediate markings. When the original criterion holds, the total number of missing tokens for enabling t at the marking m_i is less than at m'. This means that at least one place p must exist, which disables t, but $m_i(p) > m'(p)$, therefore (1) must hold. Furthermore, Definition 1 also recognizes markings which are pairwise incomparable, because if there is at least one place p with lesser tokens missing, (1) holds.

Corollary 1. *The total ordering of intermediate markings extends the set of decidable problems.*

Definition 1 is more general than the original criterion, hence it does not reduce the set of decidable problems. On the other hand, we give an example when the original criteria prevents the algorithm from finding the solution. Consider the Petri net in Figure 3 with the reachability problem $(1, 0, 0, 1) \in R(PN, (0, 1, 0, 1))$, i.e., moving one token from p_1 to p_0. The minimal solution vector is $x_0 = (1, 0, 0, 0, 0)$, i.e., firing t_0, which is disabled by p_2, therefore the only partial solution is $ps_0 = (\emptyset, x, \sigma_0 = (), r_0 = (1, 0, 0, 0, 0))$. The algorithm looks for increment constraints and finds that only t_1 can produce tokens on p_2. Consequently, the T-invariant t_1, t_2 is added to the solution vector $x_1 = (1, 1, 1, 0, 0)$. There is one partial solution $ps_1 = (\{|t_1| \geq 1\}, x_1, \sigma_1 = (t_1, t_2), r_1 = (1, 0, 0, 0, 0))$ for x_1, where the T-invariant is fired, but t_0 still could not fire. This partial solution is skipped by the T-invariant filtering optimization, and in all of the intermediate markings of σ_1, totally one token is missing from the input places of t_0. By using the original criterion, the algorithm terminates, leaving the problem as undecided. By using Definition 1 after firing t_1, less tokens are missing from p_2 than in the final marking. Continuing from here, t_0 is disabled by p_1, where t_3 can produce tokens, therefore the T-invariant t_3, t_4 is added to the new solution vector $x_2 = (1, 1, 1, 1, 1)$. A full solution is found for x_2 by the realizable firing sequence $\sigma_2 = (t_1, t_3, t_0, t_2, t_4)$.

T-invariant filtering and subtree omission. Using T-invariant filtering and subtree omission optimizations together can prevent the algorithm from finding full solutions. The order of transitions in the firing sequence of a partial solution does not matter, except in one case. When a partial solution is skipped, the algorithm checks for intermediate markings where it was closer to firing a transition in the remainder vector. By using subtree omission, intermediate markings can get lost.

As an example consider the Petri net in Figure 4 with the reachability problem $(1, 0, 0, 0, 3) \in R(PN, (0, 0, 0, 0, 3))$, i.e., we want to produce a token on p_0.

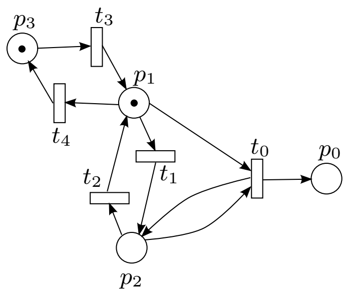


Fig. 3. Example net depicting the usefulness of the total ordering

A possible solution is the vector $x_m = (1, 1, 1, 2, 2, 3, 3)$ realized by the firing sequence $\sigma_m = (t_6, t_6, t_6, t_4, t_4, t_2, t_0, t_1, t_3, t_3, t_5, t_5, t_5)$.

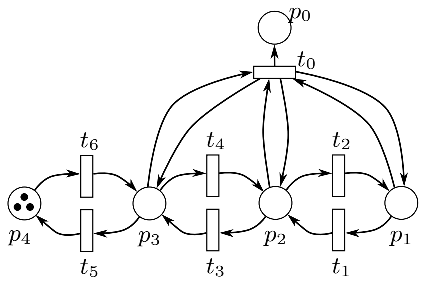


Fig. 4. An example where the order of transitions matter.

Here we present only the interesting points during the execution of the algorithm. As a minimal solution, the algorithm tries to fire t_0, but it is disabled by the places p_1, p_2, p_3. The algorithm searches for increment constraints. All the three places are in different SCCs, so the algorithm first tries to enable t_0 by borrowing one token for all three places. By the T-invariant $t_1, t_2, \ldots, t_6$ a token is carried through places p_1, p_2, p_3, which does not enable t_0, but there are intermediate markings where the enabling of t_0 is closer. Continuing from any of these intermediate markings, another token is borrowed on the places p_1, p_2, p_3, but t_0 is not yet enabled. Here comes the different order of transitions into view:

- If the two tokens are carried through places p_1, p_2, p_3 together, there are intermediate markings that are closer to firing t_0, because previously two tokens were missing, now only one. Continuing from these markings a third token is borrowed on places p_1, p_2, p_3, enabling t_0 and yielding a full solution.
- If the two tokens are carried through places p_1, p_2, p_3 separately (i.e., a token is carried through the places, while the other is left in p_4, and this procedure is repeated), there are no intermediate markings of interest, because two

tokens are still missing to enable t_0. In this case the algorithm will not find the full solution.

The order of transitions is non-deterministic, thus it is unknown which order will be omitted. Therefore, in our approach we reproduce all the possible firing sequences without subtree omission when a partial solution is skipped, and check for intermediate markings in the full tree. Although this may yield a computational overhead in some cases, we might lose full solutions otherwise.

New termination criterion. We have developed a new termination criterion, which can efficiently cut the search space without losing any full solutions. When generating increment constraints for a partial solution *ps*, as a first step the algorithm finds the set of places $P' \subseteq P$ where tokens are needed. Then it estimates the number of tokens required (n). At this point, our new criterion checks if there exists a marking m' for which the following inequalities hold:

$$\begin{aligned} \sum_{p_i \in P'} m'(p_i) &\geq n \\ \forall p_j \in P: \; m'(p_j) &\geq 0. \end{aligned} \tag{2}$$

The first inequality ensures that at least n tokens are present on the places of P' while the others guarantee that the number of tokens on each place is non-negative. These inequalities define a submarking coverability problem. Using the ILP solver, we can check if the modified form of the state equation (which we discuss in Section 4.2) holds for this problem. If the state equation does not hold, it is a proof that no such marking exists where we have the required number of tokens on the places of P'. Thus, *ps* can be omitted without losing full solutions.

This approach can also extend the set of decidable problems compared to the former approach. Consider the Petri net on Figure 5 with the reachability problem $(1, 1, 0) \in R(PN, (1, 0, 0))$, i.e., firing t_0 to produce a token on p_1. The algorithm would add the T-invariant t_1, t_2 again and again to enable t_0. Using T-invariant filtering we cannot decide whether there is no full solution or we lost it. Using our new approach we can prove that no marking exist where two tokens are present on p_0, therefore no full solution exists.

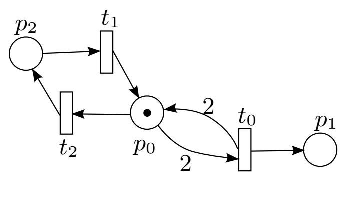


Fig. 5. Example net for the new filtering criterion

4.2 Extensions

We extended the algorithm to handle new types of problems. In this section we present two further extensions: the CEGAR algorithm for solving submarking coverability problems and checking reachability in Petri nets with inhibitor arcs.

Submarking coverability problem. In Section 2 we introduced predicates in the form $Am' \geq b$, where A is a matrix and b is a vector of coefficients. In order to use the state equation, this condition on places must be transformed to a condition on transitions.

At first we substitute m' in the predicate $Am' \geq b$ with the state equation $m_0 + Cx = m'$, which results inequalities of the form $(AC)x \geq b - Am_0$. This set of inequalities can be solved as an ILP problem for transitions. The extended algorithm uses this modified form of the state equation, and expands it with additional (jump or increment) constraints.

Petri nets with inhibitor arcs. The main problem with inhibitor arcs is that they do not appear in any form in the state equation which is used as an abstraction. Therefore, a solution vector produced by the ILP solver may not be realizable because inhibitor arcs disable some transitions. In this case tokens must be removed from some places. Our strategy is to add transitions to the solution vector, that consume tokens from the places connected by inhibitor arcs. Increment constraints are suitable for this purpose, but they have to be generated in a different way:

1. The first step is to construct a dependency graph similar to the original one. The graph consists of transitions that could not fire due to inhibitor arcs and places disabling these transitions. The arcs of the graph have an opposite meaning: an arc from a place to a transition means that the place disables the transition, while the other direction means that firing the transition would decrease the number of tokens on the place. Each source SCC of the graph is interesting, because tokens cannot be consumed from them by another SCC.
2. The second step is to estimate the minimal tokens to be removed from each source SCC. There are two sets of transitions as well, $T_i \subseteq T$ and $X_i \subseteq T$. If one transition in T_i becomes fireable, it may enable all the others in the SCC, while the needs of transitions in X_i must be fulfilled at once.
3. The third step is to construct an increment constraint c for each source SCC from the information of the set of places and the number of tokens to be removed. This yields firing additional transitions (with $r(t) = 0$) to consume tokens from these places.

When a partial solution is not a full solution, and there are transitions disabled by inhibitor arcs, the previous algorithm is used to generate increment constraint. If there are transitions disabled by normal arcs as well, both the original algorithm and the modified version must be used, taking the union of the generated constraints.

Inhibitor arcs also affect some of the optimization methods:

– Stubborn sets currently do not support inhibitor arcs.
– Using T-invariant filtering, an intermediate marking is now of interest when it has less tokens on a place which is connected by inhibitor arc to a transition that cannot fire.
– Our new termination criterion is extended to check whether a reachable marking exists where the required number of tokens are removed.

5 Evaluation

We have implemented our algorithm in the *PetriDotNet* [1] framework. Table 1 contains run-time results, where TO refers to an unacceptable run-time ($>$ 600 seconds). The measured models are published in [4], [11], [12]. In Table 1(a) we have compared our solution to the original algorithm, which is implemented in the *SARA tool* [2] (the numbers in the model names represent the parameters). We have also measured a highly asynchronous consumer-producer model (CP_NR in the table).

Table 1. Measurement results for well-known benchmark problems

(a) Comparison to the original

Model	**SARA**	**Our algorithm**
CP_NR 10	0,2 s	0,5 s
CP_NR 25	111 s	2 s
CP_NR 50	TO	16s
Kanban 1000	0,2 s	1 s
FMS 1500	0,5 s	5 s
MAPK	0,2 s	1 s

(b) Comparison to saturation

Model	**Saturation**	**Our algorithm**
Kanban 1000	TO	1 s
SlottedRing 50	4 s	433 s
DPhil 50	0,5 s	45 s
FMS 1500	TO	5 s

Our implementation is developed in the C# programming language, while the original is in C. This causes a constant speed penalty for our algorithm. Moreover, our algorithm examines more partial solutions, which also yields computational overhead. However, the algorithmic improvements we introduced in this paper significantly reduce the computational effort for certain models (see the consumer-producer model). In addition, our algorithm can in many cases decide a problem that the original one cannot.

We have also compared our algorithm to the well-known saturation-based model checking algorithm [4], implemented in our framework [11]. See the results in Table 1(b). The lesson learned is that if the ILP solver can produce results efficiently (Kanban and FMS models), the CEGAR solution is faster by an order of magnitude than the saturation algorithm. When the size of the model makes the linear programming task difficult, it dominates the run-time, and saturation wins the comparison.

6 Conclusions

The theoretical results presented in this paper are twofold. On one hand, we proved the incompleteness of the iteration strategy of the original CEGAR approach by constructing a counterexample. We also constructed a counterexample that proved the incorrectness of a heuristic used in the original algorithm. We corrected this deficiency by improving the algorithm to detect such situations. On the other hand, our algorithmic improvements reduce the search space, and enable the algorithm to solve the reachability problem for certain, previously unsupported classes of Petri nets. In addition, we extended the algorithm to solve two new classes of problems, namely submarking coverability and handling Petri nets with inhibitor arcs. We demonstrated the efficiency of our improvements with measurements.

References

1. Homepage of the *PetriDotNet* framework., `http://petridotnet.inf.mit.bme.hu/`, [Online; accessed 10-May-2013]
2. Homepage of the *Sara* model checker., `http://service-technology.org/tools/index.html`, [Online; accessed 06-Apr-2013]
3. Chrzastowski-Wachtel, P.: Testing undecidability of the reachability in Petri nets with the help of 10th hilbert problem. In: Donatelli, S., Kleijn, J. (eds.) Application and Theory of Petri Nets 1999, Lecture Notes in Computer Science, vol. 1639, pp. 690–690. Springer (1999)
4. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound. In: Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 379–393. Springer (2003)
5. Dantzig, G.B., Thapa, M.N.: Linear programming 1: introduction. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1997)
6. Esparza, J., Melzer, S., Sifakis, J.: Verification of safety properties using integer programming: Beyond the state equation (1997)
7. Lipton, R.: The Reachability Problem Requires Exponential Space. Research report, Yale University, Dept. of Computer Science (1976)
8. Mayr, E.W.: An algorithm for the general Petri net reachability problem. In: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing. pp. 238–246. STOC '81, ACM, New York, NY, USA (1981)
9. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (April 1989)
10. Valmari, A., Hansen, H.: Can stubborn sets be optimal? In: Lilius, J., Penczek, W. (eds.) Applications and Theory of Petri Nets, Lecture Notes in Computer Science, vol. 6128, pp. 43–62. Springer (2010)
11. Vörös, A., Bartha, T., Darvas, D., Szabó, T., Jámbor, A., Horváth, Á.: Parallel saturation based model checking. In: ISPDC. IEEE Computer Society, Cluj Napoca (2011)
12. Wimmel, H., Wolf, K.: Applying CEGAR to the Petri net state equation. In: Abdulla, P.A., Leino, K.R.M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 17th International Conference, TACAS 2010 Proceedings. Lecture Notes in Computer Science, vol. 6605, pp. 224–238. Springer (2011)