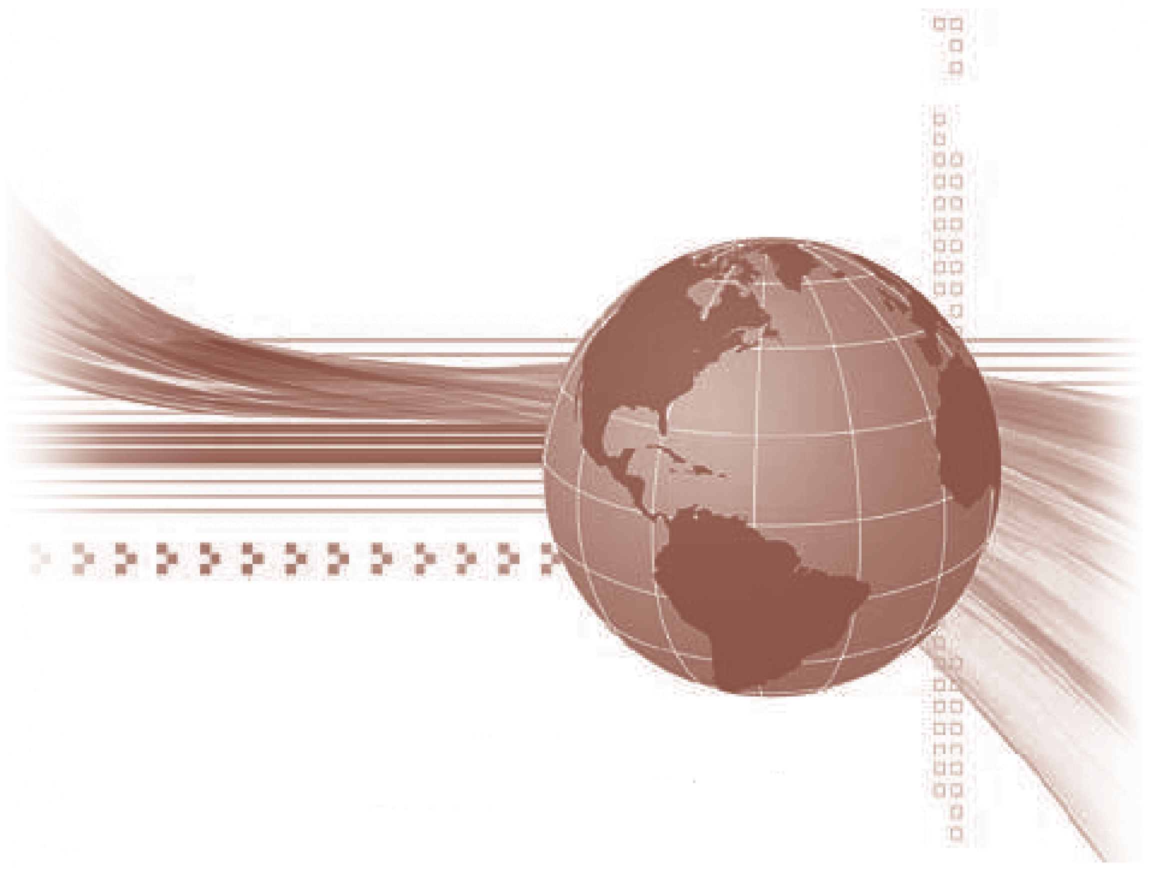
STUDIA UNIVERSITATIS
BABEŞ-BOLYAI



INFORMATICA

1/2018

STUDIA

UNIVERSITATIS BABEŞ-BOLYAI
INFORMATICA

No. 1/2018

January - June

YEAR Volume 63 (LXIII) 2018
MONTH JUNE
ISSUE 1

S T U D I A
UNIVERSITATIS BABEŞ-BOLYAI
INFORMATICA

1

EDITORIAL OFFICE: M. Kogălniceanu 1 • 400084 Cluj-Napoca • Tel: 0264.405300

SUMAR – CONTENTS – SOMMAIRE

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 1, 2018
DOI: 10.24193/subbi.2018.1.01

MEASURING ENERGY EFFICIENCY OF SELECTED WORKING SOFTWARE

CSABA SZABÓ AND EMIRA MUSTAFA MOAMER ALZEYANI

Abstract. Energy consumption is a key performance indicator of any software run on mobile devices. Working or application software is the main category of software where such energy (in)efficient performance becomes accelerated between users and other stakeholders. Measuring energy efficiency is becoming a part of automated and manual performance testing as well – both answering to the increasing usage requirements and addressing acceptance testing optimization. In this paper, we select three software tools – an e-mail client and two social network applications, those energy consumption is being measured and analyzed. We decided to apply very generic profiles during our measurements, where the actions were performed all manually. Our results show that besides the difference in the number of features covered by the software, also their implementation plays an important role in energy consumption. Focusing on a specific feature within the working software does not imply that all quality indicators of it are the best among the software group's implementations.

1. Introduction

Our research focuses on energy consumption of software, which is a relatively new field of software engineering research.

In this paper, we present the results of the application of our measurement setup that relies on the presence of a tester performing usage actions over the system under test. To unify the measurements, i.e. to make them comparable, we decided to select web-based software. In our case, this kind of software also requires a web browser application to run. This application was the same in all cases: Google Chrome.

Received by the editors: April 16, 2018.

2010 *Mathematics Subject Classification.* 68N01, 68M20.

1998 *CR Categories and Descriptors.* D.2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging – *Monitors*; D.2.8 [**SOFTWARE ENGINEERING**]: Metrics – *Product metrics*.

Key words and phrases. energy efficiency, green software, performance testing.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

These three websites were selected, and on the following basis:

iNotes email client: has been chosen because it contains large of texts and the possibility of downloading and uploading it, with the knowledge that it is possible to contain some pictures or videos sent to it.

Facebook: has been selected on the basis that it contains a different set of content such as (text - pictures - videos - some links, etc.) This is very important for our study of the possibility of monitoring changes in energy on the laptop's processor and more.

YouTube: was chosen on the basis of containing the whole on videos and it is known that because of the animation and sound, this needs high energy to run it.

Together with these assumptions, we also considered that an e-mail client will be consuming less energy during work and (almost) no energy in idle state than the other two applications. As the other two applications both present videos and video comments, it was also an aim to compare if YouTube can master video playback better than Facebook.

The structure of our paper is as follows. In the upcoming section, we present work related to the research field. Then, we present our measurements and results. Finally, we conclude and point out future research directions.

2. Related Work

Computer networks, communication systems [5], data centers [9, 16], huge production of mobile phones [4, 7], and other IT infrastructures have caused severe environmental problems by consuming significant amounts of power [15], increasing greenhouse gas emissions, and lead to pollution during the production and disposal.

This growing concern on energy efficiency may also be associated with the perspective of software developers. Unfortunately, developing energy-aware software [3] is still a difficult task. While programming languages provide several compiler optimizations[14], memory profiler tools, benchmark[10, 12] and time execution monitoring frameworks, there are no equivalent tools and frameworks to optimize energy consumption [8].

Mobile software platforms are targeted by a valuable research activity[6, 11, 13], mainly because of the speed of reaction of stakeholders as device battery drain is a much faster indicator than the energy bill integrating power consumption over days or months. The development is therefore also focusing on tool usage and research, which are capable to determine "energy leaks" or evaluate energy efficiency improvements between versions of software[1].

All these research results could be used in a different way as well. We show that way in this paper. Many researchers do also social networks or other web-based application research[2], where the application requires to run in a browser. Their focus on usability and user experience, on requirements on functionality or speed, but the combination of the above two research goals is missing. We are evaluating energy consumption of a browser running the specific applications – as these do not work without the browser, it would be worthless to evaluate energy efficiency without that required environment. Fuel economy of a car is the best if not used, or we have to consider the driver and road conditions as well while driving.

3. Measurement environment setup

Our study is based on measuring energy efficiency of the Google Chrome browser for the Microsoft Windows operating system using the following hardware/software specs:

- Intel Core i5 CPU @ 2.5 GHz
- 4 GB RAM
- Microsoft Windows 7 Home Premium 64-bit operating system with latest updates installed

The study is based on several factors:

- When using a browser for the purpose of browsing a site such as e-mail, this is known to contain a large text.
- When using the browser for the purpose of browsing a site such as Facebook and this site is known to contain many elements, including texts, images and videos.
- When using the browser for the purpose of browsing a site such as YouTube, especially on the video and contain the texts.
- In order to study, we will use an energy measurement program named “JouleMeter” to read and store the readings of energy, compare and analyze them. Hence, the differences will be identified when using the browser in several ways and how it affects the performance of the laptop and the life of the laptop battery. This tool is estimating energy consumption based on CPU, GPU, monitor brightness etc., therefore it needs to be configured before the first measurement. The setup is automatic, but requires some time as the configuration is preforming a performance benchmark on the host device.

We will use Chrome browser to see different results:

1. When we use Chrome to open email page as more as static page.
2. When we use the Chrome browser to open page as Facebook .

(3) When we use the Chrome browser to open page as YouTube.

For a successful measurement, we have to prepare our hardware setup from the operating system as well as to keep in mind technical limits of the Microsoft JouleMeter we used.

(1) We must ensure that the laptop is charged above 75%.
(2) Work on the browser for at least 45 minutes.
(3) Screen brightness was 100% in this study.
(4) Setting of battery: Here we will consider on some setting on battery of the laptop that we worked on it in the study, from the setting of Power Options there are option is Processor power management of battery this is option for these This option will change the speed of the processor clock, and this option gives us the possibility to increase or decrease the ratio according to user requirements. If the rate is changed to less than 100%, energy consumption and heat output will be reduced.

Each measurement is then a simple sequence of steps[1]:

(1) Start the energy consumption monitoring tool
(2) Start the browser or open a new browser window
(3) Enter the application page
(4) Use the application
    - open the newest e-mail, set read/unread 5 emails, click “reply” but close without sending, write an e-mail without attachments, write an e-mail in HTML form, write an e-mail with embedded pictures, write an e-mail with attachments – picture, video, archive file etc.
    - log in to Facebook, set messages read/unread, write a status then delete it, comment, browse statuses/wall, check groups, browse picture galleries, write suggestions, watch posted videos, post something and delete etc.
    - search for artists/publishers on YouTube, watch their activity, write and delete comments, create/modify/delete own watch lists, upload videos, watch videos (turn on/off automatic playback) etc.
(5) Close the browser window
(6) Stop the energy consumption monitoring tool and save collected data

[1]Based on the Intellectual output O1 part/topic “How Green Is Your Process?” of project No. 2017-1-SK01-KA203-035402 (for more details see Acknowledgment).

4. Results

As mentioned above, energy can be estimated after a proper configuration of the JouleMeter tool. The data are stored in a comfortable way that allows direct analysis (Excel or csv file).

We can compare the readings but before that we explain the details on the charts we get:

- The total power chart expresses the value of energy used as a whole.
- The CPU chart expresses the value consumed by the feeder from the total power value as a whole.
- The chart of the program expresses the value of the energy consumed from the CPU value.

4.1. **E-mail client.**

The e-mail client was stressed by actions as stated earlier in this paper. The following three figures, Fig. 1, Fig. 2 and Fig. 3 reflect charts from measurements with sampling of 100 milliseconds.

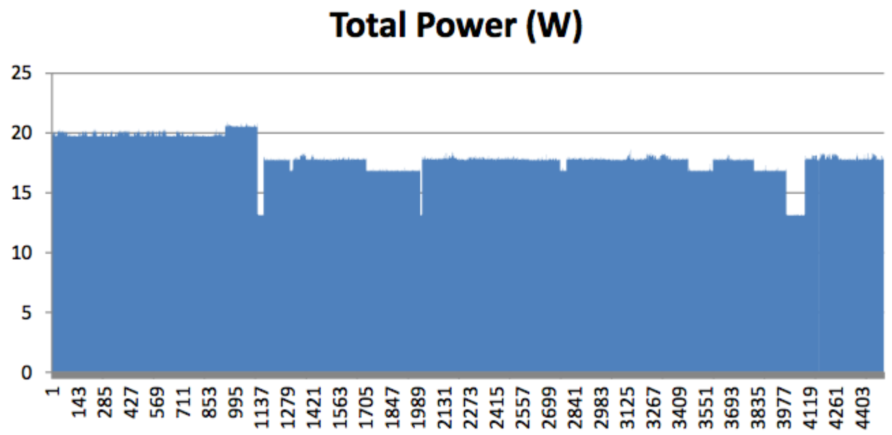


Figure 1. iNotes total power consumption data

Especially Fig. 3 reflects that the application – Chrome, but running the e-mail client only, consumed valuable energy only in the case of user actions. Reading of e-mails is “hidden” as the measurement tool includes the energy consumption of the display as part of the total power consumption chart in Fig. 1.

To finalize our data analysis, Tab. 1 statistically concludes our measurements.

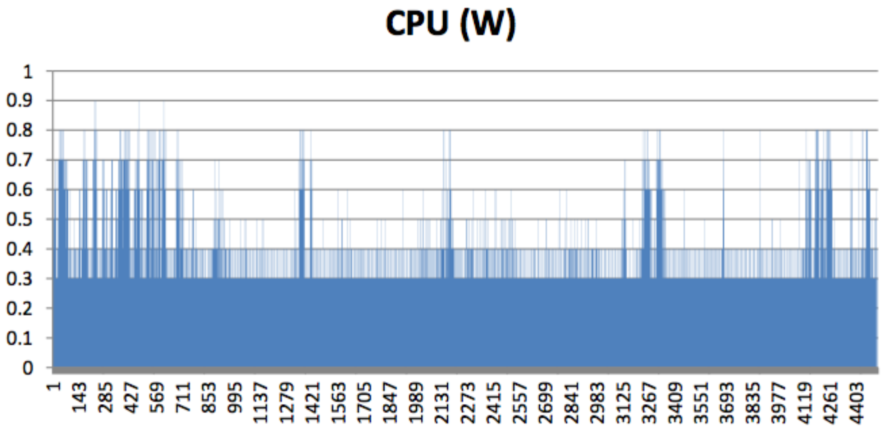


FIGURE 2. iNotes CPU power consumption data

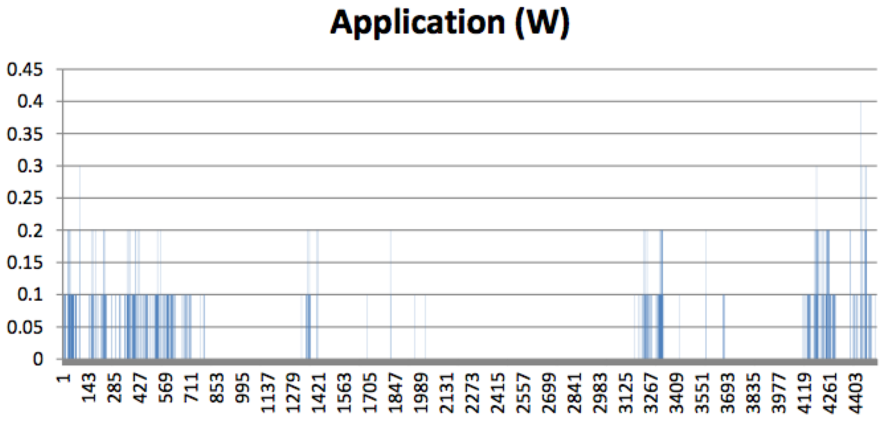


FIGURE 3. iNotes browser power consumption data

TABLE 1. Measured data recapitulation – iNotes (all values displayed in Watts)

	Total power	CPU	Display	Disk	Base	Application
avg	18.066	0.381	4.905	0.002	12.8	0.012
min	13.1	0.3	0	0	12.8	0
max	20.9	0.9	7.4	0.9	12.8	0.4

4.2. **Facebook.** The typical Facebook user (by our definition) used to read the web content, to interact with other users in different ways and discover the dynamics of the web page offered, including watching or commenting videos.

These activities are influenced by the content, but all acceptance testing is. A sample energy consumption profile is presented in Fig. 4, CPU related energy consumption data are displayed in Fig. 5 while browser power consumption data are in Fig. 6.

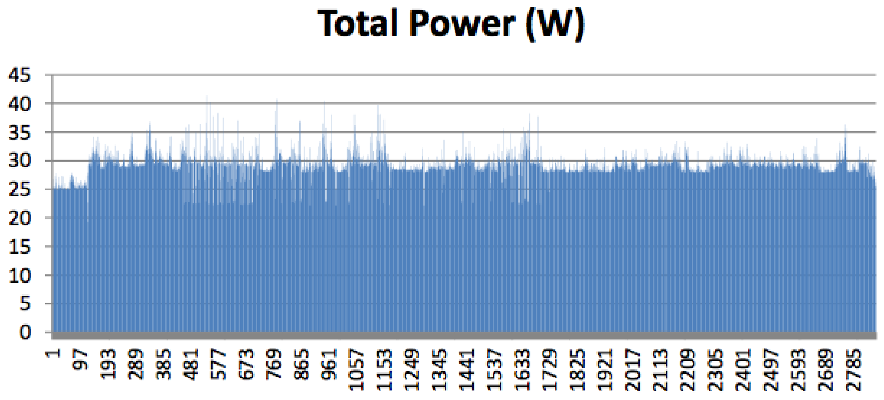


FIGURE 4. Facebook total power consumption data

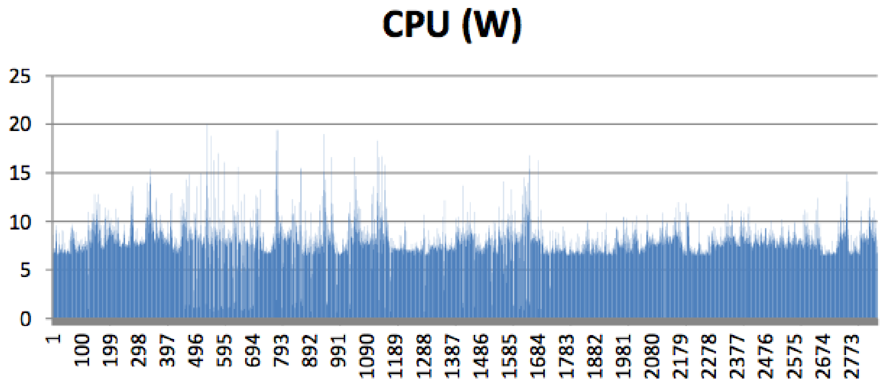


FIGURE 5. Facebook CPU power consumption data

Analysis of browser power consumption can be used to identify actions performed by the user such as starting a video playback, upload of a picture to the gallery or scrolling down the wall or a group page.

Reading of other users' posts is "hidden" as the measurement tool includes the energy consumption of the display as part of the total power consumption chart in Fig. 4.

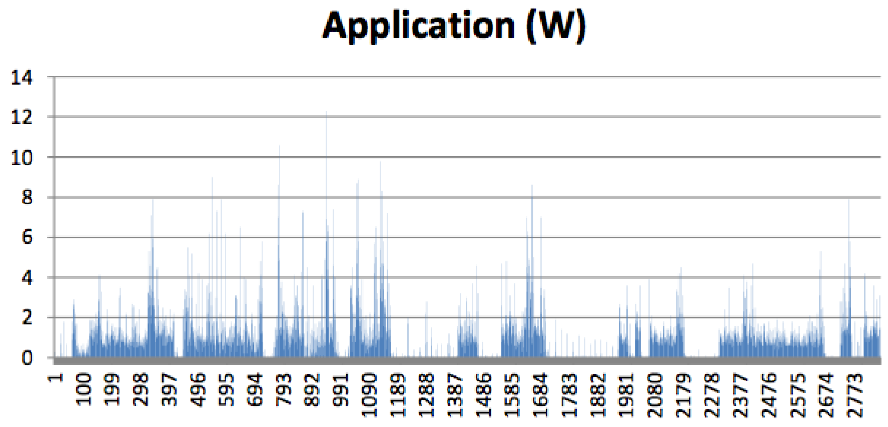


FIGURE 6. Facebook browser power consumption data

To finalize our data analysis, Tab. 2 statistically concludes our measurements.

TABLE 2. Measured data recapitulation – Facebook (all values displayed in Watts)

	Total power	CPU	Display	Disk	Base	Application
avg	29.307	8.05	9.352	0.002	11.9	0.994
min	19.3	0.6	6.3	0	11.9	0
max	41.4	20	9.5	0.7	11.9	12.3

4.3. **YouTube.** YouTube was chosen on the basis of containing the whole on videos and it is known that because of the animation and sound, this needs high energy to run it. At least this was our assumption, but this assumption was accompanied by another one related to a certain level of optimization that one can require from a specialized application.

Figure 7 shows total power consumption measured during an execution sample, more details are picked out in Fig. 8 and Fig. 9.

To finalize our data analysis, Tab. 3 statistically concludes our measurements.

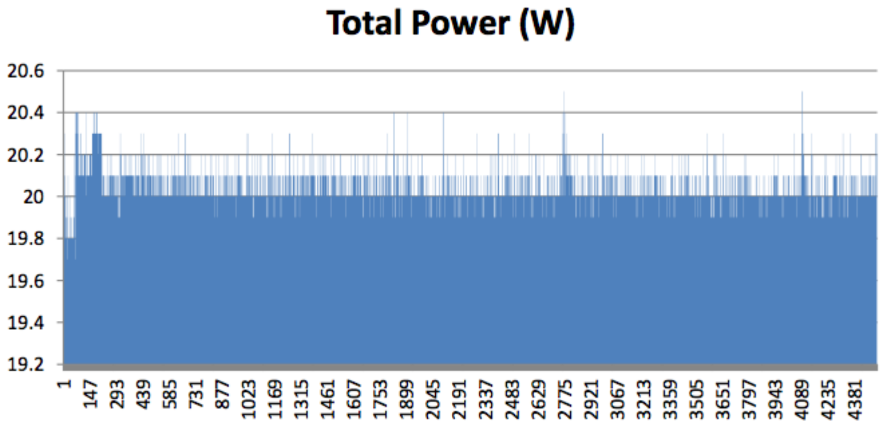


Figure 7. YouTube total power consumption data

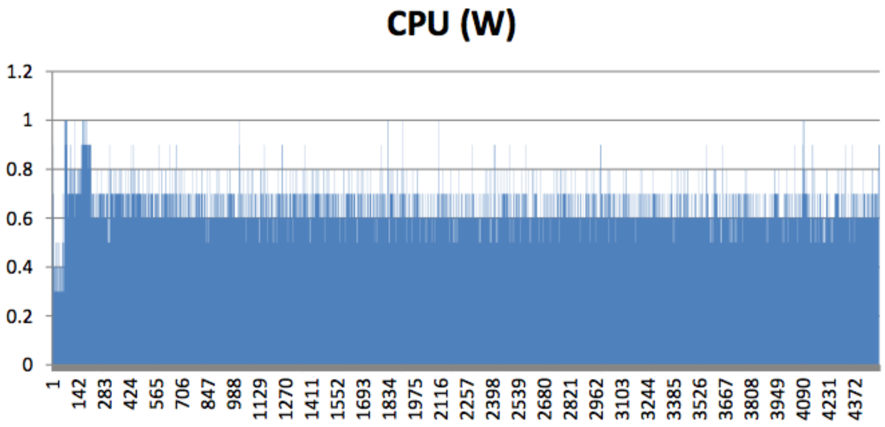


Figure 8. YouTube CPU power consumption data

Table 3. Measured data recapitulation – YouTube (all values displayed in Watts)

	Total power	CPU	Display	Disk	Base	Application
avg	25.313	9.084	4.302	0.0005	11.9	2.9493
min	17	0.8	4.3	0	11.9	0.1
max	40.8	24.1	4.8	0.3	11.9	14.9

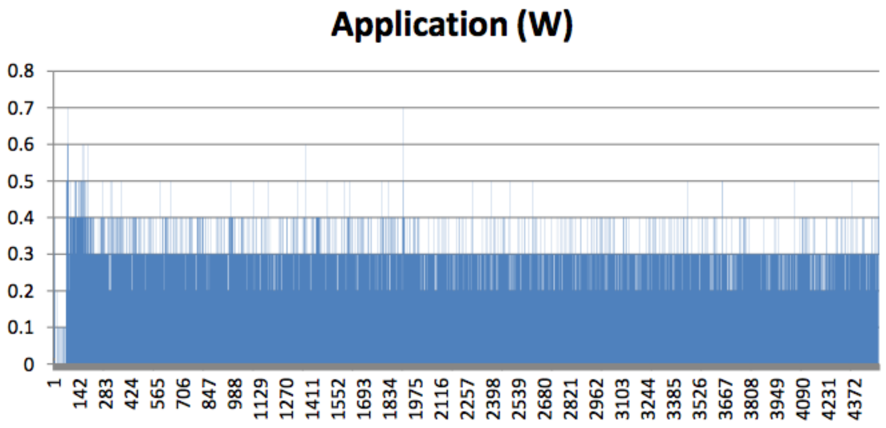


FIGURE 9. YouTube browser power consumption data

One has to note that YouTube has a different energy efficiency profile than the other two applications analyzed – it seems to consume energy all the time that cannot be only in the automatic playlist creation property. Something might be not OK in the environment.

5. Conclusion and Future Work

As mentioned in the last part of the paper, we discovered an anomaly in YouTube behavior related to energy consumption that could be partially caused by continuous preloading of content, but it could be also indicating an error in the solution used. It needs longer analysis to discover the reasons. We will start with replaying the scenarios on another operating system platform, on the top of different version of the browser and/or other browsers before making any further conclusions.

Another difference to the assumptions was the way Facebook could deal with the videos. Even that these are much shorter than the ones in the case of YouTube, the playback did not ask for so much battery power.

When talking about the actions of the measurement itself, as these are very easy, in the future we plan to automate them to provide an automated environment for measurements of comparable setups. Our future work in this area will focus on configuration of a portable integrated development, testing and energy consumption estimation environment. This environment could be used in software development or in the frame of software evolution or initial development teaching subjects to support education on energy efficiency measurement during software testing. It might limit students' creativity by offering a semi-closed sandbox, as hardware plays a very important role by the current architecture of energy consumption estimation. Some research aims to brake this limitation – we are looking forward to those results to get them also integrated.

Another interesting continuation of our work will be an evaluation of the exactly same test cases using the integrated internet browser on an Android mobile phone or tablet device. Here, the first limitation is that this platform forces the user to use specific apps instead of the browser – E-mail client, Facebook App and YouTube App. Therefore, the comparison results might look different as the limitations given by the browser are partially eliminated, which probably introduces significant differences we will have to face while running the experiments.

Acknowledgment

This paper is part of dissemination of results of the Erasmus+ Key Action 2 (Strategic partnership for higher education) project No. 2017-1-SK01-KA203-035402: "Focusing Education on Composability, Comprehensibility and Correctness of Working Software".

Co-funded by the Erasmus+ Programme of the European Union

The information and views set out in this paper are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

References

[1] C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, *The impact of source code transformations on software power and energy consumption*, Journal of Circuits, Systems, and Computers, Vol. 11, No. 5, pp. 477–502, 2002.

[2] E. Chovancová, M. Chovanec, D. Mičuta, *Social network and forum hybrid*, in Scientific Conference on Informatics, 2015 IEEE 13th International, Nov 2015, Košice, TU, 2015, pp. 124-127, ISBN 978-1-4673-9868-8.

[3] M. Couto, J. Cunha, J. P. Fernandes, R. Pereira, J. Saraiva, *Greendroid: A tool for analysing power consumption in the android ecosystem*, in Scientific Conference on Informatics, 2015 IEEE 13th International, Košice, TU, Nov 2015, pp. 73–78.

[4] J. Flinn, M. Satyanarayanan, *Powerscope: A tool for profiling the energy usage of mobile applications*, in Proc. of the Second IEEE Workshop on Mobile Computer Systems and Applications, WMCSA'99, IEEE Computer Society, 1999.

[5] M. Hudák, Š. Korečko, B. Sobota, *On architecture and performance of LIRKIS CAVE system*, in CogInfoCom 2017, Danvers, IEEE, 2017, pp. 000295-000300, ISBN 978-1-5386-1264-4.

[6] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, P. Ammann, *Ecodroid: An approach for energy-based ranking of android apps*, in Proc. of the Fourth International Workshop on Green and Sustainable Software, GREENS'15, IEEE Press, 2015, pp. 8–14.

[7] D. Li, W. G. J. Halfond, *An investigation into energy-saving programming practices for android smartphone app development*, in Proc. of the 3rd International Workshop on Green and Sustainable Software, GREENS 2014, ACM, 2014, pp. 46–53.

[8] D. Li, Y. Jin, C. Sahin, J. Clause, W. G. J. Halfond, *Integrated energy-directed test suite optimization*, in Proc. of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, ACM, 2014, pp. 339–350.

[9] K. Liu, G. Pinto, Y. D. Liu, *Data-oriented characterization of application-level energy optimization*, in Fundamental Approaches to Software Engineering, ser. LNCS, A. Egyed, I. Schaefer, eds., Springer Berlin Heidelberg, 2015, Vol. 9033, pp. 316–331.

[10] N. Pataki, Á. Sipos, Z. Porkoláb, *Measuring the Complexity of Aspect-Oriented Programs with Multiparadigm Metric*, in QAOOSE 2006 Proceedings: 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, M. Lanza, F. B. e Abreu, C. Calero, Y.-G. Guéhéneuc, H. Sahraouri, eds., Nantes, Universitá della Svizzera italiana, 2006, pp. 1-10.

[11] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, Y.-M. Wang, *Fine-grained power modeling for smartphones using system call tracing*, in Proc. of the Sixth Conference on Computer Systems, EuroSys'11, ACM, 2011, pp. 153–168.

[12] M. Santos, J. Saraiva, Z. Porkoláb, D. Krupp, *Energy Consumption Measurement of C/C++ Programs Using Clang Tooling*, in Proceedings of the SQAMIA 2017: 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Z. Budimac, ed., Belgrade, Serbia, 11-13.9.2017, Paper No. 15, 8 pages, Also published online by CEUR Workshop Proceedings No. 1938 (`http://ceur-ws.org`) ISSN 1613-0073.

[13] J. Saraiva, M. Couto, Cs. Szabó, D. Novák, *Towards Energy-Aware Coding Practices for Android*, Acta Electrotechnica et Informatica, Vol. 18, No. 1, 2018, pp. 19–25, DOI: 10.15546/aeei-2018-0003 .

[14] M. Sulír, J. Porubän, *Exposing Runtime Information through Source Code Annotations*, Acta Electrotechnica et Informatica, Vol. 17, No. 1, 2017, pp. 3–9, DOI: 10.15546/aeei-2017-0001 .

[15] Cs. Szabó, J. Saraiva, *Focusing software engineering education on green application development*, in Conference of Information Technology and Development of Education – ITRO 2017, Novi Sad, Serbia, pp. 165–169, ISBN 978-86-7672-302-7.

[16] P. R. Theja, SK. K. Babu, *Evolutionary computing based QoS oriented energy efficient VM consolidation scheme for large scale cloud data centers using random work load bench*, Annales Mathematicae et Informaticae, 46 (2016) pp. 217–241 `http://ami.ektf.hu/uploads/papers/finalpdf/AMI_46_from217to241.pdf` .

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovakia

Email address: `csaba.szabo@tuke.sk, emira.mustafa.moamer.alzeyani@student.tuke.sk`

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 1, 2018
DOI: 10.24193/subbi.2018.1.02

COMPILE-TIME FUNCTION CALL INTERCEPTION FOR TESTING IN C/C++

GÁBOR MÁRTON AND ZOLTÁN PORKOLÁB

Abstract. In C/C++, during the test development process we often have to modify the public interface of a class to replace existing dependencies; e.g. supplementary setter or constructor functions or extra template parameters are added for dependency injection. These solutions may have serious detrimental effects on the code structure and sometimes on the run-time performance as well. We introduce a new technique that makes dependency replacement possible without the modification of the production code, thus it provides an alternative way to add unit tests. Our new compile-time instrumentation technique enables us to intercept function calls and replace them in runtime. Contrary to existing function call interception (FCI) methods, we instrument the call expression instead of the callee, thus we can avoid the modification and recompilation of the function in order to intercept the call. This has a clear advantage in case of system libraries and third party shared libraries, thus it provides an alternative way to automatize tests for legacy software. We created a prototype implementation based on the LLVM compiler infrastructure which is publicly available for testing.

1. Introduction

In legacy code bases often there are few or no unit tests. Refactoring such code in order to provide tests is almost impossible because we cannot verify correctness without having unit tests; hence it is a vicious circle. We can break the circle with non-intrusive tests, i.e. without actually modifying the production code [2, 28]. Function call interception (FCI) is often the only tool

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68N15.

1998 *CR Categories and Descriptors.* D.3.3 [**Software**]: PROGRAMMING LANGUAGES – *Languages Constructs and Features*.

Key words and phrases. C++ programming language, unit testing, function call interception, compiler instrumentation.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

which enables non-intrusive testing by making it possible to replace function bodies. By replacing functions we can eliminate the unwanted dependencies in tests. With FCI we are able to intercept function calls at runtime and we can execute actions before and/or after the original function body or even completely replace it [11]. The different FCI methods have different advantages and disadvantages. Compared to languages like Java, the C and C++ languages offer less mature solutions for FCI. Java runtime reflection allows us both introspection and intercession.

In this paper, we investigate a new compile-time instrumentation based FCI approach for C/C++ programs which enables the replacement of functions and methods. By applying the instrumentation, the generated binary code will be different than the original binary program code, but the high-level C/C++ source code remains untouched. Contrary to other instrumentation methods, we instrument the call expression instead of the callee, thus we can avoid the necessity of recompilation of the function we would like to intercept. We implemented a prototype based on the LLVM/Clang compiler infrastructure.

This paper is organized as follows. In Section 2, we show the existing dynamic and static FCI methods. In Section 3, we present general test automation patterns and concepts for testing legacy code. We present how our method simplifies writing unit tests for legacy systems in Section 4. We describe our new interception technique in details in Section 5 in details. In Section 6, we describe the current limitations and possible future work. We have an overview of the related works in Section 7. Our paper concludes in Section 8.

2. Function Call Interception Techniques

We differentiate the FCI techniques based on the time FCI is applied [11]. *Dynamic techniques* perform the interception at *program load-time* or at *runtime.* Contrary to dynamic approaches, *static techniques* achieve FCI by modifying the *source files* (e.g. with the help of the preprocessor), by changing the *linkage order*, by generating object files which contains the *instrumentation* or by modifying the application *binary image*; all these modifications happen before runtime.

Load-time FCI. Most modern operating systems provide the possibility to specify shared objects to be loaded before all others. This can be used to selectively override functions in other shared objects. On Linux this behavior is controlled by the `LD_PRELOAD` environment variable [17]. With this technique, calling the original function is cumbersome. We have to use `dlsym` auxiliary function with the `RTLD_NEXT` argument [16]. In case of C++ functions we have to provide the mangled names. Furthermore, this mechanism is unreliable with member functions, because the member function pointer is not expected to have the same size as a void pointer on some platforms [10].

Run-time FCI. In Unix like systems, runtime dynamic interception is implemented with the help of the `ptrace` system call [19, 27]. If ptrace is used with the `PEEKTEXT` or `POKETEXT` argument then it is possible to attach to a running process and to read or write different segments of its memory. For instance, the GNU debugger (gdb) [7] and Intel Pin [15] both use this approach. A disadvantage of these tools is that they rely on a specific kernel functionality; thus porting these implementations to other operating systems may be hard. E.g. Intel Pin currently does not support function replacement on macOS [9]. Another property of this technique is that we cannot instrument inline functions.

Pre-compilation-time FCI. We consider some use of the C/C++ preprocessor as pre-compilation-time interception. A typical use case is to replace the `malloc` and the `free` functions from the standard C library to collect statistics about the heap usage. This approach can be applied conveniently in C, but not in C++. As soon as we use namespaces, the preprocessor might generate code which cannot be compiled because of the ambiguous use of names. Hazardous side effects of macros are also well known.

Link-time FCI. One example for the link-time static interception is the `wrap` command line option of the GNU linker (`ld`) [8]. When this program option is applied then the linker uses a wrapper function for the specified `symbol`, any undefined reference to `symbol` will be resolved to `__wrap_symbol` and any undefined reference to `__real_symbol` will be resolved to `symbol`. This approach makes it possible to replace a function and call the original. However, in case of C++ we have to specify the mangled names as symbols. We cannot use this approach if the `symbol` is defined within the very same translation unit where it is referenced.

Post-compilation-time FCI. There exist tools to modify the compiled binary code for interception. As an example, in [1] the authors describe a method which is a mixture of Link-time and Post-compilation-time techniques used to avoid typical security vulnerabilities, like buffer overflow. A modified compiler can be applied on a binary executable (or shared library) to extract type information from the debugging data and reinsert it in the same binary which is then available at runtime in a special data structure. At runtime a pre-loaded shared library intercepts the possibly dangerous calls and validates them using the data structure stored in the first step.

Compile-time FCI. Perhaps the most widely used static FCI technique is to configure the compiler to emit instrumented code in a way that interception is possible. The GNU/GCC and LLVM/Clang compilers both provide the

`-finstrument-function` program option to instrument each and every function call in a way to execute code before and after the body of the functions [6]. Actually, when this instrumentation is enabled then the compiler emits two extra calls for each function body. The prototypes of these two called functions are the following:

```
void __cyg_profile_func_enter(void *this_fn, void *call_site);
void __cyg_profile_func_exit(void *this_fn, void *call_site);
```

The arguments for these functions represent the address of the original function and the address of the instruction from where it was called. A serious limitation of this technique is that we cannot replace an intercepted function with another function; the original function will be called anyway.

3. Test Automation Conventions

The FCI techniques discussed above are frequently used in the process of creating automated tests. Thus, in this section we overview the general test automation patterns and we show the more specialized concepts about testing legacy code.

The *four-phase* test pattern is driven by the observation that each test requires some sort of setup and tear down routines. This pattern splits each test into four phases [24]. In the first phase, we set up everything that is required for the system under test (SUT) to exhibit the expected behavior. In the second phase, we interact with the SUT. In the third phase, we do whatever is necessary to determine whether the expected outcome has been obtained. In the fourth phase, we tear down the test to put the world back into the state in which we found it. This pattern is also known as the build-operate-check-clear pattern [31].

The *given-when-then* pattern of representing tests is originated from behavior-driven development [26, 3]. The *given* part describes the pre-conditions to the test. In these pre-conditions we present the state of the world before we begin the behavior we specify in the test. The *when* section represents the behavior we specify. The *then* section describes the changes we expect due to the specified behavior. We can also look at this pattern as a reformulation of the four-phase test pattern. Essentially these three states are equal to the first three states of the four-phase pattern. In the context of the four-phase pattern, Robert C. Martin states that anyone who reads the tests should be able to work out what they do very quickly, without being misled or overwhelmed by details [20]. Consequently, both the four-phase and the given-when-then patterns imply that the test setup should be strictly part of the visible test code and should not be separated from the rest of the test code. For instance, using load-time FCI to set up a test separates the "given" phase from the rest of the test code, thus it violates both patterns and makes the test hard to understand.

Unwanted dependencies embody a critical problem in software development; we often have to break existing dependencies before we can change some piece of code [28]. Breaking existing dependencies is also an important prerequisite to introduce unit tests for legacy code [2].

A *seam* is an abstract concept introduced by Feathers to identify points where we can break dependencies [2]. The goal is to have a place where we can alter the behavior of a program without modifying it in that place; this is important because editing the source code is often not an option [28]. Feathers, Rüegg and Sommerlad define four different kinds of seams for C++ [2, 28]. *Link seam*: Change the definition of a function via some linker specific setup. *Preprocessor seam*: With the help of the preprocessor, redefine function names to use an alternative implementation. *Object seam*: Based on inheritance to inject a subclass with an alternative implementation. *Compile seam*: Inject dependencies at compile-time through template parameters. The *enabling point* of a seam is the place where we can make the decision to use one behavior or another. Different seams have different enabling points.

Link and preprocessor seams can be used to write non-intrusive tests. However, object and compile seams may be used for such purpose only if the unit under test already has the proper architecture. For example, in case of object seams the unit must have a constructor (or setter) function to setup a different implementation for the dependency. In case of compile seams, the unit must be a template and it must have a template parameter via which we can mock the dependency. Often, these architectural requirements are not satisfied, therefore the use of object and compile seams ofttimes demand that we intrusively change the source code of the unit.

Some seams are realized with FCI techniques. For instance, preprocessor seams are implemented with pre-compilation-time FCI. Link seams are realized with load-time and link-time FCI. The existence of compile-time, post-compile-time and run-time FCI drives us to further extend the list of existing seams. We define a new class of seams, the *FCI seams.* More specifically we introduce three new seams for each FCI technique: *compile-time FCI seam*, *post-compile-time FCI seam* and *run-time FCI seam.*

4. Compile-time FCI Seam

In Figure 1 we present a legacy graphics program that relies on a LOGO-like API for drawing. The API is realized as a class named the `Turtle`. Also, there is `Painter` class which is responsible for drawing lines and shapes. This class has a hard-wired dependency on the concrete `Turtle` class. Still, we would

```cpp
// Turtle.hpp
class Turtle {
  int x = 0, y = 0;
public:
  void PenUp() { /* ... */ }
  void PenDown() { /* ... */ }
  void Forward(int distance) { /* ... */ }
  void Turn(int degrees) { /* ... */ }
  void GoTo(int x, int y) { /* ... */ }
  int GetX() const { return x; }
  int GetY() const { return y; }
};

class Painter {
  Turtle turtle;
public:
  void DrawLine(int x0, int y0, int x1,
                int y1) {
    turtle.GoTo(x0, y0);
    turtle.PenDown();
    turtle.GoTo(x1, y1);
    turtle.PenUp();
  }
  // ...
};
```

Figure 1. A legacy graphics program

```cpp
 1 #include "Turtle.hpp"
 2 #include <gmock/gmock.h>
 3 #include <access_private.hpp>
 4 #include <hook.hpp> // for SUBSTITUTE
 5
 6 class MockTurtle {
 7 public:
 8   MOCK_METHOD0(PenUp, void());
 9   // PenDown, Forward, ...
10 };
11
12 MockTurtle &GetMockObject(Turtle *) {
13   static MockTurtle m;
14   return m;
15 }
16
17 namespace proxy {
18 void PenUp(Turtle *self) {
19   return GetMockObject(self).PenUp();
20 }
21 // Similarly to PenDown, Forward, ...
22 }
23
24 struct TurtleTest : ::testing::Test {
25   TurtleTest() {
26     SUBSTITUTE(Turtle::PenUp, proxy::PenUp);
27     // Similarly to PenDown, Forward, ...
28   }
29 };
30
31 ACCESS_PRIVATE_FIELD(Painter, Turtle,
32                      turtle)
33
34 TEST_F(TurtleTest, TestDrawLine) {
35   using ::testing::AtLeast;
36
37   Painter painter;
38   Turtle &turtle =
39       access_private::turtle(painter);
40   MockTurtle &mockTurtle =
41       GetMockObject(&turtle);
42
43   EXPECT_CALL(mockTurtle, PenDown())
44       .Times(AtLeast(1));
45   painter.DrawLine(0, 0, 10, 10);
46 }
47
48 int main(int argc, char **argv) {
49   ::testing::InitGoogleTest(&argc, argv);
50   return RUN_ALL_TESTS();
51 }
```

Figure 2. Testing the legacy program with compile-time FCI

like to write a test which checks the `DrawLine()` function. In this example let us suppose that the turtle functions are quite expensive to use. Generally speaking, a dependency may represent a database, or a network connection, whose usage can be hard, or very expensive. Therefore, in our test we want to mock the `Turtle` class (or at least its member functions).

Our new instrumentation technique makes it possible to write non-intrusive tests easily. Figure 2 lists the test which uses our new instrumentation method. We define our mock class (`MockTurtle`) with the help of the gmock macros (lines 6-10). Our test-case is defined from line 34 to 46. In the test-case we create an instance of the `Painter` class, then we get a reference to its private `turtle` member (lines 38-39). Note that there are several different techniques to access a private member, we use a method which relies on explicit template instantiations [21]. Then we get a reference to an instance of the `MockTurtle` class which acts as a test double for the `Turtle` instance (lines 40-41). We state our expectations on the mock object (lines 43-44). In line 45 we exercise our unit under test by calling the `DrawLine()` method. With the help of our tool we setup replacement functions for each member function of the `Turtle` class (lines 26-28). These replacement functions behave as a proxy; they forward each function call on a given `Turtle` instance to a corresponding test double (lines 17-22). The way we get the reference for a relevant test double is pretty simple in this test: we return a reference to a static instance of the `MockTurtle` class (lines 12-15). We can use this simplification because we know that there is only one `Turtle` object over the lifetime of our test-case. If there were several `Turtle` objects then we should solve the mapping differently, perhaps with the help of a static hash map. Lines 48-50 contains the definition for the `main()` function which uses the functions and macros from googletest to initialize and run the test.

The most important property of this test is that the test setup is included in the test application itself. During the compilation of our test binary we have to include a header file from our auxiliary runtime library which provides the `SUBSTITUTE` macro, and we have to enable the mentioned instrumentation with a compiler switch. Also, during linking we have to link with our given runtime library. Our method has clear advantages compared to the `LD_PRELOAD` approach where we can substitute functions only if they are defined in shared libraries. With our technique it is possible to write non-intrusive tests and replace even inline functions. However, this new method requires rebuilding the application (or unit) we want to test with the specific compiler option which will disable inlining. Our technique has the following advantages: **(1)** The test setup is part of the test application and clearly visible together with the rest of the test code, thus it does not violate the given-when-then test automation pattern. **(2)** It does not introduce a new tool into the existing build chain. The functionality is embedded into the compiler. **(3)** On platforms where the compiler is supported, the new instrumentation could be supported as well. **(4)** There is no need to use mangled names. **(5)** We can use the ordinary unit test building tools and we can group unit tests into the same test application.

5. FCI with Call Expression Instrumentation

Our new interception technique and the prototype consists of two parts: a compiler instrumentation module and a runtime library. The instrumentation module modifies the code to check whether a function has to be replaced or not. The runtime library provides functions to setup the replacements.

```
char* funptr = __fake_hook(&foo);
if (funptr)
    funptr(args...);
else
    foo(args...);
```

(A)

```
char* funptr = __fake_hook(&foo);
using ReturnType = decltype(foo(args...));
ReturnType ret;
if (funptr) ret = funptr(args...);
else ret = foo(args...);
return ret;
```

(B)

FIGURE 3. Call expression substitution

5.1. **Instrumentation.** During the code generation we modify each and every function call expression to call an auxiliary function. Let us consider the following function call expression: `foo(args...);`. When our instrumentation is in action, the emitted code is equal to the pseudo code in Figure 3a. The call to `__fake_hook` resolves at runtime if we should replace the callee with another function or not. We replace a function if the returned value of `__fake_hook` is not zero, in this case the returned value is a pointer to the function we call as a substitution. If the return type of the callee function is not `void` then we create an additional storage for the return value as presented in Figure 3b. Our prototype is based on LLVM/Clang [12]. The implementation modifies the emitted LLVM Intermediate Representation (IR) [14] code. For instance, let us consider the definition of the `bar` C++ function in Figure 4a. The LLVM IR of `bar` after optimization is presented in Figure 4b. The generated code is very straightforward: there is only one basic block (`entry`) which stores the return value from the call of `foo` and then it returns with it. Note that the function names are mangled thus we see the `_Z3` prefix for the function names. When we enable our instrumentation and optimization, then the IR has the form presented in Figure 4c. Now we have four different basic blocks. The first block (`entry`) evaluates the return value of the `__fake_hook` function, compares it to zero and emits a branch based on the comparison. The `then` block is executed if the callee shall be replaced. We call the substituting function pointer, then we jump to the last basic block(`cont`). The `else` block is executed if the callee shall not be substituted; we just simply call the original function then jump to the `cont` block. At last, in the `cont` block, we store the result of either the callee or the replaced function, and we return with that.

Clang's internal architecture is built in such a way that the code generation for all kind of call expressions are eventually handled in one common routine. For example, in the case of virtual function calls the adjustment of the `this` pointer happens before calling that routine. We placed the emission of our instrumentation code inside that routine. As a result, special cases such as the `this` adjustment are automatically handled; we do not have to manually adjust the `this` pointer when we substitute a virtual function.

```
int foo(int);
int bar(int p) {
    return foo(p);
}
```

(A) C++

```
define i32 @_Z3bari(i32 %p) #0 {
entry:
  %call = tail call i32 @_Z3fooi(i32 %p)
  ret i32 %call
}
```

(B) Original LLVM IR

```
define i32 @_Z3bari(i32 %p) #0 {
entry:
  %fake_hook_result = tail call i8* @__fake_hook(i8* bitcast (i32 (i32)* @_Z3fooi to i8*))
  %0 = icmp eq i8* %fake_hook_result, null
  br i1 %0, label %else, label %then
then:                                              ; preds = %entry
  %1 = bitcast i8* %fake_hook_result to i32 (i32)*
  %subst_fun_result = tail call i32 %1(i32 %p)
  br label %cont
else:                                              ; preds = %entry
  %call = tail call i32 @_Z3fooi(i32 %p)
  br label %cont
cont:                                              ; preds = %else, %then
  %call_res.0 = phi i32 [ %subst_fun_result, %then ], [ %call, %else ]
  ret i32 %call_res.0  }
```

(C) Modified LLVM IR

FIGURE 4. LLVM IR modification for function replacement

Contradictory to `-finstrument-functions`, by instrumenting the call expressions (and not the function body) we have the convenience that we do not have to recompile dependant libraries if the call expression is in a code outside of the library. This has a clear advantage in case of system libraries, third party shared libraries and security critical applications where we have to evade library interposing. We have evaluated the prototype using various benchmarks. We measured the runtime overhead is similar to the overhead caused by the other compile-time instrumentation, `-finstrument-functions`. Detailed measurement results are available online at [23].

5.2. **Runtime Library.** The main purpose of the runtime library is to implement the `__fake_hook` function which is referenced from the instrumented code. The realization of this hook function has to find the related function pointer in case of an active substitution. Essentially, it is a simple pointer to pointer mapping which may be implemented with a simple hash function. However, in order to make the lookup as fast as possible, we chose to implement the mapping with a simple offsetting into the virtual memory (shadow memory). During program startup – more precisely, when our shared object is loaded – we initialize the shadow memory with the help of the `mmap` [18] system call. We assume that a size of a function definition is at least 1 byte, since it has to contain at least a return instruction. Let N denote the size of a pointer in bytes of a specific architecture. Since we have to store a function pointer for every function, we have to reserve a shadow memory which is N times bigger than the normal virtual address space which holds the function definitions. If the `mmap` system call is called with the `MAP_ANONYMOUS` argument then it guarantees that the reserved memory is initialized to zero. Note that in practice the OS does not zero out the mapped region during the mapping, only at the moment when a virtual addressed is being accessed the first time. We divide the user-space virtual memory into two different regions. Low memory and high memory. We handle the memory mapping differently for each region. For instance, on macOS the memory is partitioned as follows:

```
[0x7f0000000000, 0x7fffffffffff] || HighMem
[0x120000000000, 0x19ffffffffff] || HighShadow
[0x020000000000, 0x11ffffffffff] || LowShadow
[0x000000000000, 0x01ffffffffff] || LowMem
```

Let $addr$ denote the original address ,$shadowAddr$ the address of the corresponding shadow and $shadowOffset$ the offset for a region. With this formula $shadowAddr = addr * N + shadowOffset(region(addr))$ we can calculate the shadow address. By using the shadow memory instead of a simple hash map we trade execution time for space. The program occupies terabytes in virtual memory, however the resident (physical) memory usage is equal to the number of used substitutions multiplied with N. More specifically, operating systems do not reserve the specific physical pages to the process until there is no write to that memory area. Consequently, those memory pages which contain the shadow values of substituted functions will be resident physical pages registered in the process page table. In practice, this means only a few kilobytes of additional physical memory usage (given a page has 4kb size and not taking into account the Linux specific huge pages). During program startup we must make sure that our shared object gets initialized before the first function call. Our prototype achieves this by setting the `constructor` attribute [4] on the initializer function of the shared object. If there are other shared libraries linked to the final executable with such initializer functions, then it is the user's responsibility to ensure that our library is initialized first.

Another purpose of the runtime library is to provide the user interface to setup the function substitutions. Replacing a function in C is pretty simple, the shared object defines a function for that:

```
_substitute_function((const char*)&foo, (const char*)&fake_foo);
```

We may use the `SUBSTITUTE` macro in case of C++ to replace functions; this construct is more generic because it also supports member functions. Note that we have to include the header file attached to the runtime library, also we have to link with it. Our implementation is thread safe if there are multiple threads calling the very same function. Although, there is a race condition if one thread is calling the specified function while another thread is setting up the substitution; in such cases, the user code must ensure thread safety.

```
 1 template <typename Class, typename MemPtr>
 2 const char *address_of_virtual_fun(const Class *aClass, MemPtr memptr) {
 3   const char **vtable = *(const char ***)aClass;
 4   struct pointerToMember {
 5     size_t pointerOrOffset;
 6     ptrdiff_t thisAdjustment;
 7   };
 8   pointerToMember p;
 9   memcpy(&p, &memptr, sizeof(p));
10   static const size_t pfnAdjustment = 1;
11   size_t offset = (p.pointerOrOffset - pfnAdjustment) / sizeof(char *);
12   return vtable[offset];
13 }
```

Figure 5. Get the address of a virtual function

5.3. **Virtual Functions.** A pointer-to-member function may have a different layout in case of virtual functions than in case of regular member functions. Therefore, we cannot just simply cast a virtual function pointer to a `void` pointer.

5.3.1. *The naive approach.* Without compiler support, we can get the address of a virtual function in an architecture dependent way. On Figure 5 we present how we can get the address in case of the Itanium C++ ABI [10]. First, we receive the vtable from an object by dereferencing its vpointer (line 3). The vpointer is the first element in the object. We interpret the bits of the pointer to member (`memptr`) as an instance of the aggregate class `pointerToMember` (lines 4-9). Next, we setup the architecture dependent function pointer adjustment (line 10). Then, we get the offset and return with the appropriate element in the vtable (lines 11-12). We could replace virtual functions by exploiting this technique. Let us suppose we have a macro named `SUBSTITUTE_VIRTUAL` which use this technique and the following class hierarchy:

```
struct B { virtual void foo(); }; struct D : B { void foo() override; };
```

If we wanted to replace the `foo()` function when the dynamic type was `D` then we would have to get a pointer to such an instance:

```
B* dummy = new D; SUBSTITUTE_VIRTUAL(&D::foo, dummy, &D_fake_foo);
```

However, to replace the function in the base class as well, we would have to get a pointer to an instance whose dynamic type was `B`:

```
B* dummy = new B; SUBSTITUTE_VIRTUAL(&B::foo, dummy, &B_fake_foo);
```

5.3.2. *New compiler intrinsic.* The previous naive approach is ABI dependent and it also requires a reference to an existing object. Thus, we tried to find a better alternative without these restrictions. Generally speaking, in order to replace functions we just need an identifier for each function – virtual or not – which is unique in the program. Actually, each function has such a unique identifier, and it is its own address in the program's virtual memory. Unfortunately, there is no valid C++ language construct to get this unique identifier. Nevertheless, GCC has implemented this feature [5], but sadly Clang did not. Clang developers claim that this feature is fundamentally broken, because when we use it then the proper adjustment of the `this` pointer may be elided [13]. Still, our technique could use this feature since our compiler instrumentation intervenes after the this adjustment thunk is emitted. Thus, we implemented this functionality in the Clang compiler, so we are able to use it within our implementation, hidden from the users and enabled only in test code. With this approach, the replacement of the `foo()` function when the dynamic type is `D` has the following form:

```
SUBSTITUTE(D::foo, D_fake_foo);
```

This is the very same form which we can use to replace free functions or non-virtual member functions.

Internally, the `SUBSTITUTE` macro expands to a call to `_substitute_function` and the arguments of that function are generated by our new compiler intrinsic:

```
#define SUBSTITUTE(src, dst)                                              \
  do { _substitute_function((const char *)__function_id src,              \
                            (const char *)__function_id dst); } while (0)
```

We modified the compiler to parse a new kind of unary expression when the `__function_id` literal is given and the test specific instrumentation is enabled. In case of free functions and static member functions this unary expression has the very same type which we would get in case of the "address of" unary expression:

```
void foo();
void bar() {
  auto p = & foo; // void (*)()
  auto q = __function_id foo; // void (*)()
}
```

However in case of non-static member functions the two expressions yield different types:

```
struct X { void foo(); virtual void bar(); };
void bar() {
  auto p = & X::foo; // void (X::*)()
  auto q = __function_id X::foo; // void (*)()
  auto r = __function_id X::bar; // void (*)()
}
```

At runtime the value of these expressions are evaluated to hold the address of the specific raw function which can be identified by the corresponding mangled name in the compiled binary's text section.

5.4. **Overload Resolution.** We may have several functions with the same name but with different parameters. Let us consider the below code:

```
struct X {  int foo(int);  int foo(double);  };
int X_fake_foo_i(X*, int);
```

Normally, if we would like to get the address of `X::foo(int)` we have to explicitly cast a function pointer to the appropriate type:

```
int(X::*mfp)(int) = & X::foo;
```

Here, we define a pointer variable with the name `mfp` which has the type `int(X::*)(int)` and it holds the address of `X::foo`. With the `__function_id` intrinsic we have to do the same, but the type will be different:

```
int(*mfid)(int) = __function_id X::foo;
```

For safety reasons, the `__function _id` is hidden from the users of our instrumentation, but they can use the three parameter form of the provided `SUBSTITUTE` macro to replace an overloaded function. For example, to replace `X::foo` with the `X_fake_foo_i` free function one have to write:

```
SUBSTITUTE(int(int), X::foo, X_fake_foo_i);
```

6. Limitations And Future Work

Our prototype is implemented in the code generation part of the Clang compiler, however it would be architecturally better if we realized that as a transforming optimizer pass. This pass should run before all other optimizer passes. By having an optimizer pass, all the logic related to this instrumentation would be well separated and self contained. Also, it would make it possible to use our tool with other language frontends, thus this is our most important future work. Currently we do not have any check to enforce that the original function and its replacement have the same signature. In the future we plan to create a checking function template for the substitutions. The prototype works only on 64 bit x86 systems.

Replace the `operator()` of a lambda is not supported unless we can take the address of the lambda. Similarly, member functions of `structs`/`classes` which are defined inside a function cannot be replaced, because there is no valid expression to get their address. Our technique relies on that we should be able to get the address of the function we want to substitute. In case of constructors and destructors we cannot get their address with any standard C++ expression. Still, replacing constructors or destructors would be a valuable contribution in the domain of testing, thus this is an important area for further research.

7. Related Work

The different function call interception techniques are explained in details by Kang [11]. The author also discusses aspect-oriented programming implementation techniques for intercepting method calls.

The four-phase test automation pattern is introduced by Meszaros [24] and the given-when-then pattern is described by North [26]. Feathers describes different techniques about testing legacy code in his book [2]. He introduces the concept of seams via we can alter behavior without changing the original unit. Rüegg and Sommerlad elaborate this concept in C++ [28].

There are plenty of software error checking tools which are based on some kind of instrumentation. A large number of memory error detectors are based on binary instrumentation. For example, Valgrind (Memcheck) [25] or Dr. Memory. The most popular compiler instrumentation based error checker tools are the AddressSanitizer [30] and the ThreadSanitizer [29]. Our instrumentation technique was inspired by the AddressSanitizer, we reused many ideas from its implementation (e.g shadow memory). Shadow memory is often used by different error checker software. The above mentioned AddressSanitizer and ThreadSanitizer both use shadow memory to store metadata for a specific piece of memory. AddressSanitizer uses a shadow space scaled down to one eight of the normal address space and can be easily used on 32 bit systems. However, ThreadSanitizer uses 8 times larger shadow memory than the normal address range, therefore support for 32-bit platforms is problematic and is not planned by the maintainers.

8. Conclusion

Test seams are used to create non-intrusive tests for legacy systems, some of these seams are often realized via an FCI technique. We introduced our new compiler instrumentation for C and C++ programs, which makes it possible to replace the intercepted function call. While most of the existing instrumentation methods modify the function to call we instrument the caller side. We substitute the actual call with a small code snippet in compilation time, which decides at runtime whether the original or a replacement function is about to call. The decision is made using shadow memory and an offset to minimize runtime overhead. In contrast to other seams, our new instrumentation seam keeps the test setup code close to the other phases of the test. The technique makes it feasible to write non-intrusive tests which follow the given-when-then test pattern. This way, our method could help to implement high-quality tests for legacy software systems.

Compared to existing compile-time instrumentation solutions, our technique does not require the modification or even the recompilation of the intercepted functions, which is a possible advantage in case of legacy code, system libraries, third party shared libraries or in situations when we have to avoid library interposing. We have created a prototype implementation using the LLVM/Clang compiler infrastructure, which is publicly available at [22].

References

[1] Kumar Avijit et al. "Binary Rewriting and Call Interception for Efficient Runtime Protection Against Buffer Overflows: Research Articles". In: *Softw. Pract. Exper.* 36.9 (July 2006), pp. 971–998. ISSN: 0038-0644. URL: `http://dx.doi.org/10.1002/spe.v36:9`.

[2] Michael Feathers. *Working Effectively with Legacy Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004. ISBN: 0131177052.

[3] Martin Fowler. *GivenWhenThen*
. URL: `https://martinfowler.com/bliki/GivenWhenThen.html`.

[4] gcc.gnu.org. *Declaring Attributes of Functions*. 2017. URL: `https://gcc.gnu.org/onlinedocs/gcc-4.3.0/gcc/Function-Attributes.html` (visited on 06/24/2017).

[5] gcc.gnu.org. *Extracting the function pointer from a bound pointer to member function*. 2017. URL: `https://gcc.gnu.org/onlinedocs/gcc-4.9.0/gcc/Bound-member-functions.html` (visited on 06/24/2017).

[6] gcc.gnu.org. *Program Instrumentation Options*. 2017. URL: `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html` (visited on 06/24/2017).

[7] gnu.org. *GDB: The GNU Project Debugger*. 2017. URL: `https://www.gnu.org/software/gdb/` (visited on 06/24/2017).

[8] gnu.org. *Using GNU ld*. 2017. URL: `ftp://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html` (visited on 06/24/2017).

[9] Intel. *Pintool API Reference - RTN: Routine Object*. 2017. URL: `https://software.intel.com/sites/landingpage/pintool/docs/53271/Pin/html/group__RTN__BASIC__API.html` (visited on 06/24/2017).

[10] Intel et al. *Itanium C++ ABI*. 2017. URL: `http://refspecs.linuxbase.org/cxxabi-1.83.html` (visited on 06/24/2017).

[11] Pilsung Kang. "Function call interception techniques". In: *Software: Practice and Experience* (). spe.2501, n/a–n/a. ISSN: 1097-024X. URL: `http://dx.doi.org/10.1002/spe.2501`.

[12] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, Mar. 2004.

[13] llvm.org. *Clang will not accept a conversion from a bound pmf to a regular method pointer*. 2017. URL: `https://bugs.llvm.org/show_bug.cgi?id=22121` (visited on 06/24/2017).

[14] llvm.org. *LLVM Language Reference Manual*. 2017. URL: `http://llvm.org/docs/LangRef.html` (visited on 06/25/2017).

[15] Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 190–200. ISBN: 1-59593-056-6. URL: `http://doi.acm.org/10.1145/1065010.1065034`.

[16] Linux Programmer's Manual. *dlsym, dlvsym - obtain address of a symbol in a shared object or executable.* 2017. URL: `http://man7.org/linux/man-pages/man3/dlsym.3.html` (visited on 06/24/2017).

[17] Linux Programmer's Manual. *ld.so, ld-linux.so - dynamic linker/loader.* 2017. URL: `http://man7.org/linux/man-pages/man8/ld.so.8.html` (visited on 06/24/2017).

[18] Linux Programmer's Manual. *mmap, munmap - map or unmap files or devices into memory.* 2017. URL: `http://man7.org/linux/man-pages/man2/mmap.2.html` (visited on 06/24/2017).

[19] Linux Programmer's Manual. *ptrace - process trace.* 2017. URL: `http://man7.org/linux/man-pages/man2/ptrace.2.html` (visited on 06/24/2017).

[20] Robert C Martin. *Clean code: a handbook of agile software craftsmanship.* Pearson Education, 2009.

[21] Gábor Márton. *Access Private.* 2017. URL: `https://goo.gl/ynaZv5` (visited on 06/25/2017).

[22] Gábor Márton. *finstrument-mock - Instrumentation for Testing.* 2017. URL: `https://github.com/martong/finstrument_mock` (visited on 06/25/2017).

[23] Gábor Márton. *Performance Measurements of finstrument-mock.* 2017. URL: `https://github.com/martong/finstrument_mock/blob/master/measure/performance_evaluation.pdf` (visited on 03/28/2018).

[24] Gerard Meszaros. *xUnit test patterns: Refactoring test code.* Pearson Education, 2007.

[25] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. URL: `http://doi.acm.org/10.1145/1273442.1250746`.

[26] D North. *Introducing BDD, Better Software Magazine.* 2006.

[27] Pradeep Padala. "Playing with ptrace, Part I". In: 103 (Nov. 2002). ISSN: 1075-3583 (print), 1938-3827 (electronic).

[28] Michael Rüegg and Peter Sommerlad. "Refactoring Towards Seams in C++". In: *Proceedings of the 7th International Workshop on Automation of Software Test.* AST '12. Zurich, Switzerland: IEEE Press, 2012, pp. 117–123. ISBN: 978-1-4673-1822-8. URL: `http://dl.acm.org/citation.cfm?id=2663608.2663632`.

[29] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer: Data Race Detection in Practice". In: *Proceedings of the Workshop on Binary Instrumentation and Applications.* WBIA '09. New York, New York, USA: ACM, 2009, pp. 62–71. ISBN: 978-1-60558-793-6. URL: `http://doi.acm.org/10.1145/1791194.1791203`.

[30] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference.* USENIX ATC'12. Boston, MA: USENIX Association, 2012, pp. 28–28. URL: `http://dl.acm.org/citation.cfm?id=2342821.2342849`.

[31] Sai Venkatakrishnan. *Build Operate Check Clear - Test Pattern.* URL: `http://developer-in-test.blogspot.hu/2009/05/build-operate-check-clear-test-pattern.html`.

Department of Programming Languages and Compilers, Eötvös Loránd University

Email address: `martong@mailbox.elte.hu, gsd@elte.hu`

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 1, 2018
DOI: 10.24193/subbi.2018.1.03

TRANSLATING ERLANG STATE MACHINES TO UML USING TRIPLE GRAPH GRAMMARS

DÁNIEL LUKÁCS AND MELINDA TÓTH

Abstract. In this paper, we present a method that transforms event-driven Erlang state machines into high-level state machine models represented in UML. We formalized the transformation system as a triple graph grammar, a special case of graph rewriting. We argue in this paper that using this well-defined formal procedure opens up the way for verifying the transformation system, synchronizing code and formal documentation, and executing state machine models among many other possible use cases. We also provide an example transformation system and demonstrate its application in action on a small Erlang state machine. We also present our evaluation of our full system implementation tested on real world Erlang state machines.

1. Introduction

In this paper, we introduce a method to generate formal UML state machine models from executable Erlang state machine source code (e.g. `gen_fsm` applications [13]). First, this transformation makes use of the RefactorErl static analysis framework [11, 18] to analyse the application source code, then it will transform and synthesize the program representation resulting from the analysis, into a UML state machine model.

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68N15, 68Q42.

1998 *CR Categories and Descriptors.* D.2.2 [**SOFTWARE ENGINEERING**]: Design Tools and Techniques – *Computer-aided software engineering (CASE)*; D.2.1 [**SOFTWARE ENGINEERING**]: Requirements/Specifications – *Languages*; F.4.2 [**MATHEMATICAL LOGIC AND FORMAL LANGUAGES**]: Grammars and Other Rewriting Systems – *Parallel rewriting systems*; F.3.22 [**LOGICS AND MEANINGS OF PROGRAMS**]: Semantics of Programming Languages – *Program analysis.*

Key words and phrases. Erlang, triple graph grammar, UML, CASE, state machine, model transformation.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

1.1. **Motivation.** Currently, Unified Modeling Language (UML) [4] is mostly used in practice as a documentation tool to present a diverse set of views (we refer to as *models* in this paper) on high complexity software code.

As both time and tangible costs of formation and maintenance of documentation increases with software size, automatic generation of collective knowledge representation becomes more and more important as the software grows. Apart from automatically generating well-defined UML models of Erlang state machines, the focus of this paper is to achieve this in a formal way, specifically by using a model transformation approach called triple graph grammar.

Triple graph grammars [17] (TGGs) are special kind of graph rewriting systems, and as formal methods, they may have the same properties proved for them, such as correctness, completeness, determinism, and confluence. Proven transformation systems automatically produce verified documentation.

One property of special interest regarding TGG transformations is information preservation, i.e. the expectation that result models can be transformed back to the source models. Such bidirectional transformations, and the more effective model integration (finding correspondences between models) and model synchronization (completing partially complete models and correspondences) allow generation of software code from documentation and vice versa, providing a way to keep the two in sync all the time. This enables developers to work on the software in various abstraction levels.

As some UML metamodels (such as state machines) can be used to describe program semantics, model execution by automatic generation of Erlang `gen_fsm` implementations of UML state machines is also a possible future direction of this research.

Verification of the transformation system, automatic verification and synchronization of documentation, bidirectional TGGs, model execution and round-trip editing are future goals and possible applications of this research. This current paper only focuses on the target model generation aspects.

1.2. **Background.** One way to represent the execution history of a computer program is to take snapshots of the state of the program memory. State machines can be used to abstract away this low level representation. A state describes a segment of the program behavior, while a state transition describes a change in such behaviors, usually triggered by an event [16].

Among many others, UML [4] is one of the more accepted standards to represent state machines. The UML state machine language can be used to formally describe event-driven systems, i.e. systems that wait for certain events, and upon the occurrence of these events, they change their behavior and wait for a possibly different set of events. etc.

Erlang [13] is a general purpose, functional, dynamically typed, open source programming language, mostly used to develop multithreaded, real time, fault tolerant applications, like telecommunication systems, web servers, or distributed databases. The language provides various abstractions (called *behaviors*) to support these applications provided by the built-in OTP library, including the state machine (`gen_fsm`) behavior.

The requirements of the `gen_fsm` behavior are to implement a callback function, named `init`, and any number of transition functions. The function `init` will designate, at a minimum, the initial state of the state machine. The transition functions will designate, at a minimum, the next state the state machine will be in when it receives a specific event, while in a specific state. All the logic necessary to handle multiple threads, messages, events, etc. will be handled by Erlang in accordance to the `gen_fsm` semantics [13].

In our research, we use the RefactorErl static analysis framework [18] to analyse Erlang source code. RefactorErl analyses the source code and stores discovered syntactic and semantic information in a database called the *semantics program graph (SPG)*. The framework provides several features to run refactorings on the source code, to perform further analyses – like data flow, and dynamic function call analysis –, to execute various queries, to calculate certain metrics, and many other features.

2. Transformation pipeline

In this section, we overview our approach that we refer to as *transformation pipeline*. The pipeline can be considered as a simple function composition, where the output of earlier functions will be the input of the latter functions.

As depicted in Figure 1, the main input of pipeline is an Erlang SPG as stored by the RefactorErl framework. This is transformed into an SPG model (a highly detailed view of the transformed software code), in order to refine it into a high-level state machine model using model transformation methodology. Finally, this high-level state machine model is translated into a selected state machine model, e.g. UML.

In order to properly understand models and model transformations, we need to introduce basic modeling terminology. The Object Management Group Meta-Object Facility (OMG MOF) [3] highlights four cognitive layers of software modeling: concrete implementation, model, metamodel, and meta-metamodel, each in order a higher-level abstraction (or language) that enables expressing the layer below.

As most of the intermediate values in the pipeline are models, we also indicated in Figure 1 the metamodels generating these models. Models of SPGs are formalized using the SPG metamodel (see Figure 8), state machines

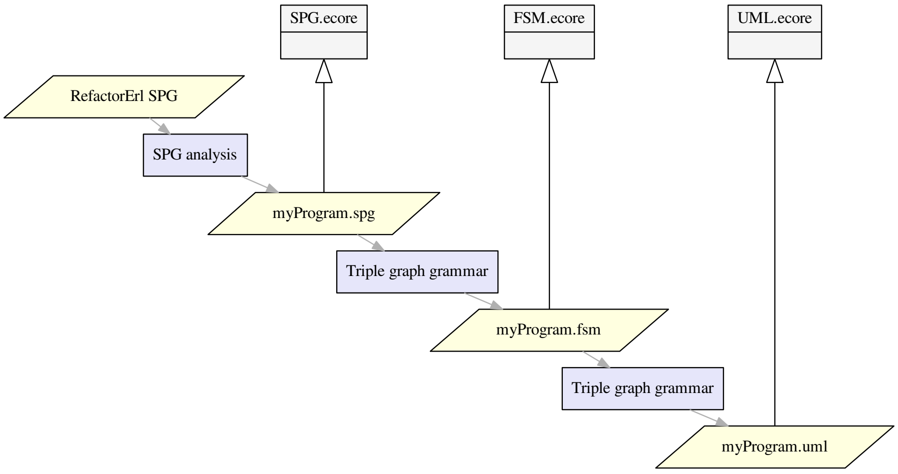


Figure 1. Transformation pipeline refining Erlang state machines to high-level UML models

are represented internally using the FSM metamodel (see Figure 2), and the output state machines are formalized in UML.

As it is conventional in model-driven engineering (e.g. in UML and EMF Ecore), we formally represent models using typed attribute graphs [7], where nodes and edges are mapped to types and unique key-value stores.

2.1. **Internal program representation of RefactorErl.** The RefactorErl analysis framework stores all information it gathers about Erlang programs via static analysis in a special data structure, called the *semantic program graph (SPG)* [11]. In this paper, we show how the SPG can be transformed into a state machine, which corresponds to the original state machine described by the original Erlang source code.

The first step in the transformation pipeline is the transformation of the RefactorErl SPG into an SPG model (a highly detailed view of the source program), which can be then transformed into a state machine model (a more abstract view of the source program) using standard model transformation tools. A diagram of our SPG metamodel can be found in our earlier work [15], and it also is depicted by Figure 8. During the transformation, we have to perform semantics queries, dataflow analysis and dynamic function call analysis provided by RefactorErl [20, 19, 10], and encode the results in the model. This way we avoid re-implementing RefactorErl static analysis facilities for models.

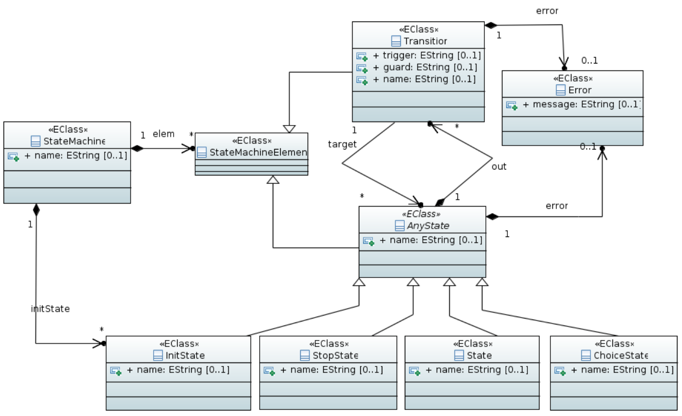


FIGURE 2. Metamodel for representing state machine models so they can be easily translated to UML

2.2. **A metamodel for describing abstract state machines.** In this section we will describe a simple state machine metamodel, already published in our earlier work [15], and depicted by Figure 2, with which we represent the target state machines of the transformation. We showed in [14] how this metamodel (and its instances) can be mapped onto the UML state machine metamodel (and its instances). This intermediate state machine language explicitly highlights the elements we utilize from UML. For implementation purposes, the intermediate state machine can be omitted altogether, by substituting the UML state machine element descriptions for the corresponding elements in our notation.

2.3. **Triple graph grammars.** The pipeline also includes two model transformation steps expressed using *triple graph grammars (TGGs).* The term *graph grammar* is a synonym for *graph rewriting system (GRS)*, where the term grammar signifies the intent of language generation as opposed to e.g. program evaluation with graph reduction. TGGs constitute a special class of graph grammars, where the graph is a *triple graph (TG)*: a side-by-side representation of two models with correspondence nodes connecting them. We may consider the correspondence nodes as hyperedges that connect multiple source and target nodes.

The domains given by the metamodels of the source and target models, plus the domain of the correspondence graph always partitions the node and edge

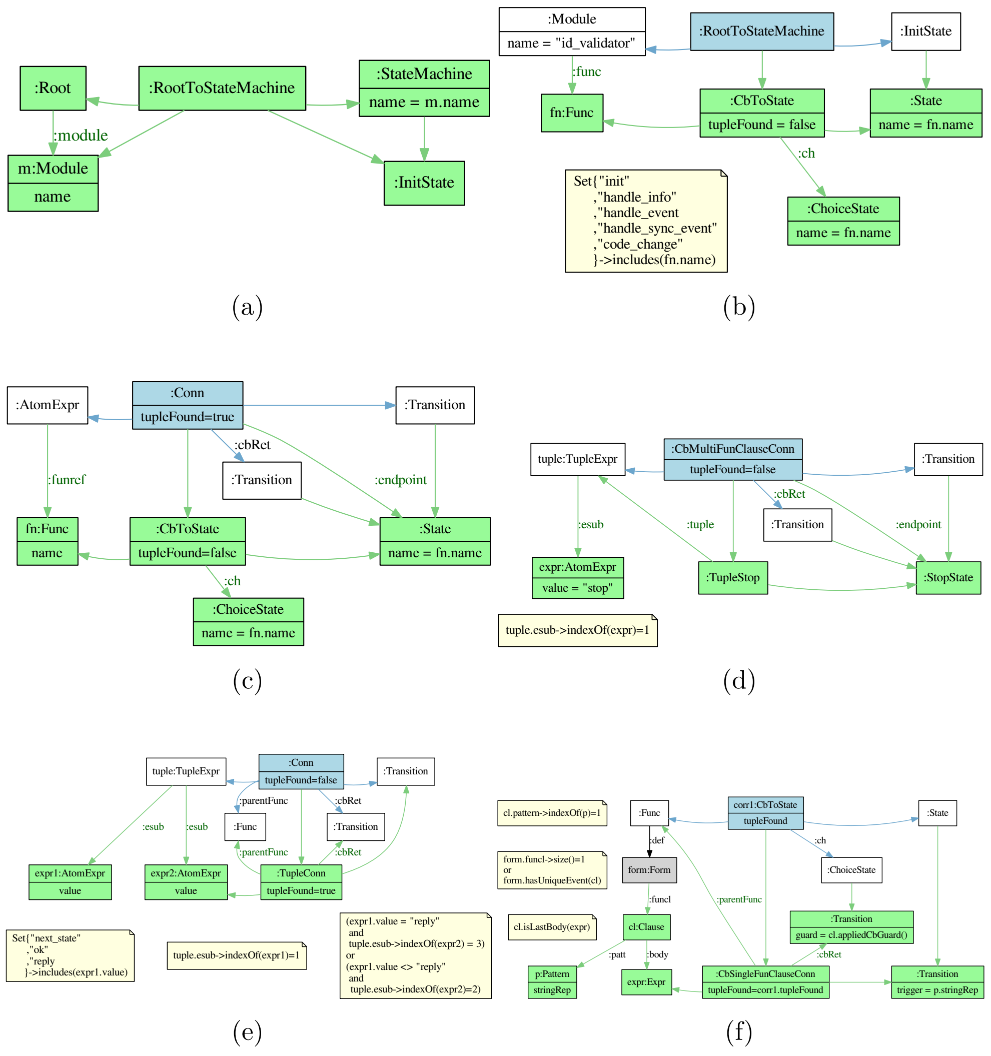


FIGURE 3. Example TGG transformation system

set of both data and rules TGs into three disjunct partitions: the *source*, the *target*, and the *correspondence domains*.

A TGG consists of two kinds of rules: axioms and production rules. TGs are generated recursively by first acquiring an initial TG using the axiom, and then applying the production rules first to the initial TG and then to successive TGs to acquire the final TG. As in GRSs, the left-hand side (LHS) of TGG production rules is matched using graph matching to a TG redex and the matched values are substituted in the right-hand side (RHS) variables. The resulting concrete RHS is then substituted in place of the redex. We say that an element is *bound* if it was already matched or replaced at least once during the transformation, otherwise we say it is *unbound*.

In this paper, we use a concise notation for diagrammatically representing TGG rules using diagrams (see Figure 3). As side-by-side representation of LHS and RHS usually includes large number of identical nodes, we merge the two in one diagram and use color coding to keep the two comparable. White elements (called *context elements*) appear on both LHS and RHS, so these are "read-only" elements, that are matched and left unchanged. Context elements can be matched both to bound and unbound elements in the data TG. The meaning of green elements depends on whether they occur in the source domain, or not. In the source domain they denote context nodes, that can only be matched to unbound elements (thus preventing infinite applications of the same rule to the same contexts). In other domains they denote RHS-only elements (called *product elements*): instead they are instantiated when the LHS matches.

The application semantics of axioms are identical to those of production rules, but axioms never contain context nodes. This guarantees that axioms are always matched and applied before production rules.

Reusable elements denoted by the color gray (see e.g. Figure 3f) can be matched both to bound and unbound elements in the data TG, but if such a match does not exist, then these elements are created in the data TG.

Most TGG formalisms also allow rules to be accompanied by *OCL constraints*. These are boolean expressions pertaining the attributes and values of the matched nodes. If a rule is successfully pattern matched, then the OCL constrains are evaluated and the match will be accepted based on whether the expression is satisfied or not.

2.4. **Transformation system.** In this section, we present a small, simplified subset of our TGG transformation system that transforms model SPGs of `gen_fsm` programs to state machine models. This rule set was included only to illustrate and to aid in the comprehension of Section 3. To develop a TGG rule set that assumes full coverage of the Erlang language, various problems had to be solved: propagation of state between rule applications, transforming parallel paths in the graph, transforming nodes with arbitrary number of incoming or outgoing edges, transforming recursively nested expressions.

As it is difficult to demonstrate all these problems in a small example, we avoided the discussion of related details in the rules not occurring in the example. We detail in [14] the whole set consisting of 32 rules, of which the largest one has 16 nodes. According to [12] practical TGGs have in average 15-20 rules, with 10-40 nodes each. The larger number of rules in our case is explained by the number of types and syntactic categories in the Erlang language.

As the rules in the system have disjunct LHSs and/or mutually exclusive OCL constraints, the system is guaranteed to be deterministic.

Figure 3a depicts an axiom rule. In the source domain, it matches in the data TG the bound SPG root node and module node implementing the `gen_fsm` behavior. It then generates and binds in the TG the root node of the state machine and a globally unique initial state, and a correspondence node between the source and target elements. The `name` attribute of the module will also be matched to a value, and the state machine will be created with its own `name` attribute set to this value.

The production rule in Figure 3b matches the SPG nodes of the initial `gen_fsm` callback functions and creates a state for each of them, and a transition into this state from the initial state. The name of the new state will be set to the name of the matched function.

The transformation traverses the SPG model starting from the initial functions and identifies next states in the function return values. As transition functions bear the name of their source state, we can now identify the transition function to traverse next. The leaves of the search tree are stop states and already bound states.

Figure 3c illustrates the identification of new functions, assuming we already matched the atom in the predecessor functions return value. It is similar to Figure 3b, but we expect the `tupleFound` attribute of the correspondence node to be true: this signifies that the atom in question is part of the expression tree headed by a tuple. Successive correspondence nodes will have their `tupleFound` attribute set to false again, as from here on we will analyse the body of another function.

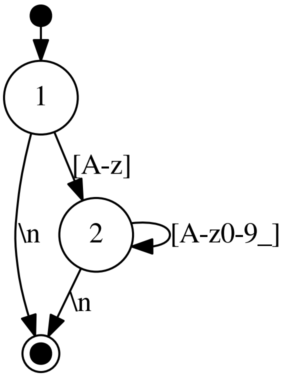
Figure 3d introduces a stop state if the return value of the transition function is a tuple whose first element is the atom `stop`.

If the first element is not the `stop` atom, Figure 3e enables the traversal of the expression tree of the second tuple element by rules specific to Erlang expression types. It also sets `tupleFound` to true to allow Figure 3c to match when an atom is found. To keep the rules simple we omitted the case where the first element of the tuple is an unreduced expression, but this case can be handled similarly.

As most functional languages, Erlang also allows functions to have multiple clauses. When `gen_fsm` transition functions have multiple clauses, each clause may declare transitions into different states, therefore in such cases, we introduce a `ChoiceState` for each event handled by the function. Clause conditions (expression patterns and guards) will be mapped to guards of the transitions commencing from the `ChoiceState`. On the other hand, when a transition function has only one clause, we do not want to introduce a `ChoiceState`, as it would only have a single transition. Figure 3f introduces a rule for handling function clauses which have a unique state machine event pattern (this includes clauses that are entirely single).

```
-module(id_validator).
-behavior(gen_fsm).

init(_) ->
  {ok, pos1, []}.

terminate(_, _,_) -> ok.

pos1($\n, _, _) -> {stop, normal, {[], reject}, []};
pos1(X, _, _) ->
  case alpha(X) of
    true -> {reply, {[X], step}, posOther, []};
    false -> {stop, normal, {[X], reject}, []}
  end.

posOther($\n, _, _) -> {stop, normal, {[], accept}, []};
posOther(X, _, _) ->
  case (alphanumeric(X) or X == $_) of
    true -> {reply, {[X], step}, posOther, []};
    false -> {stop, normal, {[X], reject}, []}
  end.
```

(a)



(b)

FIGURE 4. Source code and schematic diagram of the Erlang state machine that accepts the language of identifiers

3. Demonstration

The goal of this section is to demonstrate the example transformation system on a small Erlang state machine. To keep the example system and the demonstration section small and concise, we only transform the first part of the model SPG of this program. We included a trace of the transformation of the full syntax tree in [14].

Figure 4a depicts the code of an Erlang `gen_fsm` state machine that accepts the language of identifiers, i.e. those words (event sequences) that start with a letter and continue either with letters, numbers, or underscores (see Figure 4b). Initially, the `gen_fsm` behavior first executes the `init/1` function, and sets the current state to the state name (`pos1`) inside the tuple returned by this function. When an event is sent to the state machine, the `gen_fsm` behavior will evaluate the transition function corresponding to the current state (`pos1/3`) and now in turn sets the current state to the return value of this function. The state machine stops (and either rejects or accepts the event sequence received beforehand) when one of the transition functions return a tuple with the (`stop`) atom.

Figure 5 depicts the transformation trace, i.e. the triple graph resulting from applying the transformation system in Section 2.4 to the model SPG of this program. Nodes not featured in the final result were omitted for the sake of simplicity. The left-hand side of the TG stores the original model SPG unchanged, while the right-hand side stores the resulting state machine. In middle, the correspondence graph tells us the history (i.e. the rule application sequence) of the transformation.

First, the axiom rule (Figure 3a) was the only rule that could have been matched to the root node, and thus the state machine root was created. Next, the rule in Figure 3b is matched, as `init/1` is part of the five callback functions expected by the `gen_fsm` behavior specification. We handle these functions similarly to transition functions, and therefore we create a special state for `init/1` too. On the only function clause, the rule in Figure 3f is applied. This rule selects the last expression of the function body and assigns a transition corresponding to this clause. The transition will be labeled by the wildcard trigger which is the first parameter pattern of the clause. As the selected expression may be nested, further rules must be applied to find the name of the next state in this expression. In the current case, the expression is a tuple and matches the rule in Figure 3e, since the first element of the tuple is the atom `reply` and its third element is an atom. The `tupleFound` attribute of the correspondence node is set, so that rules aimed to match only subexpressions of the tuple may match. And indeed, the rule in Figure 3c matches: it finds that the third element of the tuple is the atom `pos1`, and thus it binds the node representing the transition function `pos1/3` and creates the corresponding state.

As `pos1/3` has two clauses the rule in Figure 3f matches both. In the case, where the event is the end-of-line character, the result is a tuple with the `stop` atom as its first element, thus the rule in Figure 3d matches and creates a stop state. The last expression of the other clause is a branching expression. A choice state will necessarily correspond to these expressions, and then each branch will have a corresponding transition from the choice state. For brevity, we omitted the rules needed to perform these transformations, along with the rest of the trace. Both can be found in [14].

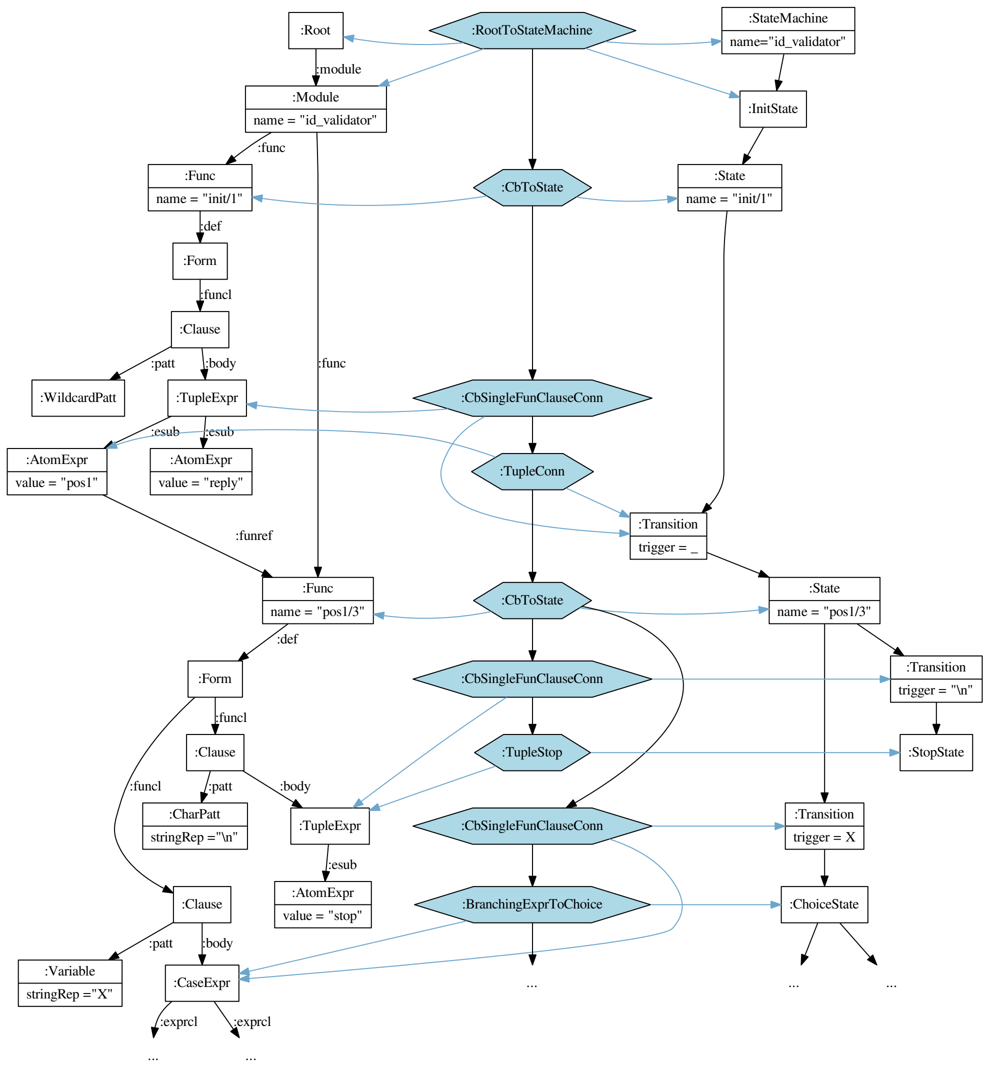


FIGURE 5. Partial trace of the transformation applied to the SPG of the Erlang state machine in Figure 4a

The final result of the transformation (including the translation between our abstract state machine metamodel and UML) is shown by Figure 6 visualized by txtUML [6].

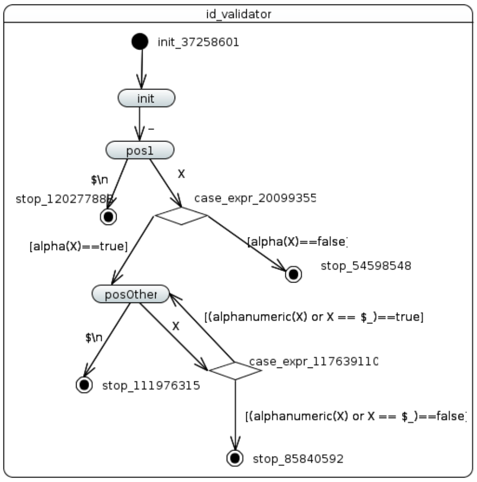


FIGURE 6. The final UML state machine resulting from the transformation of the Erlang state machine in Figure 4a

4. EVALUATION

In this section, we discuss and evaluate our implementation of the approach presented in this paper for transforming Erlang state machines to UML. As a superset of the small example transformation system introduced earlier, our implementation consists of 32 rules, of which the largest one has 16 nodes, and in our experience it is capable of transforming arbitrary Erlang state machines implementing the `gen_fsm` behavior. This larger system is detailed in [14].

In the implementation, we used the RefactorErl framework and its semantic query, dataflow analysis, and dynamic function call analysis facilities to construct the model SPG in EMF Ecore. We implemented the TGG transformation system in the TGG Interpreter [8] tool, as it supports custom correspondence metamodels, OCL constraints, and reusable nodes. Finally, we used the txtUML [6] framework to visualize UML state machines.

For evaluating the implementation we select state machine modules from large, open source Erlang projects: the Ejabberd communication server, the Riak distributed NoSQL database, and the Erlang OTP library. The UML state machine model resulting from the transformation of a smaller module is depicted in Figure 7. For each module, we list the number of lines of code, the average, variance, median, and extrema of required time (in miliseconds) for performing the transformation based on 10 measurements, and the number of states and transitions in the result. States include choice states and entry and exit states, so the number of states and transitions in the model may exceed the actual number of states in the original Erlang state machine. Time values only concern the transformation of SPG metamodel instances to abstract state machine metamodel instances. The time needed to load the Erlang application in RefactorErl is mostly independent of the state machine module, as in most cases its (often much larger) dependencies also have to be loaded. In our experience, the time needed to translate abstract state machines to UML was negligible compared to the SPG transformation.

Module	LoC	#States	#Transitions	Avg (ms)	Var (ms)	Med (ms)	Min (ms)	Max (ms)
Ejabberd								
ejabberd_c2s	3128	46	90	31614.7	9504.77	28402.0	22933	47799
ejabberd_http_bind	1236	22	23	26350.2	7598.95	23545.0	21857	47411
ejabberd_http_ws	355	14	13	4924.8	429.09	4742.5	4543	5879
ejabberd_odbc	692	7	10	3128.3	185.25	3109.5	2827	3498
ejabberd_s2s_in	712	34	48	22819.6	3478.57	22498.0	17400	29095
ejabberd_s2s_out	1367	80	104	75686.1	6822.30	73770.0	65274	86216
ejabberd_service	404	22	23	17991.3	3778.44	17035.0	13501	24403
eldap	1196	19	32	27353.0	7569.92	25156.5	21774	47353
mod_irc_connection	1581	26	25	28931.5	9540.48	24327.0	18330	40679
mod_muc_room	4501	32	73	37370.5	7095.74	36863.0	29746	52170
mod_proxy65_stream	291	29	32	14975.9	3016.94	13558.0	12955	22192
mod_sip_proxy	458	19	24	10418.7	1423.34	10162.5	8329	12292
Riak								
riak_kv_2i_aae	695	15	22	11688.4	2191.49	10927.0	10183	17181
riak_kv_get_fsm	787	16	16	5521.9	1408.34	4878.0	4324	8377
riak_kv_put_fsm	1055	25	39	12630.0	1822.41	12816.0	10862	16779
riak_kv_mrc_sink	439	14	23	7795.4	1302.02	7417.0	6455,	10546
Erlang OTP								
ssh_connection_handler	1721	42	65	67467.6	3578.80	66465.5	63174	73920
tls_connection	975	61	80	56788.5	3906.75	55821.5	50383	63514

Table 1. Runtime evaluation results

The time needed weakly positively correlates with the number of lines of code (LOC). We can explain this by assuming that more complex state machines also need more time to be processed, and complexity grows linearly in the best case (exponentially in the worst case) with the depth of Erlang expressions (e.g. larger call chains), which in turn may correlate with the LOC. And indeed, based on Table 1, transformation time grows linearly with the number of states in the state machines. Future work may consider more elaborate source code and model metrics to provide better estimates of transformation time and output.

Comparing the time demand of the TGG approach to our less formal, more implementation-centered approach in [15], we can conclude – based on the performance of the current implementation and TGG Interpreter – that the price of the formal guarantees provided by TGGs is the decreased runtime efficiency of the transformation. Still, we believe the advantages provided by TGGs make this approach superior, especially as a reference implementation, that can be used as a foundation for later optimizations.

5. Related work

This work is the continuation of our earlier research published in [15]. There, we present a less formal approach to automatically generate UML models from Erlang state machines. The algorithm presented there is an extended depth first search that selects the neighbouring nodes to discover based on predefined rules, while it simultaneously constructs the output state machine. While this procedure was more efficient, it lacked several features TGG promises for future research: verification of transformation properties, model executability, and model synchronization. We intend this current paper as a foundation to realize these features.

One work with similar goals is Erlesy [1], a readily usable, lightweight solution to visualize Erlang state machines in various output formats, like Graphviz, PlantUML or D3.js. Unlike our approach, Erlesy uses loop edges to model the handle callbacks of the `gen_fsm` specification: an advantage of this approach is that it follows `gen_fsm` semantics more closely, a disadvantage is that it inevitably clutters the resulting state machine graphs with loop edges.

There is also a mature methodology for discovering deterministic finite state machines using dynamic code analysis, called state machine induction and behavioral inference. Procedures applying this methodology execute the analysed program based on specific use case scenarios (e.g. a sequence of function calls), and collect information to generate a state machine model. The Erlang language is also well suited for this task due to its statelessness and advanced program execution tracing facilities [5].

To implement the transformation system introduced in this paper, we used the TGG Interpreter [8] tool. A survey of various TGGs can be found in [9], that compares MoTE, eMoflon and TGG Interpreter regarding their usability, expressivity and provided formal guarantees. All three tools are based on the EMF framework [2]. Other related tools are Henshin-TGG, EMorF, and OMG QVT.

6. Conclusions and Future Work

In this work, we presented an approach to transform Erlang state machines into high-level state machine models represented in UML using triple graph grammars. For demonstrating this approach, we provided an example transformation system, which we used to explain basic ideas about the semantics of triple graph grammars and the core problems regarding transformation of Erlang syntax trees to high-level state machine models. For the full system consisting of 32 TGG rules, we referred the reader to our earlier technical report. Our implementation of the system used the static analysis facilities of the RefactorErl framework to construct the model SPG in EMF Ecore, and we implemented the transformation system in the TGG Interpreter tool. We also evaluated the results and efficiency of this implementation.

One certain use case for a tool generating high-level models from code is automatic documentation generation, but TGGs promise possibilities way beyond that. Future research may consider the possibilities of developing a system for reverse transformation to achieve executable UML state machines. A bidirectional TGG system may make automatic synchronization of code and documentation possible and open up the way for round-trip editing to enable development on the most adequate abstraction level.

As TGGs are a special kind of graph rewriting systems, they may have the same properties formally proved for them, such as correctness, completeness, determinism, and confluence. To enable automatic verification of the resulting documentation artifacts, future research may also set out to prove the transformation system introduced in this paper.

References

[1] Visualising Erlang development . `https://github.com/haljin/erlesy`. Accessed: 2016-06-30.

[2] Eclipse Foundation. Eclipse Modeling Framework (EMF). `https://eclipse.org/modeling/emf/`. Accessed: 2015.11.30.

[3] Object Management Group. OMG Meta Object Facility (MOF) Core Specification. `http://www.omg.org/spec/MOF/`. Accessed: 2015.11.30.

[4] Object Management Group. OMG Unified Modeling Language Superstructure. `www.omg.org/spec/UML/`. Accessed: 2016-06-30.

[5] Arts, T. and Holmqvist, C. In the need of a design... reverse engineering Erlang software. 10th International Erlang User Conference, EUC. 2004.10.

[6] Dévai, G., Kovács, G. F., and Ancsin, A. Textual, executable, translatable UML. Proceedings of 14th International Workshop on OCL and Textual Modeling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014) Valencia, Spain, September 30, 2014., pages 3-12.

[7] Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. (2006). *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series).* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[8] Greenyer, J. and Rieke, J. (2012). Applying advanced tgg concepts for a complex transformation of sequence diagram specifications to timed game automata. In Schürr, A., Varró, D., and Varró, G., editors, *Applications of Graph Transformations with Industrial Relevance*, pages 222–237, Berlin, Heidelberg. Springer Berlin Heidelberg.

[9] Hildebrandt, S., Lambers, L., Holger, G., Rieke, J., Greenyer, J., Schäfer, W., Lauder, M., Anjorin, A., and Schürr, A. (2013). A Survey of Triple Graph Grammar Tools. In *Bidirectional Transformations*, volume 57, pages 1–18. EC-EASST.

[10] Horpácsi, D. and Kőszegi, J. (2013). Static analysis of function calls in erlang. *e-Informatica Software Engineering Journal*, 7:65–76.

[11] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, A. N., Nagy, T., Tóth, M., and Király, R. (2009). Modeling semantic knowledge in Erlang for refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, volume 54(2009) Sp. Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania.

[12] Kindler, E. and Wagner, R. (2018). Triple graph grammars: Concepts, extensions, implementations, and application scenarios.

[13] Logan, M., Merritt, E., and Carlsson, R. (2010). *Erlang and OTP in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

[14] Lukács, D. (2016). Erlang állapotgépek modell alapú és transzformációja UML-re. Scientific Students' Associations Conference, ELTE, Budapest, Hungary.

[15] Lukács, D., Tóth, M., and Bozó, I. Transforming Erlang finite state machines. In CEUR Workshop Proceedings 2046: pp. 197-218. (2018) Proceedings of the 11th Joint Conference on Mathematics and Computer Science (MACS16). Eger, Hungary, 20-22 May, 2016.

[16] Samek, M. (2009). *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. Electronics & Electrical. Taylor & Francis.

[17] Schürr, A. (1995). *Specification of graph translators with triple graph grammars*, pages 151–163. Springer Berlin Heidelberg, Berlin, Heidelberg.

[18] Tóth, M. and Bozó, I. (2012). Static Analysis of Complex Software Systems Implemented in Erlang. In *Central European Functional Programming School*, volume 7241 of *Lecture Notes in Computer Science*, pages 440–498. Springer.

[19] Tóth, M., Bozó, I., Horváth, Z., and Tejfel, M. (2010). First order flow analysis for Erlang. In *Proceedings of the 8th Joint Conference on Mathematics and Computer Science (MACS), ISBN:978-963-9056-38-1*.

[20] Tóth, M., Bozó, I., Kőszegi, J., and Horváth, Z. Static Analysis Based Support for Program Comprehension in Erlang. In Acta Electrotechnica et Informatica, Volume 11, Number 03, October 2011. Publisher: Versita, Warsaw, ISSN 1335-8243 (print), ISSN 1338-3957 (online), pages 3-10.

7. Appendix

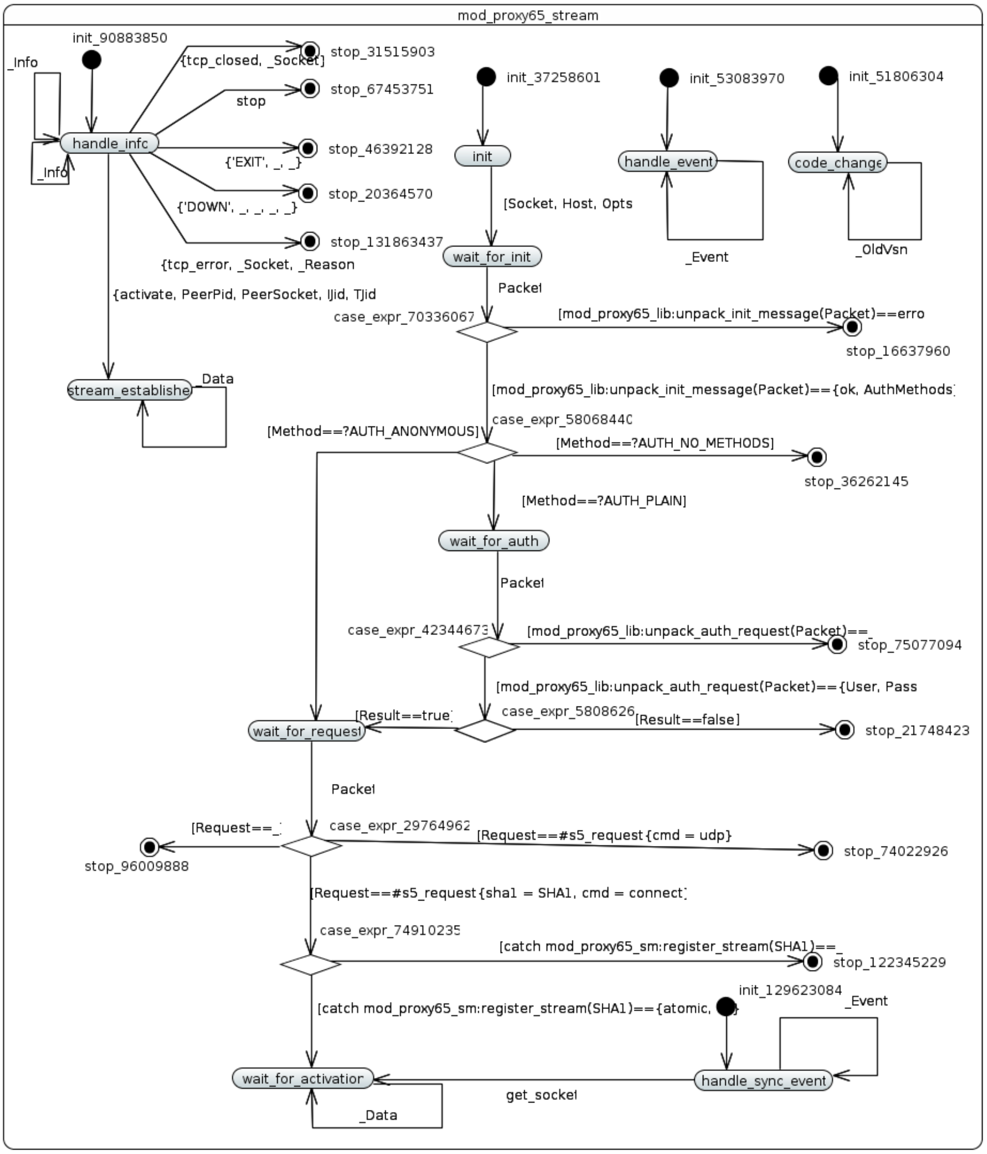


Figure 7. The final UML state machine resulting from the transformation of the `mod_proxy65_stream` state machine in Ejabberd

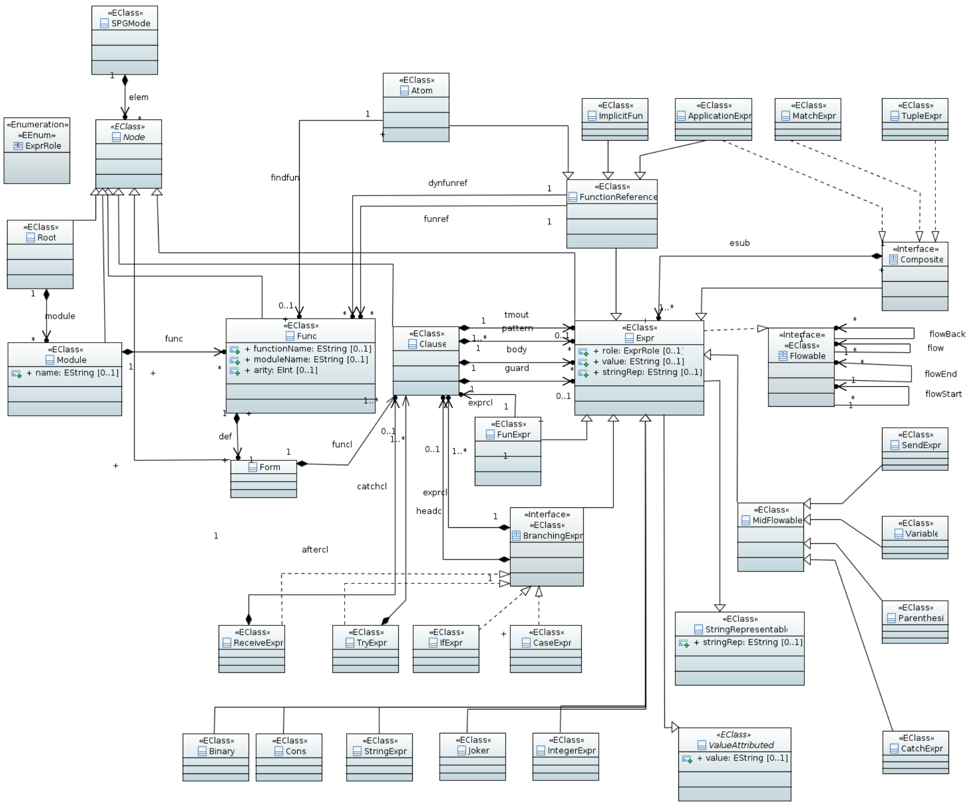


FIGURE 8. Metamodel for representing RefactorErl semantic program graphs as models

EÖTVÖS LORÁND UNIVERSITY, BUDAPEST, HUNGARY
Email address: {dlukacs,tothmelinda}@caesar.elte.hu

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 1, 2018
DOI: 10.24193/subbi.2018.1.04

DETECTING BINARY INCOMPATIBLE SOFTWARE COMPONENTS USING DYNAMIC LOADER

ÁRON BARÁTH AND ZOLTÁN PORKOLÁB

Abstract. Modern programming languages support modular development dividing the system into separate translation units and compile them individually. A linker is used then to assemble together these units either statically or dynamically. This process, however, introduces implicit dependences between the translation units. When one or more units are modified in inconsistent way binary incompatibility occurs and may result in unexpected program behavior. Current mainstream programming languages neither specify what are the binary compatibility rules nor provide tools to check them.

In this paper we discuss the details of various cases of binary incompatibility. We implemented a prototype solution in the Welltype programming language to detect binary compatibility by dynamic loader.

1. Introduction

Most of the modern programming languages provide some way for modular development. Program code is usually written into separate source files and compiled individually. These are so called *translation units* [1] then organized into higher abstraction packages, modules or libraries. The separation level of the compilation of these translation units are vary in different programming languages. The Java language [2] requires the proper setting of the `CLASSPATH` environment variable to make connection between translation units. In C and C++ languages [3, 4] usually the header files included to multiple translation units provide the consistency.

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68N15.

1998 *CR Categories and Descriptors.* D.3.3 [**Software**]: PROGRAMMING LANGUAGES – *Languages Constructs and Features*;

Key words and phrases. programming language, linking, binary compatibility.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

To create an executable code, the individually compiled translation units are assembled into an executable program. For classical programming languages like C, C++, Fortran, etc. where the compilation step results machine specific binary code, some kind of linker [5, 6] connects the translated units. This can happen either *statically*, where the assembled units form a unified entity, or *dynamically*, when the necessary code is collected only in run-time. Modern software systems tend to use the dynamic approach [7, 8] as it results smaller binary code and faster compile/link time.

For programming languages using some virtual execution environment, e.g. the virtual machine in Java, the run-time environment provides the proper connection between the units.

There are a number of advantages of this code organization: programmers can work on individual source files with minimal interference. Libraries created from a set of translation unit form reusable subsystems. Compilers can better localize the possible issues when translate the source code. On incremental development only the modified code should be recompiled thus the development time is shorter.

Although the translation units are compiled individually, in many cases there are implicit dependences between them. One unit can use variables or functions defined in some other unit. Objects are defined in one unit as instances of types defined in an other unit. When one or more components are changed most programming languages require full recompilation of the system to ensure the complete consistency between the units. In practice, however, the full recompilation of the system is rarely the case.

This paper is organized as follows. In Section 2 we overview how the current mainstream languages support binary compatibility. In Section 3 we describe a typical industrial scenario to point to the importance of the binary compatibility and its verifiability. In Section 4 we introduce our prototype solution for the problem in the Welltype experimental programming language. We evaluate this approach in Section 5. We briefly discuss our future plans in Section 6. Our paper concludes in Section 7.

2. Related work

Binary compatibility is an issue poorly recognized by language designers, but can cause serious headache for maintainers of large software projects. When already compiled clients are linked against different versions of libraries, incompatible library versions can cause the client code to crash or even worse, to running in undefined way. This problem frequently occurs with C/C++ programs using dynamic libraries, but the issue is not limited to C++, also happens in Java and other languages. Welltype deeply validates modules to link and forbids incompatible usage.

A classical solution to create binary compatible versions for classes is using the handle-body programming pattern [9]. In C++ this is frequently called as the PIMLP pattern and implemented as a single private data member – a smart or raw pointer – referring to an implementation class written in a separate translation unit. As the evolution of the class is reflected in changes only of the implementation, the object layout of the original class used by the clients never changes. On the negative side of this solution we usually have to allocate the implementation class on the heap which may result run-time overhead.

In C and C++ the GNU compiler team developed a solution [10, 11, 12] to append version number to symbols in the ELF (Executable and Linkable Format) [13, 14] files. These informations later can be used by the static or dynamic linker. This solution might be useful to detect some sort of binary incompatible components.

Even the Java programmers must be aware binary compatibility, although, the Java language is not known about program crashes due to binary incompatible components. The binary compatibility has an own chapter in the *Java Language Specification* [15], suggesting the importance of this topic. The chapter detailing what will produce a binary compatible output, and what are the traps. To understand the importance of binary compatibility in Java programs, we must take a closer look to the problems caused by library upgrades [16]. Another example – which is related to library upgrades – is when a refactoring is made [17].

Apart Welltype, other languages were developed to be aware of binary compatibility. ZL is a C++ compatible language in which high-level constructs, such as classes are defined using macros over a C-like core language [18]. This approach makes many parts of the language easily customizable, e.g. the programmer can have complete control over the memory layout of objects. Using this capability, one can develop binary compatible new versions of ZL language objects.

3. Motivation

Suppose, we have a large software system, implemented in an object-oriented programming language, like C++. Here, many of the subcomponents of the system, like networking, logging, database connections, etc. are implemented as classes or a group of classes placed into libraries. Each of these subcomponents have their own maintenance cycle: they evolves implementing new features, are changed due to bug fixes or performance improvements. If the system is large enough, it is not realistic to recompile the whole system when one or more subsystems have changed.

In the industry a typical solution is the following. Each subsystem is implemented in separate translation units and compiled into *dynamically loadable libraries* (e.g. DLLs in Windows, shared objects in UNIX systems). The public *interface* of the subsystems are exposed in *header files.* Applications are using this common header files via the `#include` preprocessor directive. The applications also using the *implementation* of the subsystems picking the corresponding dynamically loadable libraries in run-time when the application starts.

This scenario allows a relatively good opportunity to maintain even large systems. When any of the subsystems requires changes for maintenance purposes it is enough to change and recompile the one in question. Replacing the old `.DLL` or `.so` with the new version the changes will be enabled for the applications on their next start. However, this upgrade scenario does not allows changes on the interface of the subsystems. Any time the public interface of a subsystem changes, applications using it should be recompiled.

Unfortunately, not changing the public interface does not guarantee the binary compatibility of the system components. The C++ programming language uses *value semantic*, i.e. objects are mapped into bytes in memory directly instead of being represented by some reference which points to heap allocated memory (like Java does it). Every time we declare a variable of a type we allocate the corresponding number of bytes. Client code using that object is directly compiled to utilize *size*, and *offsets* corresponding the object's known layout. Changing the layout, e.g. adding a new (non static) member to a class or changing existing ones brakes these assumptions. To avoid inconsistency between the object's actual layout and the layout known by the already compiled clients we should apply only *binary compatible* changes.

What is a binary compatible change is very hard to decide. Language specifications, like the ISO C++ standard [19], do not even mention binary compatibility. Subtle changes, like making an existing member function virtual or adding a new exception may brake compatibility. Experts are collecting traps and pitfalls [20, 21, 22], but those are specific to platform, compiler or even compiler flags to set optimization level. At the moment there are no reliable tool or method to check whether a new library is binary compatible with its previous version.

Nowadays, the programs are stored in well-known binary formats, and the used format varies through operating system. For example, modern Unix and Linux systems use the ELF [13, 14] format (Executable and Linkable Format) since the nineties. Windows systems use the PE [23] format (Portable

Executable). The common in these formats that they provide symbol sharing across dynamic libraries and other programs. The mechanism is unfortunately is too simple to provide possibility to detect binary incompatibility. The C-style linkage consists of basically just function names without any additional information. The C++-style linkage, however, uses mangled names which encapsulates additional information about the function: encodes the type names of the arguments, including `namespace`s and templates.

4. Welltype Dynamic Loader

As we seen above, binary compatibility is really an issue in long-term development. The incompatibilities can cause the program to crash or, even worse, miscalculations that break invariants. Thus, programmers must take actions to avoid such incompatibilities. While in the current, mainstream programming languages only conventions can get rid of binary incompatibilities, the Welltype language explicitly and strictly specifies the binary compatibility.

The Welltype language [24, 25, 26] is an imperative programming language, designed to be safe: strict syntax and strong type system. While the Welltype language is safe, it is still feature rich – supports algebraic data types. Welltype programs can be dynamically linked, performed by the dynamic loader.

The Welltype Dynamic Loader will validate whether the program to be loaded meets the already loaded restrictions. If the loader finds a program is binary incompatible, then it will be refused from loading.

In order for the dynamic loader to be capable to make decision like that, programs must define their public interface: what elements they require and what elements they publish into the environment. These *elements* can be types, functions, operators, etc. Note that this is a notable difference from the standard ELF or PE formats, where only the function symbol names are stored, and everything else is assumed. However, this applies mainly to the C language, because the C++ symbols contain more information, even type names, since it is required to resolve function overloads.

The Welltype specification states that all external references must be explicitly indicated, the source code must contain what elements needs to be *imported*. These references are used twice: First, during the compilation to let the compiler know what undefined but declared elements can be used. Second, during the dynamic linking to bind the externals to the program. After all externals are bound, the program can be executed. Naturally, the indicated externals will be part of the compiled program if they actually referenced – the unreferenced externals will be ignored.

On the other hand, any program can *export* a set of elements into the runtime environment. These exported elements later can be used by other programs. The *one definition rule* [27] – which is a basic concept of the Welltype language – cannot be violated when exporting functions into the runtime environment. This rule will specify a loading order among the programs, since no program can actually import an undefined element.

The importance of the one-definition rule can be easily understood. It is trivial to define two functions with the same signature while their implementation is different. In order to use the same implementation by all the loaded programs, it is mandatory to enforce this rule. Note that, however, it is not required to load *one* specific implementation into the environment. This dynamic mechanism allows to load different implementations to the same program – while the signatures are match.

The loading procedure will process all export and import sections, in the order specified in the program binary. The algorithm used to export elements is the following:

```
procedure export_element(environ, program, elem)
  if not element_is_exported(environ, elem) then
    if element_is_function(elem) then
      error: one definition rule violated
    end if
  end if
  rep := build_representation(program, elem)
  register(environ, rep)
end procedure
```

The algorithm used to import elements is a bit more complicated but straightforward:

```
procedure import_element(environ, program, elem)
  rep := find_element_by_primary_attr(environ, elem)
  if rep is NIL then
    error: unresolved external
  end if
  if not element_is_compatible(elem, rep) then
    error: element is incompatible
  end if
  write_import_info(program, elem, rep)
  add_reference(program, rep)
end procedure
```

The mentioned *primary attributes* are unique to all elements described in Section 4.1.

4.1. **Elements in the binary interface.** The concept of the binary interface consists of the following elements: *function signatures*, *operators*, *exceptions*, and the *types.*

The mentioned **function signature** – as expected – take part in the binary interface. The function signature used in the binary consists of

- the name of the function, which is almost a custom zero-terminated string with lesser exceptions;
- the number of argument;
- the exact types of all the arguments;
- the number of return values;
- the exact types of all return values;
- the `pure` attribute
- and (in case of export) the address of the entry point of the function.

If the signatures match, the loader assumes they functions are binary compatible. The current version of the Welltype language does not specify other attributes to identify a function. This specification might looks inadequate, but the strength of the dynamic loader (and the Welltype language itself) are the types.

Primary attributes of the function signature are: name, number of argument, and the types of the arguments. This is similar to the C++ language, because this is the minimal information required to resolve function overloads.

The mentioned *exact types* in the listing above refers to the in-depth type matching. The binary interface must hold the specification of all types that are involved in export or import mechanism. This applies recursively to other types as well. This topic will be discussed later.

In addition to the already mentioned elements, **operators** are also take part in the binary interface. The Welltype specification aimed the goal to load programs that ,,speak the same interface". Therefore, the used operators are also matched, while it is somewhat unnecessary. The reason behind this design decision is that two expressions are not the same if the operator precedences are not compatible. For example, the expression `a *+ b *- c` can be interpreted two ways depending on the precedences:

(1) `(a *+ b) *- c`
(2) `a *+ (b *- c)`

Moreover, the associativity of the operator is also important, because the expression `a *+ b *+ c` also can be interpreted two ways:

(1) `(a *+ b) *+ c`
(2) `a *+ (b *+ c)`

Therefore, the different precedence and associativity cause the source code to be not the same with different settings. Thus the operator specification consists of:

- operator symbol (we distinguish two varieties: the *classic* operator form which consists of one or more operator symbol character with lesser exceptions; and the *identifier* form which is any identifier that accepted by the parser) – this is the only primary attribute;
- precedence ranging from 3 to 15;
- and the associativity (left or right).

The built-in operator set is fixed in the Welltype specification. Multi-defined operators are also ignored. It is done because if all the three attributes match, it defines exactly the same operator, and will be turn into a simple import.

The **exceptions** are also part of the binary interface. A Welltype program can raise only exceptions that are defined in the program or imported into the program. Note that the declared but not exported exceptions will be implicitly exported, because the program needs a global exception identifier (that globals to the runtime environment). This mechanism will not cause any problems, because exceptions can be exported multiple times while not violating the one-definition rule. Technically, the duplicate exception exports are ignored, and the program will implicitly import it. This can be done because the only informations about an exception is its name. Furthermore, the implicit export is used to make reference to the original program that actually exported the exception. Using the method, all program will know which exception to be raised, and which exception to be caught. Since the exceptions are identified in the binary only by its name, the name of the exception is the primary attribute.

4.2. **Types in the binary interface.**

In this section we discuss all the types in details that are part of the binary interface.

Types that specified in the current version of the Welltype language are: enumeration type, function type, data type, record, private record and limited record. The primary attribute is common to all types listed here: the name of the type.

The **enumeration type** consists of:

- name of the type;
- number of enumerators;
- identifier of all the enumerators.

Exact match of the enumeration type is important because the programs will communicate with enumerator indices, and every index must refer to the same enumerator. Also, code might be generated on the imported side. Therefore, the number and the identifier of the enumerators must match.

The primary attribute of the **function type** is only its name, because the Welltype specification identifies all types by only their name. This is a little bit contrary to the specification of the function signature, but the *function type* is a type, not an actual function. However, the function type consists the same attributes as the function signature:

- name of the type;
- the number of argument;
- the exact types of all the arguments;
- the number of return values;
- the exact types of all return values;
- and the `pure` attribute.

In order to import a function type, all attributes listed above must exactly match.

The **data type** is the algebraic data type implementation in Welltype. Therefore, the binary representation must reflect the complexity of this type. The following attributes are stored:

- name of the type;
- number of constructors;
- index of the default constructor;
- for each constructor:
  - name of the constructor;
  - number of types in the constructor;
  - list of the types.

The reason why the index of the default constructor is included in the binary representation is similar to the explanation to the operators. This attribute is somewhat unnecessary, but using different default constructor can result totally different program from the same source code. Moreover, if the actual data type does not have a default constructor, then an important attribute will change. Without default constructor the type is not *default constructible*, and this recursively affect other types. The dynamic loader takes actions to avoid to alter such important attribute like the *default constructible*.

The **record** type is quite similar to the `struct` used in C programming language with one major difference: the complete layout is stored in the compiled binary, and validated by the dynamic loader. The stored layout consists of:

- the name of the type;
- the number of fields the record has;
- the list of the field types;
- and the names of all the fields.

The number of the fields and the types of all the fields are very important, because code will be generated on the importer side, which is specific to the actual layout. For example, the size of the whole record depends on its fields. The records passed through different programs must use the same layout, otherwise crash or miscalculation will occur. The field names are stored only in order to ensure that the records are the same. Example to the importance of the field names can be seen in Figure 1. The layout is definitely the same (two `int` in both cases), but with entirely different semantics. For example, accessing the `y` field will use different memory slots. Therefore, the two records are binary incompatible, but this scenario can be detected only if the record fields are stored.

```
record vec
{
  int x;
  int y;
}
/* or */
record vec
{
  int y; // NOTE: the fields are
  int x; //       swapped
}
```

FIGURE 1. Example code breaking the binary compatibility.

```
record my_record
{
  int  first;
  bool second;
  long third;
}
```

FIGURE 2. Example record to demonstrate the serialization.

For example, the record can be seen in Figure 2 will be compiled into the following sequence:

```
"my_record", 3, 2, 1, 4, "first", "second", "third"
```

Where the type indices are `int`=2, `bool`=1 and `long`=4.

The record type might not the best choice for all situations, because the record may evolve, or the representation intended to be hidden. Since the record type matches all fields, and the fields are free to access, this construct is not optimal to these purposes. The **private record** is used instead. Only the name of the type is stored (hence it is *„private”*), and the representation is entirely hidden. For the private record type additional functions are required to be imported (or additional functions must be exported). Because the fields are unknown, thus a constructor function is required to construct them. Note that all private record are still *default constructible* despite of the representation is unknown.

The *non-default constructible* version of the private record is the **limited record**. The semantics is the same as the private record, but the constructor function is not imported/exported. Apart from the *default constructible* attribute, the limited record and the private record are the same construction.

5. Evaluation

Although the binary interface is quite strict, the mechanism is actually usable. A few large (over 20k sloc) Welltype programs are written that highly uses the dynamic linking feature: this provides an ability to these programs to replace the back-end implementation. With the help of the opaque types (`private record` and `limited record`) the details can be hidden, and the back-end can be reduced to a simple API (Application Programming Interface) instead of a over-complicated and embedded implementation. Thus, this organization makes the back-end implementation replaceable, not least easy to understand.

Also, this supposed to be a motivating force to programmers design a compact and clear API. Moreover, this approach forces not to leak implementation details, which in most of the cases is absolutely unnecessary. In C++, the needlessly leaked implementation details are considered as *bad practice*, in Welltype, however, they considered as *never do that.*

6. Future work

The current version of the Welltype language does not support classes. However, introducing the class construction brings issues into the strict syntax, and the forced binary compatibility. As we seen, the Java language has problems with binary compatibility. Thus, this language construction required to be carefully designed to suit into the strict syntax, semantics and binary interface.

7. Conclusion

Binary compatibility is a serious but often underestimated issue in modern programming languages. Current mainstream programming languages neither specify nor provide tools to solve the problem. In this paper we discussed the problem in details and suggested a set of rule to check avoiding inconsistencies between binary components. The Welltype experimental programming language is defined to avoid various traps and pitfalls of the current mainstream languages. One of the improvements of Welltype is the application of the dynamic loader with the capability to detect the possible binary incompatible modules. We implemented a prototype tool-chain of Welltype. Practical experiments show that the rules detecting the binary incompatibility in Welltype are strict enough to filter out critical issues, but still allow maintenance of evolving individual subsystems as binary components.

References

[1] ISO, "ISO/IEC 9899:TC3 – committee draft of the C99 standard – section 5.1.1.1." `http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf`.

[2] J. Bloch, *Effective Java: A Programming Language Guide. The Java Series (2nd ed.)*. Addison-Wesley, 2008.

[3] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[4] B. Stroustrup, *The C++ programming language, 4th Edition*. Addison-Wesley, 2013.

[5] S. Chamberlain and I. L. Taylor, "Using LD the GNU linker," 2010. `http://lib.hpu.edu.vn/handle/123456789/21416`.

[6] "LLD – the LLVM linker," 2018. `https://lld.llvm.org/`.

[7] M. Franz, "Dynamic linking of software components," *Computer*, vol. 30, no. 3, pp. 74–81, 1997.

[8] S. Drossopoulou, G. Lagorio, and S. Eisenbach, "Flexible models for dynamic linking," in *European Symposium on Programming*, pp. 38–53, Springer, 2003.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[10] "LD VERSION command." `https://sourceware.org/binutils/docs/ld/VERSION.html`.

[11] "ELF symbol versioning with glibc 2.1 and later," 1999. `https://lists.debian.org/lsb-spec/1999/12/msg00017.html`.

[12] M. Stevanovic, *Dynamic Libraries Versioning*, pp. 187–231. Berkeley, CA: Apress, 2014. `https://doi.org/10.1007/978-1-4302-6668-6_10`.

[13] Tool Interface Standard, *Executable and Linking Format (ELF) Specification, Version 1.2*, May 1995. `http://refspecs.linuxbase.org/elf/elf.pdf`.

[14] Santa Cruz Operation, *System V Application Binary Interface*, March 1997. `http://www.sco.com/developers/devspecs/gabi41.pdf`.

[15] Oracle, *Java Language Specification, Chapter 13. Binary Compatibility*, 2018. `https://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html`.

[16] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pp. 64–73, IEEE, 2014. https://doi.org/10.1109/CSMR-WCRE.2014.6747226.

[17] I. Savga, M. Rudolf, and S. Goetz, "Comeback!: a refactoring-based tool for binary-compatible framework upgrade," in *Companion of the 30th international conference on Software engineering*, pp. 941–942, ACM, 2008. https://doi.org/10.1145/1370175.1370198.

[18] K. Atkinson, M. Flatt, and G. Lindstrom, "ABI compatibility through a customizable language," in *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010* (E. Visser and J. Järvi, eds.), pp. 147–156, ACM, 2010. http://doi.acm.org/10.1145/1868294.1868316.

[19] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[20] A. Koenig, "The nightmare of binary compatibility," in *Dr.Dobb's*, 2014. http://www.drdobbs.com/cpp/the-nightmare-of-binary-compatibility/240166914.

[21] "How to design a C++ API for binary compatible extensibility," 2010. https://stackoverflow.com/questions/1774911/how-to-design-a-c-api-for-binary-compatible-extensibility.

[22] Microsoft, "C++ binary compatibility between Visual Studio 2015 and Visual Studio 2017," 2017. https://docs.microsoft.com/en-us/cpp/porting/binary-compat-2015-2017.

[23] M. Pietrek, "Peering inside the PE: a tour of the win32 (R) portable executable file format," *Microsoft Systems Journal-US Edition*, pp. 15–38, 1994.

[24] Á. Baráth and Z. Porkoláb, "Welltype: Language elements for multiparadigm programming," in *Position Papers of the 2017 Federated Conference on Computer Science and Information Systems*, pp. 91–101, 2017. http://dx.doi.org/10.15439/2017F546.

[25] Á. Baráth, "Welltype legacy web page." http://baratharon.web.elte.hu/welltype, 2014.

[26] Á. Baráth, "Welltype project web page." http://repo.hu/projects/welltype, 2018.

[27] ISO/IEC, "ISO/IEC 14882:2003(E): Programming Languages - C++ §3.2 One definition rule [basic.def.odr]," 2003. http://eel.is/c++draft/basic.def.odr.

Department of Programming Languages and Compilers, Eötvös Loránd University

Email address: [baratharon,gsd]@caesar.elte.hu

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 1, 2018
DOI: 10.24193/subbi.2018.1.05

TOWARDS GREEN COMPUTING IN ERLANG

ÁRON ATTILA MÉSZÁROS, GERGELY NAGY, ISTVÁN BOZÓ,
AND MELINDA TÓTH

Abstract. Energy efficiency in computing was identified as low energy usage of the hardware for a while. However, nowadays, we can talk about energy efficiency in terms of software as well. Therefore, we have to investigate how the different design decisions and programming language constructs affect the energy consumption. The green computing is a relatively new research area, guidelines are required for the software developers in terms of energy efficiency. In our research we are focusing on the functional programming language Erlang. We have investigated the effect of different language constructs (such as higher order functions), parallelism, data structures and styles of programming on energy usage. Additionally we present a tool to measure and visualise the consumed energy.

1. Introduction

Environment friendly tools and devices are needed in every area of manufacturing, thus it is crucial to have computing devices with energy usage as low as possible. Therefore, we have to take into account the amount of energy used by a certain devices (i.e. a PC) when running a software.

In the field of green computing we are investigating the energy usage of a software, this means the amount of energy used by the hardware when running the software.

Researches have been already presented on energy efficient computing (see Section 2), however most of them are focusing on mainstream languages. The goal of our research is to investigate the energy usage of Erlang [1] programs.

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68N18.

1998 *CR Categories and Descriptors.* F.2.m [**ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY**]: Miscellaneous – *Energy usage*; F.3.22 [**SOFTWARE ENGINEERING**]: Miscellaneous – *Green computing* .

Key words and phrases. green computing, energy efficiency, Erlang, RAPL.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

Erlang is a widely used functional programming language, designed for building concurrent/distributed soft-real time applications. Since Erlang is functional language and the main building blocks of the language are the functions, we have created a tool (see in Section 3) to measure and visualise the energy consumption of Erlang functions. The tool is based on the Intel provided *RAPL* [2] tool and the *Rapl-read* [3] program.

In this paper we present the measurements on some key elements of the language, such as the usage of lists, higher order functions and parallelisation as well (in Section 4). We demonstrate our finding on different algorithms.

The ultimate goal of our research is to extend the static source code analysis and transformation framework RefactorErl [4, 5]. We would like to define static analyses to find those source code fragments that are presumably more energy-intensive than other equivalent solutions. We also want to define a set of refactorings that can be applied, either automatically or semi-automatically, to reduce the energy used by the Erlang programs.

In this paper we are presenting our first findings on the energy usage on Erlang programs, and the latter mentioned goal is the target of our future work.

2. Related work

In recent years there have been lots of studies on the topic of energy-efficiency. Many of these works studied the energy consumption of imperative languages, for example in Java 6.2% energy savings have been obtained [6] and thread management constructs also have been studied [7]. There also have been studies regarding the power consumption of CMOS digital circuits [8], power analysis of embedded software [9], how code obfuscation affects energy usage [10] and finally the impact of commonly used refactorings have also been studied [11].

Large amount of researches were carried out for imperative languages, but green computing is just as important in the area of functional languages. Lima et al. [12] analysed the energy behaviour of Haskell. They presented tools for testing the energy footprint of a program and also showed that some constructs can be beneficial in some situations, while in others they may not be a good choice. They collected energy consumption data using Running Average Power Limit (RAPL [2]) and accessing the data through model-specific registers (MSRs).

In the case of Erlang only a few research has been done regarding green computing. Ortiz [13] wrote her MSc thesis on the topic of green computing in Erlang and showed how some steps of refactoring and different data structures affect energy consumption in the case of Fibonacci and Karatsuba algorithms. Varjão [14] gave a talk on the Erlang Factory SF conference in March 2017, where he presented a tool for measuring the energy consumed by Erlang functions. His tool is based on RAPL, but it has not been made publicly available yet, therefore we could not use it.

3. Measuring and visualising energy usage

In this section we present the tools used to measure the energy usage of Erlang functions. We provide an overview of the workflow of all our different program components working together in order to make measuring energy consumption more convenient.

3.1. **RAPL.**

Running Average Power Limit (RAPL) is a tool created by Intel, as part of their power-capping interface. It is available under Linux operating system, on CPUs that have at least Sandy Bridge or newer architecture. The onboard power meter have been introduced in the Sandy Bridge microarchitecture. This provides information on power meters and power limits of the CPU, and exports power information through a set of Model-specific Registers (MSRs).

RAPL provides counters to get energy and power consumption information. RAPL is not an analog power meter, but an accurate software power model, that estimates usage by using hardware performance counters and I/O models [2].

In RAPL, platforms are divided into domains, in order to get more detailed information on the energy consumption. The domains are:

- PKG - The entire package
  - PP0 - Only the cores
  - PP1 - The uncore part of the package
- DRAM - Main memory

Among the listed domains the following inequality always holds: PP0 + PP1 $\leq$ PKG and DRAM is independent of the other three [15].

3.2. **Rapl_read.**

There are three methods to extract power and energy consumption data from RAPL.

- `sysfs` - Reads files from /sys/class/powercap/intel-rapl/intel-rapl:0 using the powercap interface. Requires at least Linux 3.13 with no special permissions.
- `perf_event` - Uses the perf_event interface. Requires at least Linux 3.14, and root access or the /proc/sys/kernel/perf_event_paranoid value to be less than 1.
- `msr` - Reads data directly from the MSRs under /dev/msr, and requires root privileges.

It is important to note that all methods provide readings for an entire CPU socket, there is no way to get readings for individual cores and processes this way.

We used a program written in C called rapl-read [3], that provides an interface for all three methods. We had to slightly modify it to support multiple sockets and to send and receive signals to and from our Erlang program. We also had to split all measuring functions to a pre and post versions, so that we can read the data before and after running an Erlang function, and thus obtaining the energy consumption.

3.3. **Erlang server.** It is important for the accuracy of measurement to get the readings as close to the beginning and end of a function as possible. Because of this, we needed the Erlang program (`energy_consumption.erl`) and the `rapl-read.c` program to communicate. This is accomplished using ports in Erlang and using the `read()` and `write()` functions in C. The process of communication can be seen on Figure 1.

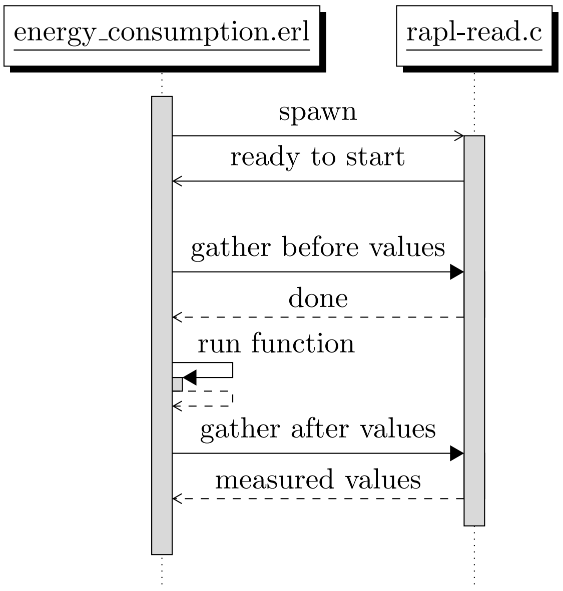


FIGURE 1. Sequence diagram of `energy_consumption.erll` and `rapl-read.c` communicating throughout one measurement

When the measured values have been received by the Erlang component from `rapl-read.c`, it saves it using a `dets` (disk-based term storage) table.

The results are stored as tuples in the following format:

$$\{\{\{Module, Function, InputSize\}, Method, Domain\}, Value\}$$

where $Value$ is the measured value, $Method$ is one of the three modes to access RAPL, and $Domain$ is one of the four available RAPL domains. $Module$ and $Function$ are the module and name of the measured function, while $InputSize$ is a value provided by the user describing the size of the test arguments. If no such value is provided we take the head of the argument list and use it as $InputSize$. This is useful when the argument is a single number.

We used the following methodology to measure the energy consumption. Each time all three methods and all available domains are measured and sent to the Erlang program. In addition to this we also measure the run-time of the function and store it with the energy usage values.

This process is repeated N times, where N is a parameter given by the user. After this the Erlang program reads back all the data and for each method-domain pair calculates the average energy consumption, disregarding the lowest and highest values. This average is then inserted to the `dets` table and also printed to a text file, thus we can easily plot the data.

To make measurement processing easier we created a function that performs all of the above described functionalities. This function takes six arguments, that in addition to the ones mentioned above, are the executable file compiled from `rapl-read.c`, and the names of the output files:

```
measure(Program, {Module, Function, Attributes, InputSize},
        N, ResultOutput, AvgOutput, LogFile)
        %InputSize is optional
```

3.4. **Visualisation.** As we have mentioned earlier, the measured results were exported to text files. These files contain one result in each line in the following structure:

Module Function InputSize Method Domain Value

Above the arguments the output contains the method used in measurement, the referenced domain and the measured value.

We use the same framework for measuring the run-time of the functions. The output in this case includes the same fields, but when measuring the execution time, the method and domain are irrelevant, thus to preserve the same structure, these fields are containing the measured time as well.

To visualise the measured data, it would had taken a lot of time to analyse manually these data files (for example in a spreadsheet). We decided to process the raw data with a Python script.

The script that processes the raw data has the following stages:

(1) Grouping the measured values by the functions. A function can be uniquely identified by the name of the module and the name of the function.
(2) Extracting the values separately by the different methods and domains.
(3) Draw the figures using *matplotlib* [16]: the Y-axis shows the energy consumption in Joule and the X-axis shows the different input size values.
(4) Optionally, there is a feature for export the required values to another data file and generate a latex file from it that contains only the diagram.

The steps are fully customisable to help analyse the data more precisely. We used command line arguments to give the exact specification of the diagrams. The following flags are available:

- –files: The script can take multiple files, all the contained data is processed. This gives flexibility to visualise different functions. For example, we stored all the different functions in distinct files, so we could compare them as we wanted.
- –methods: Specify which methods to display. It can take the three methods and time, and also several methods at once. In this case all of them will be drawn.
- –domains: Specify which domain to display.
- –output: Optional: the name of the output data file and .tex file.
- –logscale: This flag changes the Y-axis to a logarithmic scale.

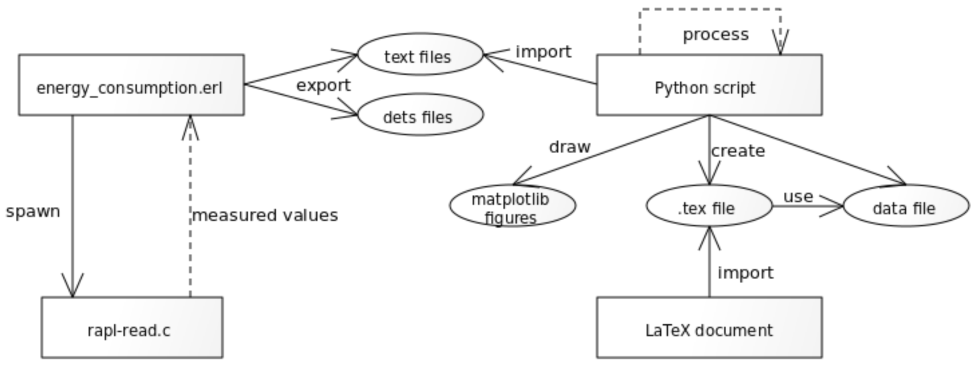


FIGURE 2. Process from measurement to visualisation

4. Energy usage in Erlang

We used two different problems and many implementations for them to gain information on the energy consumption. We used different language constructs and data structures in the implementations. The main aspects for our implementations were the following:

- Using different data structures:
  - Lists
  - Extendible arrays
  - Fix-sized arrays
- Using or avoiding higher order functions (HOFs)
- Parallel or sequential implementations

We also paid attention to the run-time of each implementation, in order to gain a better understanding of the effects of language constructs on energy consumption. The measurements were made on a system with Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz and 8 GB of DDR3 RAM @ 1600MHz, using Ubuntu 16.04 LTS. All plotted data was measured using MSR method and PKG + DRAM domains.

4.1. Goals.

Our goal was to find any relation between language constructs, different data structures and energy usage. We intended to find out if extendible arrays, fix-sized arrays or lists are more efficient. We also wanted to, among other things, gain information on the effects of higher-order functions on energy consumption. Another thing we wanted to find out was the effect of parallelising on energy consumption. In the followings we are demonstrating different implementations of two well-known algorithms (placing queens on a chessboard and sparse matrix multiplication) and the energy used when evaluating the different implementations.

4.2. N-queens.

Problem. Place N queens on an N×N chessboard, so that no two queens attack each other.

Solutions. For this problem we measured the following five implementations:
Lists with HOFs (`queens_lists`): This version uses the higher-order functions `lists:flatmap/2` and `lists:all/2` to get the results.

```erlang
attacks({RowA, ColA}, {RowB, ColB}) ->
        RowA == RowB orelse ColA == ColB
        orelse abs(RowA - RowB) == abs(ColA - ColB).
legal_list(Queen, Queens) ->
        lists:all(fun(Q) -> not (attacks(Queen, Q)) end, Queens).
solve_list(N, Row, Queens) when Row > N -> [Queens];
solve_list(N, Row, Queens) ->
```

```
  lists:flatmap(
        fun(Qs) -> solve_list(N,Row+1,Qs) end,
        [ [{Col,Row} | Queens] ||
        Col <- lists:seq(1,N),legal_list({Col,Row},Queens) ]
  ).
queens_list(N) when N > 0 -> solve_list(N,1,[]).
```

Lists without HOFs (**queens_nohof**): The same as the previous one, but instead of `lists:flatmap/2` and `lists:all/2` it uses a custom implementation for these functionalities. These functions do not need function parameters, because we have hardcoded this information into them.

```
all_nohof(_,[]) -> true;
all_nohof(Queen,[Q|Queens]) ->
        G = attacks(Queen,Q),
        if G -> false;
        true -> all_nohof(Queen,Queens) end.
flatmap_nohof([],R,_,_) -> R;
flatmap_nohof([H|T],R,N,Row) ->
        L = solve_nohof(N,Row+1,H),
        P = concat_to(L,R),
        flatmap_nohof(T,P,N,Row).
concat_to([],R) -> R;
concat_to([H|T],R) -> concat_to(T,[H|R]).
```

Extendible arrays (**queens_array**): This version uses the same algorithm as the one using lists, but instead uses arrays created with `array:new()`. No HOFs are used in this implementation.
Fix-sized arrays (**queens_array_fix**): This is the same as the one with extendible arrays, but instead of `array:new()`, we create fix-sized arrays with `array:new(Size)`.
Parallel version (**queens_par**): This version uses a parallel map (`par_map/2`) instead of map. The underlying data structure is a list.

```
par_map(F, Xs) ->
        Me = self(),
        [spawn(fun() -> Me ! F(X) end) || X<-Xs],
        [receive Res -> Res end || _ <- Xs].
```

Each implementation was measured for inputs from 6 through 12.
Results. The energy consumption and run-time of these implementations can be seen on Figure 3. We can see that in the case of the sequential versions, the slower program consumes more energy, as expected. In the parallel case, even though the parallel version is not always the slowest, it clearly consumes the most energy. The reason for this may be that this parallel version just replaced `map/2` with `par_map/2`, not paying attention to the underlying data structure,

but in the case of Erlang all data sent between processes is copied [17]. Because the items mapped are lists, lots of data is copied each time `par_map/2` is called, which is slow, and thus consumes lots of energy.

Both implementations with arrays performed the same for smaller inputs, but for larger inputs the fix-sized array version consumed more energy.

It is clear that the most efficient versions were the ones using lists. It seems like eliminating the higher order functions from the implementation improved energy consumption, and also made the program faster.

In conclusion, we saw that eliminating HOFs, such as map can improve energy consumption. We also saw, that parallelising can be dangerous, we have to pay attention to what data is sent. Arrays seem to be performing worse than lists, but are still more efficient than our naive parallel algorithm.

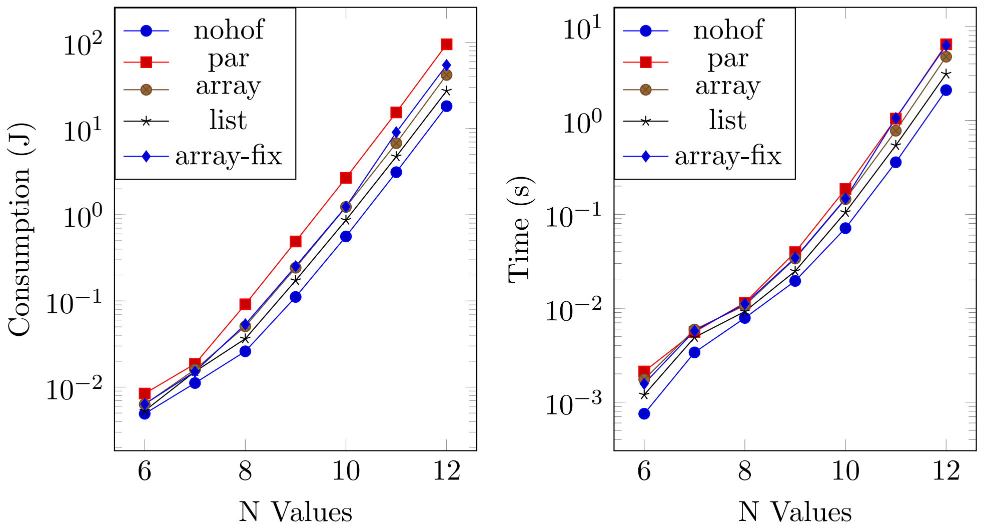


FIGURE 3. Energy consumption and run-time of all N-queens implementations.

4.3. **Sparse matrix multiplication.**

Problem. Multiply two matrices, whose elements are mostly zeros.

Solutions. All implementations solve the problem by reducing it to a series of matrix-vector multiplication, and then to vector-vector multiplication. For this problem we measured the following implementations:

Lists (`mxm_lists`): One of the matrices is represented as list of tuples, whose first element is the row number, the second element is also a list of tuples, whose first element is the column number and the second is the value of the matrix element: $[\{Row, [\{Col, Value\}]\}]$. The other matrix is represented the same way, but with row and column values swapped, in order to make multiplying them easy.

```
vxv_list(Row,Col) -> vxv_acc_list(Row,Col,0).
vxv_acc_list([],_,Acc) -> Acc;
vxv_acc_list(_,[],Acc) -> Acc;
vxv_acc_list([{I,R}|Row],[{I,C}|Col],Acc) ->
        vxv_acc_list(Row,Col,Acc+R*C);
vxv_acc_list([{I,R}|Row],[{J,C}|Col],Acc) ->
        if I < J -> vxv_acc_list(Row,[{J,C}|Col],Acc);
                true  -> vxv_acc_list([{I,R}|Row],Col,Acc)
        end.
mxv_list( Rows, Col ) ->
        Product = [ {I,vxv_list(Row,Col)} || {I,Row} <- Rows ],
        filter( fun({_,V}) -> V /= 0 end, Product ).
mxm_list(Rows, Cols) ->
        Product = [{I,mxv_list(Rows,Col)} || {I,Col} <- Cols],
        filter( fun({_,V}) -> V /= [] end, Product).
```

Lists without HOFs (`mxm_nohof`): In this version we replaced all occurrences of the filter function with our own implementation, that does not need a function as its argument. For example the one in `mxv_list` was replaced by `filter_zeros`:

```
filter_zeros([],R) -> lists:reverse(R);
filter_zeros([{_,0}|P],R) -> filter_zeros(P,R);
filter_zeros([H|P],R) -> filter_zeros(P,[H|R]).
```

Parallel (`mxm_par`): The representation of the matrices is the same as in the version with lists, but the matrix-vector and vector-vector multiplications are executed in parallel, using spawn in the list comprehension. Receiving the data sent back by the processes is done in another list comprehension, with each process sending back its ID too, in order to find the proper place for the result in the matrix.

Parallel with process pools: We also implemented a parallel version with a parallel ordered map implementation, that uses process pools to limit the number of processes spawned. First, we replaced both instances of the list comprehensions in the parallel code, but this way if we limit the process pool to 20 processes, then because of the recursion in our code $20^2 = 400$ processes were spawned. An easy way to fix this was to only parallelise the outer matrix-vector multiplications and use the sequential program for solving vector-vector multiplications. These versions were named, respectively, `mxm_ppool` and `mxm_parseq`.

Array (`mxm_array`): In this version the matrices are represented as arrays of arrays, where for one matrix the first index determines the row and the second the column, and for the other matrix it is the other way around. Zero elements of the matrix are left undefined. We used `array:sparse_map/2` and `array:sparse_foldr/3` to replace the list comprehensions and recursion on lists, so for example the vector-vector multiplication became the following code:

```
vxv_array(Row,Col) ->
A = array:sparse_foldr(fun(_,Val,Acc)->Acc + Val end,
  0, array:sparse_map(fun(Index,Elem) ->
    C = array:get(Index,Col),
    if C == undefined -> undefined;
       true -> Elem*C
    end
  end,Row)),
if A == 0 -> undefined;
   true -> A
end.
```

The implementation does not depend on whether we use extendible or fix-sized arrays, so no separate versions were made for these, but when measuring we used both types of arrays.
Array without HOFs (`mxm_array_nohof`): The same way we eliminated higher order functions from the list version, we also eliminated HOFs (`sparse_map` and `sparse_foldr`) in the version using arrays. The `sparse_map` of the previous `vxv_array` function became the following non higher order function:

```
vxv_array_map(Index,Size,_,Row) when Index == Size -> Row;
vxv_array_map(Index,Size,Col,Row) ->
ElemR = array:get(Index,Row),
ElemC = array:get(Index,Col),
if ElemR == undefined -> vxv_array_map(Index+1,Size,Col,Row);
   ElemC == undefined -> vxv_array_map(Index+1,Size,Col,
    array:set(Index, undefined, Row));
   true -> vxv_array_map(Index+1,Size,Col,
    array:set(Index, ElemC*ElemR, Row))
end.
```

Results. Even though our implementations can handle non-square matrices, for ease of distinguishing between the sizes of test cases we only used square matrices. Test matrices were generated randomly, in sizes from 10×10 up to 400×400. For each size we generated three test cases with different ratio of non-zero elements. These ratios were 1%, 10% and 30%.

The array implementations were tested with both extendible and fix-sized arrays. The measured energy consumption values and run-times are shown on

Figure 4. Fix-sized array versions are not shown as they were not different from extendible array values in any meaningful way.

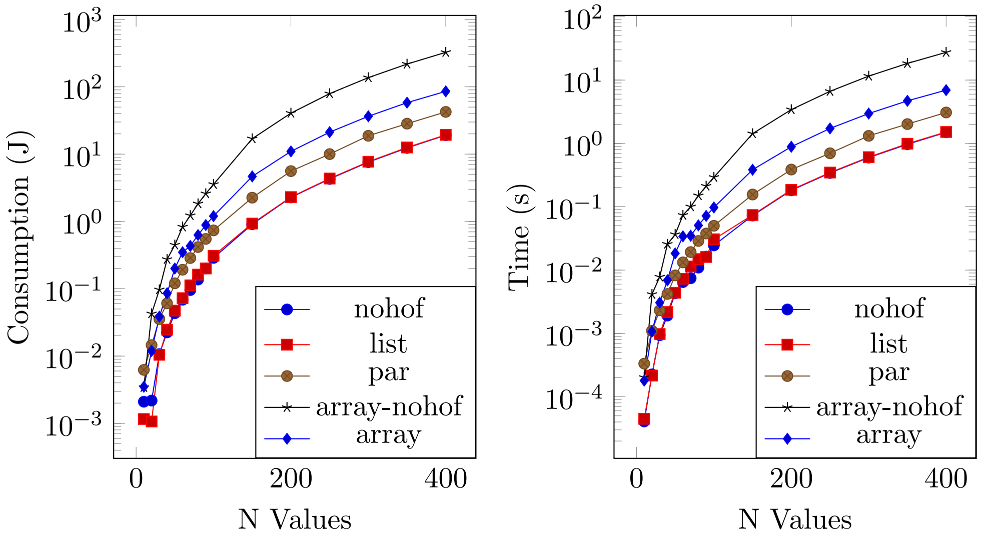


FIGURE 4. Energy consumption and run-time of sparse matrix multiplication implementations.

In all cases we see correlation between run-time and energy consumption, as expected. We can see that the versions using lists performed almost the same. The one without HOFs is in almost all cases slightly better than the one using filter as can be seen in Table 1. It can also be seen that all implementations using arrays performed worse than lists. One reason for that might be that in the case of arrays, the matrices were stored directly in the array, not as pairs of indices and values (as in the case of lists), thus resulting in lots of undefined elements in the arrays and increasing the size of stored data. Contrary to the versions using lists, in the case of arrays the one containing HOFs performed better than the one without HOFs. The reason for this might be that we do not know how arrays are implemented, and thus we cannot implement the non higher order versions of `sparse_map` and `sparse_foldr` as efficiently.

The parallel version performed worse than the sequential ones, probably because of the same reasons as mentioned before in the case of the N-queens problem. The effect of using process pools can be seen on Figure 5. We can see that spawning too many processes is really bad for power consumption. The basic parallel version and the process pool version with the number of processes limited to 4 and 20 processes (thus only creating 16 and 400 processes at a

	50	100	150	200	250	300	350	400
mxm_nohof	0.043	0.287	0.909	2.277	4.278	7.599	12.374	19.120
mxm_list	0.047	0.309	0.930	2.300	4.340	7.711	12.556	19.315

TABLE 1. Energy consumption of the given functions for different input sizes, measured in Joules, with 30% non-zero elements

time) perform almost the same. The reason for this might be that even though the process pool version uses fewer processes, it also sends more messages. The most effective parallel version was the half parallel, half sequential process pool version, limited to 20 processes. This shows that it can be worth it to limit the number of processes. These results require further analysis in order to find the true connection between the number of processes, messages and energy consumption.

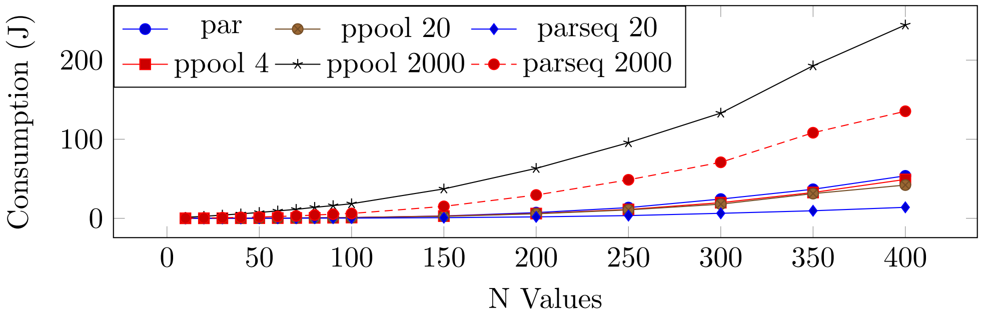


FIGURE 5. Energy consumption of different parallel implementations of sparse matrix multiplication. The numbers after the function names denote the number of processes in the process pool (note that in the case of pool, this number has to be squared to get the actual number of processes). These measurements were made using 12 cores on a system with Intel(R) Core(TM) i7-8700K CPU and 16 GB of DDR4 RAM, using Ubuntu 17.04.

On all figures for this problem the inputs with 30% non-zero elements are plotted.

5. EVALUATION

In both cases we have found that naively parallelising made the algorithm consume more energy. This is probably because we did not use any special strategy to parallelise the algorithms, we simply replaced map with parallel map and list comprehension with list comprehensions that spawn processes. Even though spawning processes has a relatively low cost, since we spawned so many of them, in the case of N-queens sometimes more than 500 000, it may have increased energy consumption and run-time. Another problem may have been, that in Erlang data sent to a process is copied [17], so our program made a copy of large lists each time a process was spawned.

We have seen that using process pools to reduce the number of processes can be beneficial, but we have to choose the maximum number of processes wisely, because it greatly affected energy consumption.

We also observed, that in most cases eliminating higher-order functions improves energy consumption. This is most visible in the case of the N-queens problem. It can also be seen in the case of the sparse matrix multiplication problem, but to a lesser extent. In that algorithm the HOF filter does not take up that much part of the whole solution, so it contributes less to overall energy consumption. While in the N-queens algorithm map is used as the base of the algorithm, so it contributes much more to the energy consumed. An outlier to this rule is the array version of sparse matrix multiplication, where eliminating HOFs made the algorithm much worse. This might be because in the version using HOFs we did not use the traditional `map` and `foldr`, but instead the sparse version of them, which may be implemented much more efficiently than our own implementations.

The third thing we noticed was that in both cases arrays performed worse than lists. That may be because in the case of N-queens our algorithm was first developed for lists and then adapted to arrays. In the case of the matrix multiplication the reason may be that we stored data in a completely different way in arrays than in lists. From the results it seems like the array version is not efficient in storing sparse matrices. Even though our observation was that arrays are not as efficient as lists, there are cases, for example calculating Fibonacci numbers, where they may perform better [13].

6. Conclusion and future work

We wanted to measure the energy consumption of Erlang programs and discover patterns and relations between language constructs and power consumption. We used RAPL to measure energy consumption, and created an framework to measure and store energy consumption values.

After measuring several implementations of the N-queens problem and the sparse matrix multiplication we found that eliminating higher order functions may make the program more efficient. In our cases we also found that using arrays instead of lists was not a good idea. Parallelising the solutions can make energy usage worse, because it spawns lots of processes, but this could be solved using process pools.

In the future we would like to investigate different parallelisation techniques and we would like to further examine the effect of limiting the number of processes and messages sent on energy consumption. We also would like to measure the effect of the number of cores used when running the parallel program. Additionally, we would like to confirm our current findings using different algorithms, such as the N-body problem.

Finally, we would like to create a tool as part of RefactorErl, that automates the process of finding patterns that could be refactored into more energy efficient version and then helps transform the code into a more energy aware version.

References

[1] Joe Armstrong. *Programming Erlang.* The Pragmatic Bookshelf, 2nd edition, October 2013. ISBN 978-1-93778-553-6.

[2] Srinivas Pandruvada. Running Average Power Limit - RAPL. https://01.org/blogs/2014/running-average-power-limit---rapl. [Accessed: 03.10.2018.].

[3] Vincent M. Weaver. Reading RAPL energy measurements from Linux. http://web.eece.maine.edu/~vweaver/projects/rapl/. [Accessed: 03.10.2018.].

[4] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. RefactorErl, Source Code Analysis and Refactoring in Erlang. In *Proceeding of the 12th Symposium on Programming Languages and Software Tools*, Tallin, Estonia, 2011.

[5] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. Modeling Semantic Knowledge in Erlang for Refactoring. In *Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, volume 54(2009) Sp. Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, July 2009.

[6] Rui Pereira, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. The Influence of the Java Collection Framework on Overall Energy Consumption. *CoRR*, abs/1602.00984, 2016. URL `http://arxiv.org/abs/1602.00984`.

[7] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding Energy Behaviors of Thread Management Constructs. *SIGPLAN Not.*, 49(10):345–360, October 2014. ISSN 0362-1340. doi:

10.1145/2714064.2660235.

[8] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, Apr 1992. ISSN 0018-9200. doi: 10.1109/4.126534.

[9] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, Dec 1994. ISSN 1063-8210. doi: 10.1109/92.335012.

[10] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause. How Does Code Obfuscation Impact Energy Usage? In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 131–140, Sept 2014. doi: 10.1109/ICSME.2014.35.

[11] Cagri Sahin, Lori Pollock, and James Clause. How Do Code Refactorings Affect Energy Usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 36:1–36:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2774-9. doi: 10.1145/2652524.2652538.

[12] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 517–528, March 2016. doi: 10.1109/SANER.2016.85.

[13] Jessica Tatiana Carrasco Ortiz. Green computing in Erlang, 2017.

[14] Filipe Varjão. Measuring Erlang energy consumption, and why this matters. http://www.erlang-factory.com/sfbay2017/filipe-varjao.html. [Accessed: 03.10.2018.].

[15] Wander Lairson Costa. Power profiling overview. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Power_profiling_overview, [Accessed: 03.10.2018.].

[16] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[17] Ericsson. Erlang Efficiency Guide, Processes. http://erlang.org/doc/efficiency_guide/processes.html. [Accessed: 03.10.2018.].

ELTE, Eötvös Loránd University, Pázmány Péter sétany 1/C, Budapest, Hungary, 1117

Email address: {`archy, nagygeri97, bozoistvan, tothmelinda`}`@caesar.elte.hu`

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 1, 2018
DOI: 10.24193/subbi.2018.1.06

NOSQL DATABASE PERFORMANCE BENCHMARKING - A CASE STUDY

CAMELIA-FLORINA ANDOR AND BAZIL PÂRV

ABSTRACT. This paper describes an experimental study regarding NoSQL database performance. Two NoSQL databases were considered (MongoDB and Cassandra), two different workloads (update heavy and mostly read) and several degrees of parallelism. The results refer to throughput performance metric. Statistical analysis performed is referring to throughput results. Conclusions confirm that MongoDB performs better than Cassandra in the context of a mostly read workload, while Cassandra outperformed MongoDB in the context of an update heavy workload where the number of operations was high.

1. INTRODUCTION

It is hard to figure out what kind of database fits best a certain application nowadays. There are many NoSQL databases that are highly configurable and flexible, but to determine the right choice for a given application is a tedious task. NoSQL databases differ from one another on many levels, from data model to distribution model and it is not easy to make a fair performance comparison between them. Just reading the documentation of a certain NoSQL database is not enough to make sure you make the right decision for your application, but performance benchmarking gives you the opportunity to see that database in action, on your chosen hardware configuration.
In order to find out how NoSQL databases perform on a general performance benchmark, we ran performance benchmarking tests using the `YCSB` client against Cassandra and MongoDB database servers. We generated a dataset that fits in memory using the `YCSB` client and then ran benchmarking tests

Received by the editors: February 9, 2018.
2010 *Mathematics Subject Classification.* 68P15, 68P99.
1998 *CR Categories and Descriptors.* H.2.1 [**Database Management**]: Logical design – *Data models*; H.2.4 [**Database Management**]: Systems – *Distributed databases, Parallel databases*.
Key words and phrases. NoSQL database, performance benchmarking, MongoDB, Cassandra.

using various combinations of workload, number of operations and number of client threads on each database server.

2. Background

2.1. **NoSQL models.** NoSQL models appeared as a response for the need of big companies like Google or Amazon to store and manage huge amounts of data. The fact that the relational model was not built to offer horizontal scalability and the difference between in-memory data structures that are used in application programming and the relational model, known as impedance mismatch[16] are the key factors that contributed to the emergence of the NoSQL databases.

There are four main NoSQL data models: key-value, document, column-family and graph. For this paper, two data models were chosen: the document model and the column-family model. The document model and the column-family model are based on the key-value model. In a key-value database, data is stored as key-value pairs, with the key part of the pair as the unique identifier for the value that is stored in the value part of the pair. Complex values like objects or arrays can be stored as values for keys, but their structure remains invisible to the database.

In a document database, data is stored as documents. A document is similar to a key-value pair, but the difference is that the value part has a structure that is visible to the database. In a key-value database, the value part is not visible to the database. What used to be a record in a relational table becomes a document in a collection inside a document database. Still, there are some key differences between them. In a relational table, all the records have the same schema and every field can store only simple values. In a document collection, documents can have different schemas and complex values like arrays or embedded documents can be stored as values for a given field. The most popular document format is JSON[12], but there are others, like XML[21] or YAML[22]. Document databases simplify application development process. It is a lot easier to store objects as documents than to create the relational representation of an object and store it in more than one table. Also, document databases have flexible schema, so it is easier to modify it as the application evolves.

In column-family databases, data is stored as rows in column families. A column family is similar to a relational table, but it has a flexible schema. Rows in the same column family can have different columns. Each column is composed of a timestamp and a key-value pair, with the name of the column as key and the value for that column as value. Complex values like collections or arrays can be stored as values for a given column. Column-family databases are generally optimized for writes. When designing applications for column-family databases, it is a good practice to know in advance what kind of queries are needed in order to optimize read operations.

2.2. **NoSQL tools.** For each selected NoSQL model, a NoSQL database was chosen. In our benchmark, the document model is represented by MongoDB[14]. There are other document databases available on the market, like CouchDB[5] or OrientDB[15]. The column-family model is represented in our benchmark by Apache Cassandra[2]. Other column-family databases are Bigtable[3] and HBase[10].
MongoDB is an open source distributed database that was built to offer schema flexibility, horizontal scalability, replication and high availability. It has a rich query language and a good support for ad hoc queries. It was developed by 10Gen, known today as MongoDB Inc. In a MongoDB cluster there are shards or nodes that store data, config servers that store cluster metadata and query routers that route queries to the shards.
Cassandra is an open source distributed database that offers high availability, horizontal scalability and data replication, including multiple datacenter replication. It was initially developed at Facebook[13] and its data model was based on Bigtable and Dynamo[7]. In a Cassandra cluster every node is identical, which means that there is no master node and nodes can be added or removed from the cluster with no downtime.
Cassandra 3.11.0 and MongoDB 3.4.4 were the database versions installed on our servers.

2.3. **NoSQL benchmarking.** Benchmarking is very useful when evaluating NoSQL systems because it reveals the actual performance of a database on a given hardware configuration for a specific application use case. It is a difficult task to make a comparison between different NoSQL databases, and the lack of benchmarking tools for this category makes this task even harder. As a consequence of this fact, `Yahoo! Cloud Serving Benchmark`[4] (or `YCSB` for short) emerged as an open source benchmarking framework for cloud or NoSQL systems. `YCSB` was written in Java and it has two main components: the `YCSB` client, which is a workload generator and the Core workloads that represent a set of workload scenarios to be executed by the generator[23]. Both components are extensible. New workloads can be defined, so that specific application workloads can be run and database performance for those workloads can be evaluated. Other benchmarking tools are `cassandra-stress tool`[19], a tool for benchmarking Cassandra clusters and `cbc-pillowfight`[18], a tool for benchmarking Couchbase. In [6], `cbc-pillowfight` was used as a tool for workload generation, while the benchmarked database was MongoDB. Also, in a more general context, we can mention `BigBench`[1] tool. These tools were not used in our evaluation because (a) they cannot be used for all databases considered, and/or (b) we cannot find straight Windows implementations for them. There are benchmarking studies using YCSB discussed in the literature: [9], [8] and [11]. All these studies use a different testing environment, more precisely they employ a cloud-based infrastructure. By using virtual machines, cloud solutions are easier to manage because all the resources needed are available as Software-as-a-Service or Infrastructure-as-a-Service. Our solution, discussed in the next section, implied a big amount of work for installing and configuring all software applications needed. For our performance benchmark, we chose to use `YCSB` version 0.12.0 as benchmarking framework, because it is free, available, and can be used for evaluating Cassandra and MongoDB.

3. Case study

In database performance benchmarking, there are two important metrics: the throughput, measured in operations per second and the latency, measured in microseconds per operation. These two metrics are present in every test output we obtained using `YCSB`, but from lack of space only the throughput was analyzed in this paper.

3.1. **Experimental setting.** A total of three servers having the same hardware configuration were used to run the experiment. The `YCSB` client ran on the first server, Apache Cassandra ran on the second server and MongoDB ran on the third server. Each server had the following hardware configuration:

- OS: Windows 7 Professional 64-bit
- CPU: Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 4 cores, 8 logical processors
- RAM: 16 GB
- HDD: 500 GB.

The data set used in our tests is composed of 4 million records and it was generated by the `YCSB` client. Every record has 10 fields and each field contains a random generated 100 byte string value. Because of its size, this data set could fit within memory entirely. Two `YCSB` core workloads were chosen: Workload A (50% update, 50% read), an update heavy workload[20] and Workload B (5% update, 95% read), a read mostly workload[20]. The number of operations parameter is in fact the number of operations performed in a test run. Each workload was tested with the following values for the number of operations: 1000, 10000, 100000 and 1000000. For every workload and number of operations combination, tests were run on 1, 2, 4, 8, 16, 32, 64, 128, 256 and 512 client threads, with every test repeated three times for

every number of client threads.
MongoDB was installed with default settings and the default storage engine for version 3.4.4, Wired Tiger.
Cassandra was installed with default settings, but in order to avoid write timeouts, we followed the setting recommendation found in [8], which is:

- `read_request_timeout_in_ms` set to 50000
- `write_request_timeout_in_ms` set to 100000
- `counter_write_request_timeout_in_ms` set to 100000
- `range_request_timeout_in_ms` set to 100000.

For both databases, the asynchronous Java driver was used.
A combination of workload, database and number of operations will be considered in this context a batch of tests. The database server was restarted before each execution of a batch of tests, and database server status information was captured before and after each run of batch of tests. When all combinations of tests were run for a certain workload, the data set for that workload was deleted and a data set with the same parameters corresponding to the next workload was loaded.

3.2. **Results.** Each test was repeated three times for every combination of database, workload, number of operations and number of client threads. In order to create the following charts, a throughput average was computed for every combination of database, workload, number of operations and number of threads. The first eight graphics (Figures 1 to 8) show a comparison between Cassandra and MongoDB for every combination of workload and number of operations. The last four graphics (Figures 9 to 12) show the evolution of throughput for every combination of workload and database considered in our experimental study.

Figures 1, 2, 3, and 4 show that MongoDB outperforms Cassandra when number of operations is small (1000 and 10000, respectively), for both workloads used.

Figure 5 shows that Cassandra's performance is closer to MongoDB's when the number of operations is increased to 100000, in the case of a update-heavy workload A. For the same number of operations, MongoDB still outperforms Cassandra when we use a read-heavy workload B, as Figure 6 shows.

Cassandra outperforms MongoDB only when the number of operations is 1000000 and the workload is update-heavy, as in Figure 7. For read-heavy workloads and the same number of operations, MongoDB's performance is better, as shown in Figure 8.

Figures 9 and 10 show the individual performance of the databases considered when using a heavy-update workload A, as function of the number of

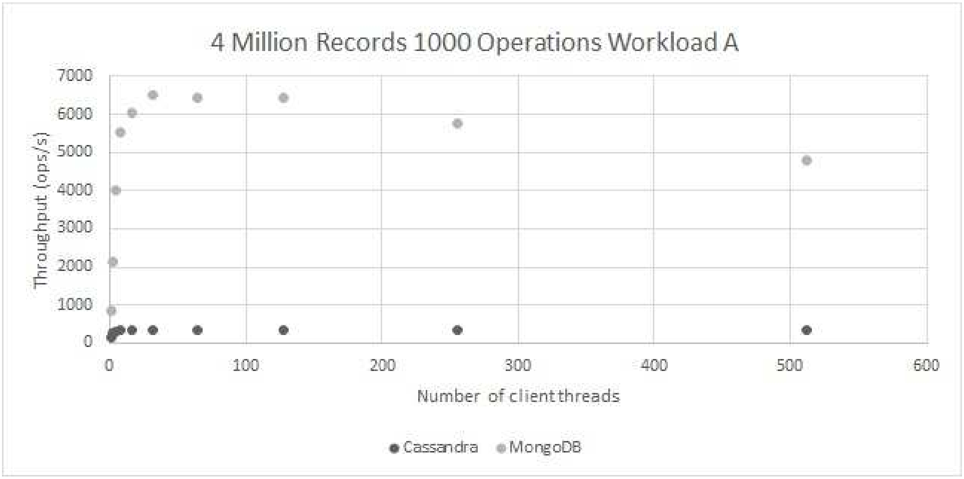


FIGURE 1. 4 Million Records 1000 Operations Workload A

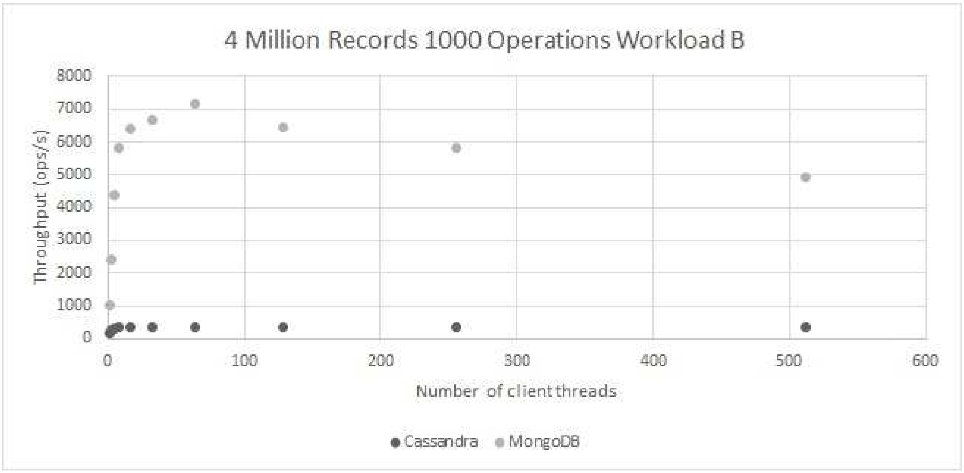


FIGURE 2. 4 Million Records 1000 Operations Workload B

threads used and number of operations involved. After the initial steep increase (128 threads for Cassandra, 32 for MongoDB), the performance flattens (with a very small decrease in the case of MongoDB). In the case of Cassandra, the performance (Figure 9) depends on the number of operations in a quasi-logarithmic fashion, while MongoDB's (Figure 10) throughput is almost the same when the number of operations is greater than or equal to 10000, with

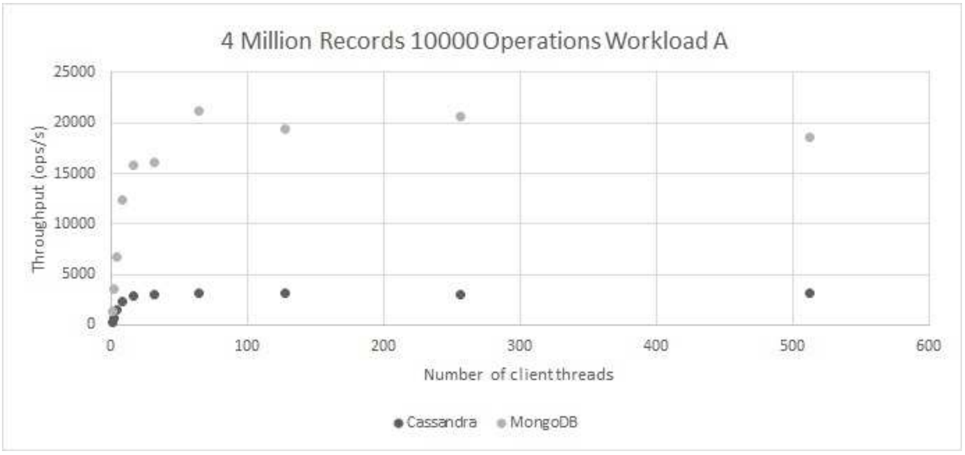


FIGURE 3. 4 Million Records 10000 Operations Workload A

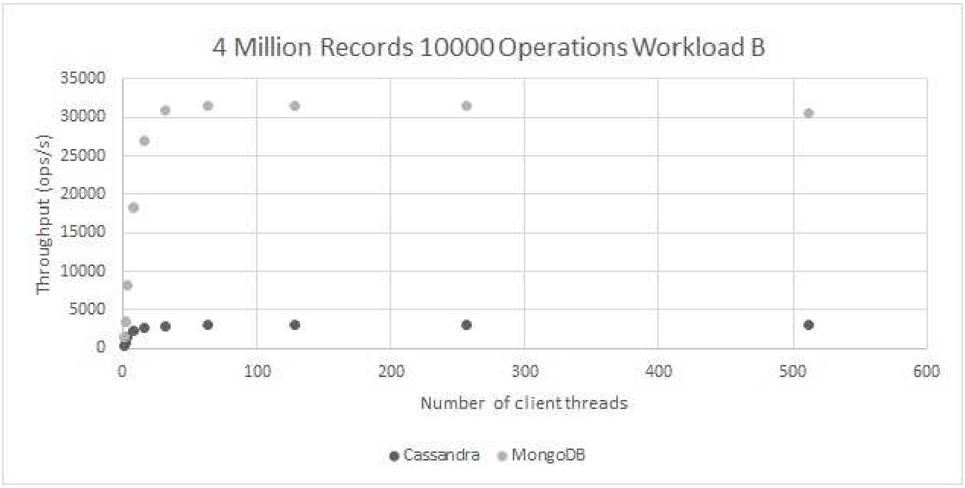


FIGURE 4. 4 Million Records 10000 Operations Workload B

the remark that it is slightly smaller when the number of operations increases from 100000 to 1000000.

The same comparison was performed in the Figures 11 and 12 to show the individual performance of the databases considered when using a heavy-read workload B. A first remark is that the performance decreases in the case of Cassandra (from 43000 to 30000, Figure 11) and increases in the case of MongoDB (from 23000 to 75000, Figure 12). After the initial steep increase (64

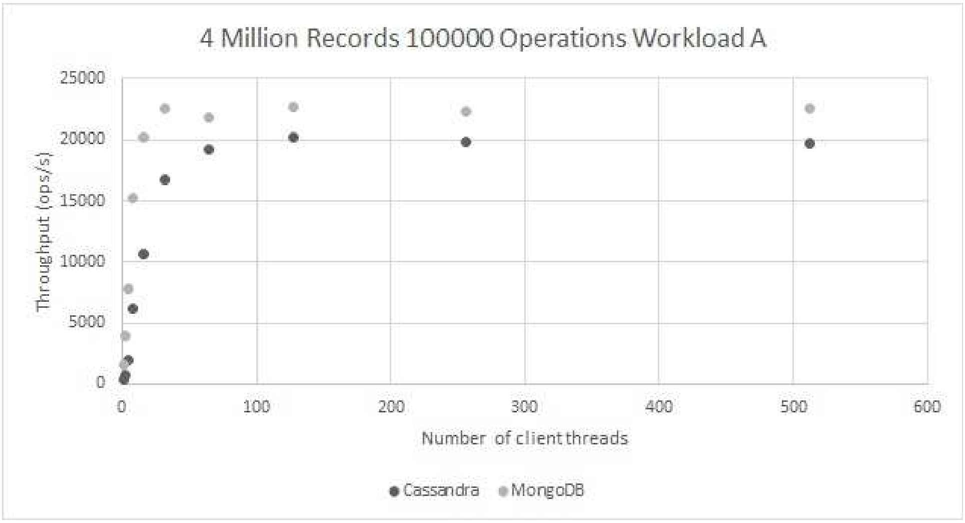


FIGURE 5. 4 Million Records 100000 Operations Workload A

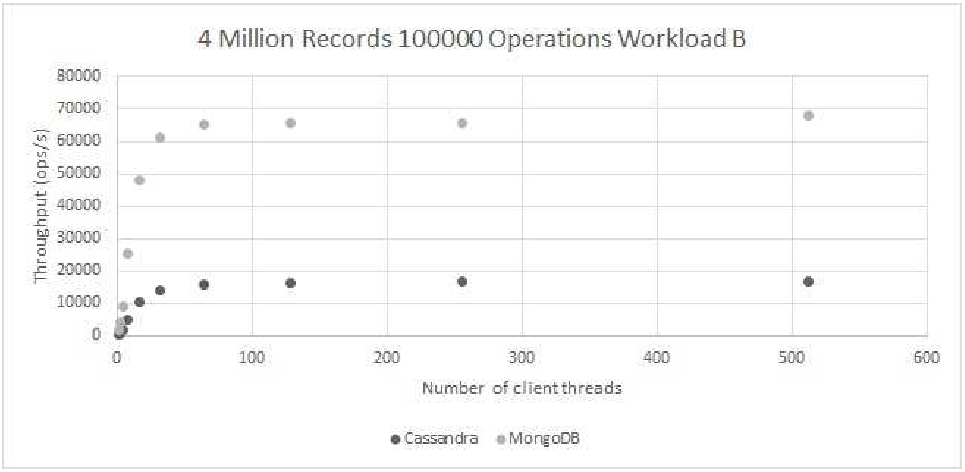


FIGURE 6. 4 Million Records 100000 Operations Workload B

threads for Cassandra, 32 for MongoDB), the performance flattens, following the same patterns.

3.3. **Statistical analysis.** Statistical analysis of the experimental results was performed using two-way ANOVA (Analysis of Variance) procedure from R Statistics Package[17]. A synthesis of the results is given in Table 1. For

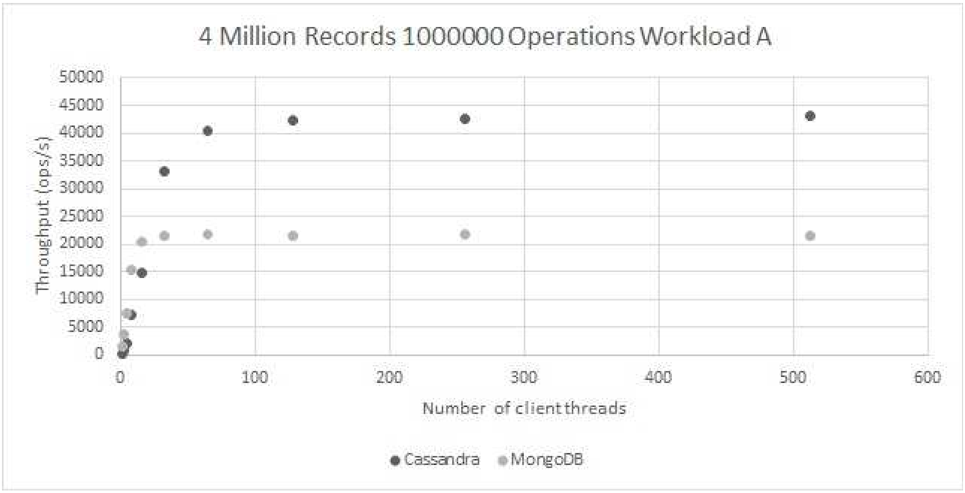


FIGURE 7. 4 Million Records 1000000 Operations Workload A

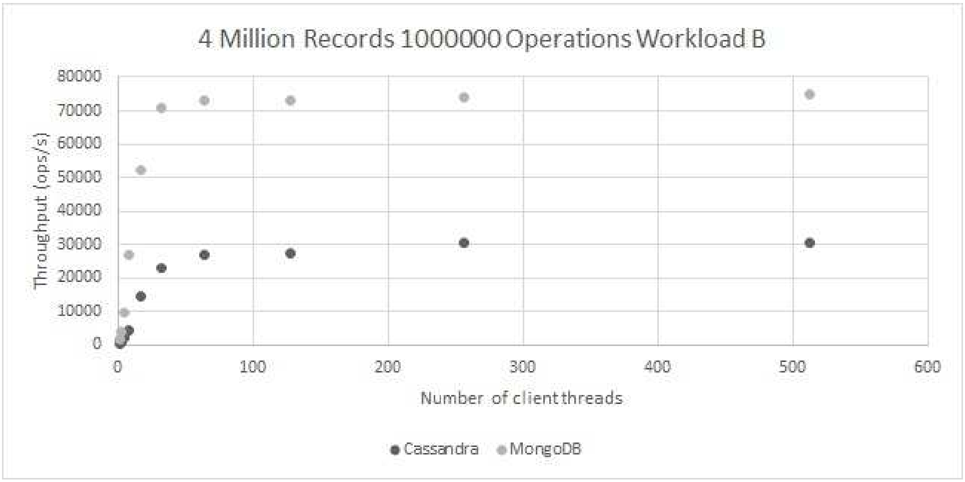


FIGURE 8. 4 Million Records 1000000 Operations Workload B

each experiment, two factors were considered: database (DB, with two levels: Cassandra and MongoDB), and the number of threads (NT, with ten levels: 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512). The interactions between DB and NT were also considered. The column labeled ”Sgf” is referring to the P-value and describes textually the level of significance, 0.1%, 1%, 5%, and 10%, according to the following conventions: $0 *** 0.001 ** 0.01 * 0.05 . 0.1$ (*blank*) 1. In

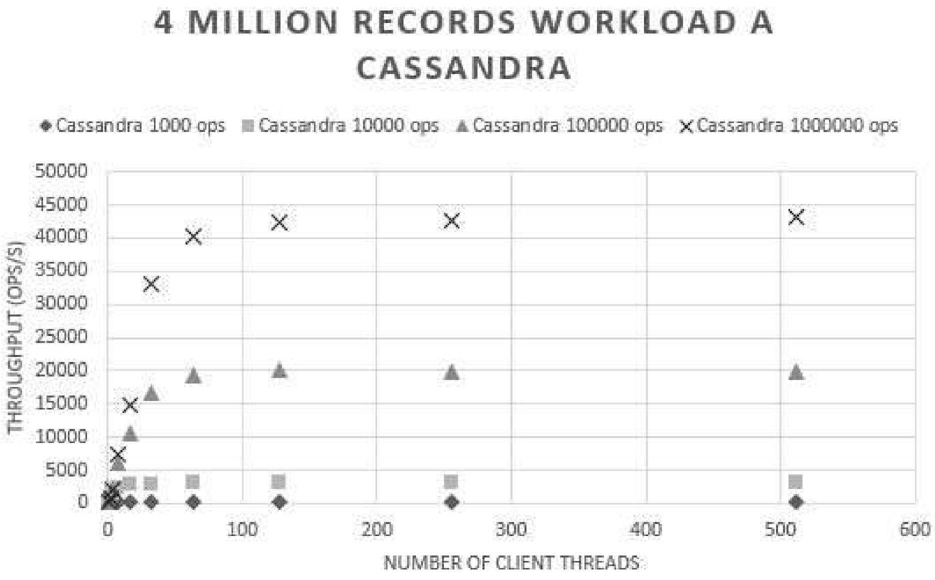


Figure 9. 4 Million Records Cassandra Workload A

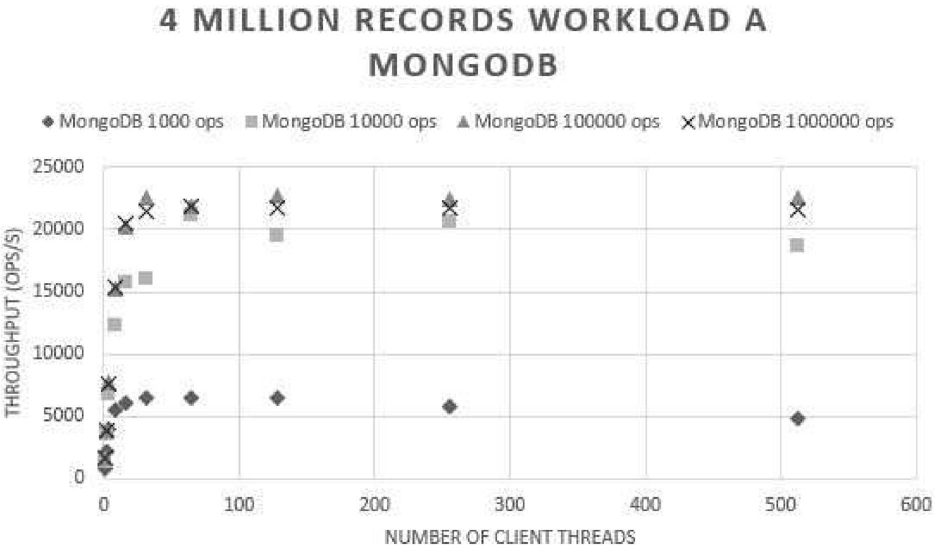


Figure 10. 4 Million Records MongoDB Workload A

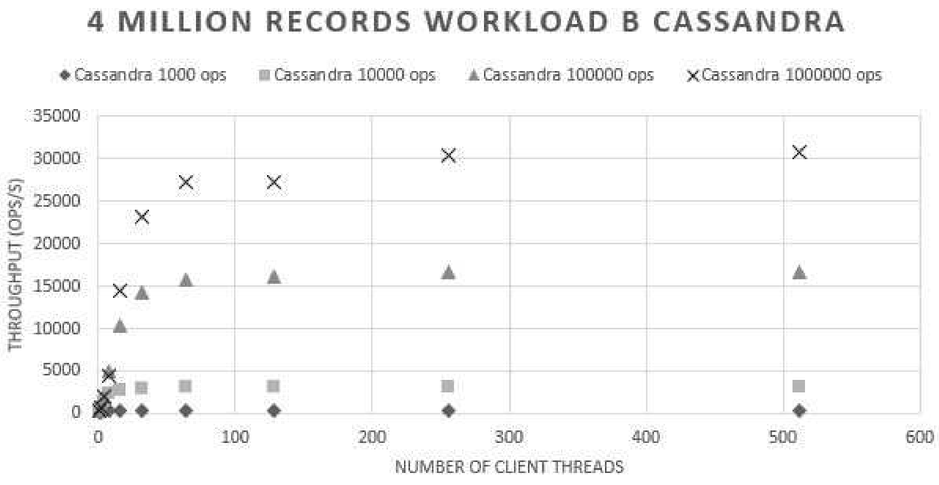


Figure 11. 4 Million Records Cassandra Workload B

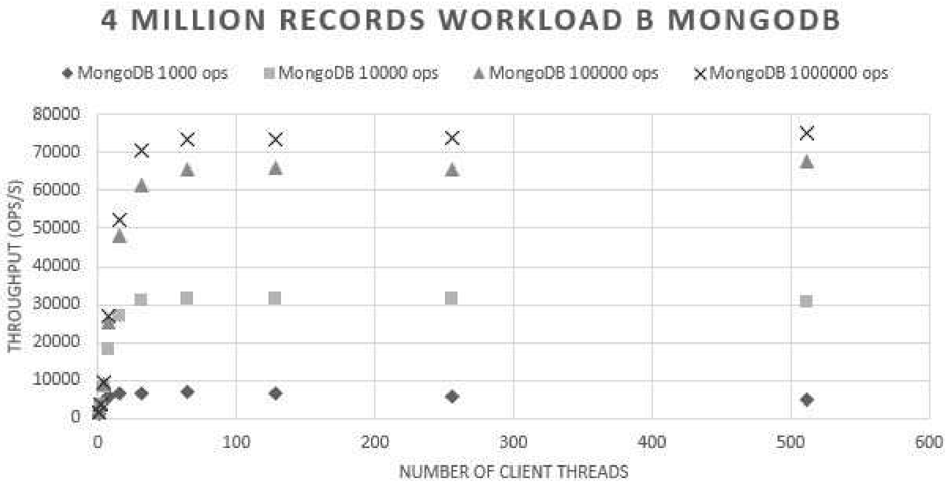


Figure 12. 4 Million Records MongoDB Workload B

other words, if P-value is $\leq 0.1\%$ (i.e. $***$ according to the legend), it means that the differences between means have a strongest statistical significance, while a P-value greater than 10% (i.e. blank space) shows that the differences between the means of the levels considered are within the experimental error.

Table 1. Analysis of variance - results

Wrk ld	No ops	Database			No of threads			DB:NT		
		F-value	Pr(>F)	Sgf	F-value	Pr(>F)	Sgf	F-value	Pr(>F)	Sgf
A	1000	162.4446	<2E-16	***	1.1394	0.2904		0.9268	0.3398	
A	10000	94.3802	1.29E-13	***	12.521	0.0008174	***	7.1367	0.0098707	**
A	100000	6.3535	0.01459	*	26.268	3.82E-06	***	0.3309	0.56742	
A	1000000	5.9014	0.018362	*	31.2701	6.94E-07	***	8.7875	0.004449	**
B	1000	178.571	<2E-16	***	0.75	0.3902		0.5777	0.4504	
B	10000	96.271	9.06E-14	***	11.05	0.001568	**	7.963	0.006596	**
B	100000	56.322	5.07E-10	***	22.632	1.42E-05	***	7.61	0.007827	**
B	1000000	36.373	1.35E-07	***	27.8642	2.19E-06	***	3.4366	0.06904	.

4. Conclusions and further work

After the results were analyzed, it became obvious that for a read-mostly workload (Workload B), MongoDB performed much better than Cassandra. MongoDB outperformed Cassandra in every test combination where the workload parameter was set to Workload B.
For an update-heavy workload (Workload A), Cassandra outperformed MongoDB when the number of operations was increased at 1000000 (Figure 7). In this update-heavy context, MongoDB performed much better than Cassandra in the first two test scenarios where the number of operations was set to 1000 (Figure 1), respectively 10000 (Figure 3). In the third test scenario, where the number of operations was set to 100000 (Figure 5), Cassandra's performance was comparable to MongoDB's, but not greater. After the number of operations was set at 100000, MongoDB's performance stopped growing, while Cassandra's one kept growing. Due to big differences in the infrastructure (cloud-based versus on-premises) and tool and database management systems versions used for benchmarking studies, the results of our work cannot be compared with the results reported by other studies. However, it is important to notice that the general trends are preserved, as they are mentioned by the technical documentation issued by providers.
As further work, we intend to analyze the latency metric results for this experiment, to perform post-hoc ANOVA tests and to run performance benchmarking using data sets that don't fit within memory on single server and cluster configurations. We also plan to run performance benchmarking on servers that use SSDs as disk storage and to enable replication for database servers to see how it affects performance. Moreover, the other variable in our future case studies will be the benchmarking tool, i.e. we'll try to use benchmarking tools available on Linux platforms.

Acknowledgments

Parts of this work were supported through the MADECIP project *Disaster Management Research Infrastructure Based on HPC.* This project was granted to Babeş-Bolyai University, its funding being provided by the Sectoral Operational Programme *Increase of Economic Competitiveness*, Priority Axis 2, co-financed by the European Union through the European Regional Development Fund *Investments in Your Future* (POSCEE COD SMIS CSNR 488061862).

References

[1] M. H. F. R. M. P. A. C. H.-A. J. Ahmad Ghazal, Tilmann Rabl. Bigbench: towards an industry standard benchmark for big data analytics. *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1197–1208, 2013.

[2] Apache cassandra. `http://cassandra.apache.org/`. Accessed: 2017-09-25.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *OSDI '06 Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 7, 2006.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[5] Couchdb. `http://couchdb.apache.org/`. Accessed: 2017-09-25.

[6] Datagres. Perfaccel performance benchmark:nosql database mongodb. Technical report, Datagres Technologies Inc., 2015.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, oct 2007.

[8] Fixstars. Griddb and cassandra performance and scalability. a ycsb performance comparison on microsoft azure. Technical report, Fixstars Solutions, 2016.

[9] A. Gandini, M. Gribaudo, W. J. Knottenbelt, R. Osman, and P. Piazzolla. Performance evaluation of nosql databases. *EPEW 2014: Computer Performance Engineering, Lecture Notes in Computer Science*, 8721:16–29, 2014.

[10] Hbase. `https://hbase.apache.org/`. Accessed: 2017-09-25.

[11] N. E. P. D. K. P. C. M. John Klein, Ian Gorton. Performance evaluation of nosql databases: A case study. *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*, pages 5–10, 2015.

[12] Json. `https://www.json.org/`. Accessed: 2018-03-16.

[13] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44:35–40, 2010.

[14] Mongodb. `https://www.mongodb.com/`. Accessed: 2017-09-25.

[15] Orientdb. `http://orientdb.com/`. Accessed: 2017-09-25.

[16] M. F. Pramod J. Sadalage. *NoSQL distilled : a brief guide to the emerging world of polyglot persistence.* Addison-Wesley Professional, 2012.

[17] R statistics package. `https://www.r-project.org/`. Accessed: 2017-09-25.

[18] Stress test for couchbase client and cluster. `http://docs.couchbase.com/sdk-api/couchbase-c-client-2.4.8/md_doc_cbc-pillowfight.html`. Accessed: 2017-09-25.
[19] The cassandra-stress tool. `https://docs.datastax.com/en/cassandra/2.1/cassandra/tools/toolsCStress_t.html`. Accessed: 2017-09-25.
[20] The ycsb core workloads. `https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads`. Accessed: 2017-09-25.
[21] Xml. `https://www.w3.org/TR/2008/REC-xml-20081126/`. Accessed: 2018-03-16.
[22] Yaml. `http://yaml.org/`. Accessed: 2018-03-16.
[23] Ycsb github wiki. `https://github.com/brianfrankcooper/YCSB/wiki`. Accessed: 2017-09-25.

Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania
Email address: {`andorcamelia, bparv`}`@cs.ubbcluj.ro`

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 1, 2018
DOI: 10.24193/subbi.2018.1.07

AN ADAPTIVE GRADUAL RELATIONAL ASSOCIATION RULES MINING APPROACH

DIANA-LUCIA MIHOLCA

Abstract. This paper focuses on adaptive Gradual Relational Association Rules mining. Gradual Relational Association Rules capture gradual generic relations among data features. We propose *AGRARM*, an algorithm for mining the interesting Gradual Relational Association Rules characterizing a data set that has been extended with a number of new attributes, through adapting the set of interesting rules mined before extension, so as to preserve the completeness. We aim, through *AGRARM*, to make the mining process more efficient than resuming the mining algorithm on the enlarged data. We have experimentally evaluated *AGRARM* versus mining from scratch on three publicly available data sets. The obtained reduction in mining time highlights *AGRARM*'s efficiency, thus confirming the potential of our proposal.

1. Introduction

Data mining is widely applied in various domains, such as medicine [5], bioinformatics [6] or software engineering [10] [9] [14], to discover relevant patterns in large data sets.

Association Rules (ARs) *mining* [4] is a data mining procedure for identifying frequent associations in data. Classical association rules capture frequent co-occurrences of attribute values, while ignoring any possible frequent relation between attribute values.

Ordinal Association Rules (OARs) [3] customize *Association Rules* (ARs) [1] so as to express ordinal relations among numeric attributes that characterize a data set. But different informative relations, that are not ordinal, may exist between the attribute values. OARs fail to capture them.

Received by the editors: April 2, 2018.

2010 *Mathematics Subject Classification.* 68P15, 68T05, 62B86.

1998 *CR Categories and Descriptors.* H.2.8[**Database management**]: Database Applications – *Data Mining*; I.2.6[**Computing Methodologies**]: Artificial Intelligence – *Learning*.

Key words and phrases. data mining, Gradual Relational Association Rules, adaptive algorithm.

Consequently, *Relational Association Rules* (RARs) [18] [2] generalize *Ordinal Association Rules* so as to capture relations that may not be ordinal, between not necessary numeric attributes. Compared to the classical *Association Rules*, RARs express more powerful rules which may lead to valuable data mining results.

Subsequently, *Adaptive Relational Association Rule Mining* (*ARARM*) [8] has been proposed as a method for adapting the set of all interesting RARs discovered within a data set before extending its features set, so as to obtain all interesting RARs within the extended data set.

There are situations when the degree to which a relation between two attributes is satisfied is relevant. So, RARs have been further extended to *Gradual Relational Association Rules* (or GRARs) [7] which, through the use of *fuzzy* relations instead of *boolean* relations, are also aware of the degree to which the relations are satisfied.

For discovering all the interesting Gradual Relational Association Rules that describe a data set, *Gradual Relational Association Rules Miner* (*GRANUM*) [7] has been proposed. *GRANUM* mines a known set of objects that are measured against a known set of features and discovers all interesting GRARs characterizing the data set. But there are also situations where the data is horizontally dynamic, in the sense that the feature set characterizing its objects evolves (i.e. new attributes are added). Clearly, for obtaining, in such a setting, the interesting GRARs, the mining algorithm can be re-applied, from scratch, every time the feature set changes (i.e. one or more new attributes are added). But this could be inefficient and unworthy especially if the attribute set is only very slightly expanded, for instance by adding just one new attribute.

Consequently, we propose, in the current paper, an alternative to resuming the *GRANUM* mining algorithm when the data set is enlarged with a number of new attributes. We propose, therefore, *Adaptive Gradual Relational Association Rules Miner* (*AGRARM*), which is an algorithm that adapts the set of all interesting GRARs mined before extension so as to obtain all interesting GRARs that characterize the extended data. *AGRARM* is the equivalent of *ARARM* [8], but for mining GRARs instead of RARs, within a dynamic data set.

The remaining of this paper is structured as follows. We start by giving, in Section 2, a background on *Gradual Relational Association Rules.* The proposed *Adaptive Gradual Relational Association Rules Miner* (*AGRARM*) is presented in Section 3. In Section 4, we detail the experiments performed in order to evaluate *AGRARM* against *GRANUM* applied from scratch and we discuss the results obtained. A comparison to related approaches is also given in Section 4. Finally, the conclusions and directions for further work are stated in Section 5.

2. Background on Gradual Relational Association Rules

We briefly present in the following the concept of *Gradual Relational Association Rules* [7].

Gradual Relational Association Rules (*GRARs*) generalize *Relational Association Rules* (*RARs*) [18] by using *fuzzy relations* instead of crisp relations and thus enhancing them with gradualness. The gradual rules are able to express additional semantically relevant characteristics of data and have been proven to be more noise-tolerant [7].

Let $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$ be a set of *instances* (entities, records or objects). Each instance e_i in $\mathcal{E}$ consists of a sequence of values for m *attributes* (or features), $\mathcal{A} = (a_1, \ldots, a_m)$. Each attribute a_j takes values from a non-empty and non-fuzzy domain D_i, which also contains a *null* (or empty) value. If we denote by $\Phi(e_i, a_j)$ the value of the instance e_i for the attribute a_j, an instance will be $e_i = (\Phi(e_i, a_1), \Phi(e_i, a_2),$
$\Phi(e_i, a_3), ...\Phi(e_i, a_m))$.

A fuzzy binary relation $\mathcal{G}$ between two attribute domains D_i and D_j is defined as follows:

$$\mathcal{G} = \{< (v_1, v_2),\ \mu_R(v_1, v_2) >: v_1 \in D_i, v_2 \in D_j\}$$

$\mu_R : D_i \times D_j \to [0, 1]$ is a *membership* function which associates to each pair $(v_1, v_2), v_1 \in D_i, v_2 \in D_j$ the *membership degree* $\mu_R(v_1, v_2)$ which numerically expresses the degree to which the relation $\mathcal{G}$ is satisfied.

We denote by $\mathcal{F}$ the set of all fuzzy binary relations which can be defined between any two crisp attribute domains.

Definition 2.1. *A **Gradual Relational Association Rule**, gRule, is a sequence* $(a_{i_1}\ \mathcal{G}_1\ a_{i_2}\ \mathcal{G}_2\ a_{i_3} \ldots \mathcal{G}_{\ell-1}\ a_{i_\ell})$, *where* $\{a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_\ell}\} \subseteq \mathcal{A} = \{a_1, \ldots, a_m\}$, $a_{i_j} \neq a_{i_k}$, $j, k = 1..\ell$ *and* $\mathcal{G}_j \in \mathcal{F}$ *is a binary fuzzy relation over* $D_{i_j} \times D_{i_{j+1}}$ [7].

*The **membership degree** of the gradual relational association rule gRule for data instance* $e \in \mathcal{E}$ *is defined as* $\mu_{gRule}(e) = \min\{\mu_{R_j}(\Phi(e, a_{i_j}), \Phi(e, a_{i_{j+1}}))$, $j = 1, 2, \ldots, \ell - 1\}$ *and expresses the magnitude to which the rule is satisfied.*

a) *If* $a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_\ell}$ *are non-missing in p instances from the data set then we call* $\frac{p}{n}$ *the **support** of the rule.*
b) *If we denote by* $\mathcal{E}' \subseteq \mathcal{E}$ *the set of instances where* $a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_\ell}$ *are non-missing and* $\mu_{gRule}(e) > 0$ *for each instance e from* $\mathcal{E}'$, *then we call* $\frac{|\mathcal{E}'|}{n}$ *the **confidence** of the rule.*

c) *Using the notation from b), we call* $\frac{\sum_{e \in \mathcal{E}'} \mu_{gRule}(e)}{n}$ *the rule's* ***membership.***

The number l of attributes in a rule gives the rule *length.*

When introducing the concept of *Gradual Relational Association Rules* in the literature [7], we kept the definition of *interestingness* previously proposed for non-gradual *Relational Association Rules.* In accordance with this, a rule is *interesting* if its support and confidence are greater or equal to given thresholds. In a later work [14], we suggested that we could customize *interestingness* by including an additional minimum threshold condition for *membership.* So, the current work is in accordance with the definition for *interestingness* customized as follows:

Definition 2.2. *We call a GRAR* interesting *if its support s, confidence c and membership m are greater than or equal to given thresholds, i.e. $s \geq s_{min}$, $c \geq c_{min}$ and $m \geq m_{min}$.*

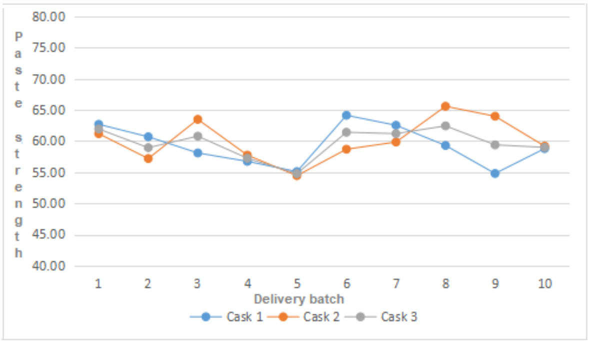
Definition 2.3. *The* inverse *of binary fuzzy relation $\mathcal{G} = \{< (x, y), \mu_{\mathcal{G}}(x, y) >: x \in X, y \in Y\}$ will be denoted in the following by $\mathcal{G}^{-1}$ and is defined as $\mathcal{G}^{-1} = \{< (x, y), 1 - \mu_{\mathcal{G}}(x, y) >: x \in X, y \in Y\}$.*

GRANUM [7] has been proposed as an *Apriori* mining algorithm for discovering all *interesting* GRARs within a data set. For more details about *GRANUM* and *GRARs* in general, we refer the reader to [7].

2.1. **Example.** We exemplify in the following the previously presented concept of *Gradual Relational Association Rules.* Therefore, we mine a small real data set taken from [12] and depicted in Figure 1. The data consist of the results obtained by testing chemical pastes as described in the following. The chemical paste product is delivered in batches of casks. Immediately after the arrival of a batch, the material from three randomly selected casks is analyzed, errors arising from both the sampling and the analysis. The data instances correspond to ten delivery batches chosen at random, while the data attributes are given by the average of the percentage paste strengths obtained by two analyzes of the contents of the three selected casks.

We propose to compare the paste strengths obtained by analyzing the contents of the three randomly selected casks. Since there are errors in data, we opt for GRARs [7] instead of non-gradual RARs.

Having $\mathcal{F} = \{\approx$ (*approximately equal*), $\lesssim$ (*fuzzy less*)*and* $\gtrsim$ (*fuzzy greater*)$\}$ as the set of gradual relations and setting the minimum support, confidence and membership thresholds at $s_{min} = 1$, $c_{min} = 1$ and $m_{min} = 0.9$, the



Cask 1	Cask 2	Cask 3
62.70	61.20	61.95
60.70	57.20	58.95
58.10	63.50	60.80
56.75	57.75	57.25
55.10	54.45	54.78
64.15	58.70	61.43
62.55	59.85	61.20
59.30	65.60	62.45
54.80	64.00	59.40
58.80	59.20	59.00

FIGURE 1. *Strength of chemical pastes* data set

GRANUM mining algorithm will discover as interesting rules the rules given in Table 1.

Rule	Length	Support	Confidence	Membership
Cask 1 ≈ **Cask 2**	2	1.0	1.0	0.935
Cask 1 ≈ **Cask 3**	2	1.0	1.0	0.982
Cask 2 ≈ **Cask 3**	2	1.0	1.0	0.983
Cask 1 ≈ **Cask 2** ≈ **Cask 3**	3	1.0	1.0	0.935

TABLE 1. Interesting rules on data set from Table 1 for $s_{min} = 1$, $c_{min} = 1$ and $m_{min} = 0.9$

Interpreting the obtained GRARs, we can conclude that the results of the analyzes performed for the three selected casks are approximately equal (since *Cask 1* ≈ *Cask 2* ≈ *Cask 3* with a rather large membership degree of 0.935). Furthermore, we deduce that the strengths of the material from the third selected cask differ in almost equal extents from the strengths obtained for the other two casks (since *Cask 1* ≈ *Cask 3* with membership 0.982 and *Cask 2* ≈ *Cask 3* with membership 0.983), while these two are not as close to each other (since *Cask 1* ≈ *Cask 2* with a smaller membership of 0.935). These conclusions are confirmed by analyzing the graphical data representation from Figure 1.

3. METHODOLOGY

We introduce in the current section *AGRARM*, the ***A****daptive* ***G****radual* ***R****elational* ***A****ssociation* ***R****ules* ***M****ining* method we propose for mining all interesting *GRARs* in a dynamic data set whose feature set is extended with one or more new features.

Let $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$ be a data set. Each entity is initially defined by the values for m features (attributes or characteristics), $\mathcal{A} = (a_1, \ldots, a_m)$, thus being a m-dimensional sequence: $e_i = (e_i^1, \ldots, e_i^m)$. Subsequently, $\mathcal{A}$ is extended with $s \geq 1$ new features, thus obtaining an extended feature set $\mathcal{A}^{ext} = (a_1, \ldots, a_m, a_{m+1}, \ldots,$
$a_{m+s})$ and an afferent extended data set $\mathcal{E}^{ext} = \{e_1^{ext}, e_2^{ext}, \ldots, e_n^{ext}\}$. Each extended data instance $e_i^{ext} \in \mathcal{E}^{ext}$ is therefore given by the values for the $m+s$ attributes that describe the extended data set $\mathcal{E}^{ext}$: $e_i^{ext} = (e_i^{ext,\ 1}, e_i^{ext,\ 2}, \ldots, e_i^{ext,\ m+s})$.

In this context, the problem we are approaching is to find the set $\mathcal{G}Rules^{ext}$ of all interesting *GRARs* that occur in the extended data set $\mathcal{E}^{ext}$, starting from the set $\mathcal{G}Rules$ of all interesting *GRARs* in the non-extended data set $\mathcal{E}$. The motivation is that we expect a better time performance through completing the rules already mined on the data before extension than by applying the mining process from scratch on the extended data.

So, we further present *AGRARM* (***A**daptive **G**radual **R**elational **A**ssociation **R**ule **M**iner*), a complete algorithm that, starting from $\mathcal{G}Rules$ and considering the newly added features, adapts the rule set so as to obtain $\mathcal{G}Rules^{ext}$.

```
Function AGRARM($\mathcal{E}$, $\mathcal{E}^{ext}$, $\mathcal{F}$, $\mathcal{G}Rules$, $c_{min}$, $s_{min}$, $m_{min}$)
Input:    $\mathcal{E}$ - the initial non-extended set of $m$-dimensional entities,
          $\mathcal{E}^{ext}$ - the final extended set of $m+s$-dimensional entities,
          $\mathcal{F}$ - the set of fuzzy binary relations used in the mining process,
          $\mathcal{G}Rules$ - the set of all interesting $GRARs$ mined on the non-extended data
      set $\mathcal{E}$,
          $c_{min}$, $s_{min}$ and $m_{min}$ - the minimum thresholds for support, confidence
      and membership, respectively
Output:   $\mathcal{G}Rules^{ext}$ - the set of all interesting $GRARs$ that characterize $\mathcal{E}^{ext}$, the
      extended data set
      $AdaptiveRules \leftarrow the\ binary\ (2-length)\ rules\ from\ \mathcal{G}Rules$
      $Cand \leftarrow \{\ (a_{i_1}\ \mathcal{G}\ a_{i_2}) \mid a_{i_1}, a_{i_2} \in A,\ i_1 = 1 \ldots m+s, i_2 = m+1 \ldots m+s,\ i_1 <$
      $i_2,\ \mathcal{G} \in \mathcal{F}\ \}$
      Foreach $gRule$ in $Cand$ do
         If $IsInteresting(gRule, \mathcal{E}^{ext}, c_{min}, s_{min}, m_{min})$ then
            $AdaptiveRules \leftarrow AdaptiveRules \cup \{gRule\}$
         EndIf
      EndFor
      $\mathcal{G}Rules^{ext} \leftarrow AdaptiveRules$
      $l \leftarrow 3$
      $complete \leftarrow false$
      While ($\neg complete$) do
         $Cand \leftarrow GenCandidates(AdaptiveRules)$
         $AdaptiveRules \leftarrow l - length\ rules\ from\ \mathcal{G}Rules$
      Foreach $gRule$ in $Cand$ do
        If $IsInteresting(gRule,\ \mathcal{E}^{ext},\ c_{min},\ s_{min},\ m_{min})$ then
          $AdaptiveRules \leftarrow AdaptiveRules\ \cup\ \{gRule\}$
        EndIf
      EndFor
      If $AdaptiveRules = \emptyset$ then
        $complete \leftarrow true$
      else
        $l\ \leftarrow l + 1$
        $\mathcal{G}Rules^{ext} \leftarrow \mathcal{G}Rules^{ext} \cup AdaptiveRules$
      EndIf
    EndWhile
    $AGRARM \leftarrow \mathcal{G}Rules^{ext}$
  EndFunction
```

The *AGRARM* algorithm discovers all interesting *GRARs* through an iterative process. At each iteration, the length-level generation of rules is followed by the verification of their *interestingness.* As we mentioned in Section 2 , the *interestingness* of a *GRARs* is a property that is tested in relation to given support, confidence and membership minimum thresholds. We give in the following the function that checks if a candidate *GRAR* is or is not interesting at the level of the extended data set $\mathcal{E}^{ext}$.

```
  Function $IsInteresting(gRule,\ \mathcal{E}^{ext},\ c_{min},\ s_{min},\ m_{min})$
Input:   $\mathcal{E}^{ext}$ - the final extended set of $m{+}s$-dimensional entities,
         $gRule$ - the gradual relational association rule whose interestingness on $\mathcal{E}^{ext}$
    is verified
         $c_{min}$, $s_{min}$ and $m_{min}$ - the minimum thresholds for support, confidence
    and membership, respectively
Output:  $true$ - if $gRule$ is interesting on $\mathcal{E}^{ext}$ (i.e. it satisfies $c_{min}$, $s_{min}$, and
    $m_{min}$ minimum thresholds) or
         $false$ - otherwise
    $n \leftarrow |\mathcal{E}^{ext}|$
    $requiredSuppport \leftarrow \lceil n \cdot s_{min} \rceil$
    $requiredConfidence \leftarrow \lceil n \cdot c_{min} \rceil$
    $requiredMembership \leftarrow \lceil n \cdot m_{min} \rceil$
    $support \leftarrow 0$
    $confidence \leftarrow 0$
    $membership \leftarrow 0$
    $remainingEntities \leftarrow n$
    Foreach $instance$ in $\mathcal{E}^{ext}$ do
      $UpdateSuppConfM(gRule, instance, support, confidence, membership)$
      $remainingEntities \leftarrow remainingEntities - 1$
      If $(support + remainingEntities < requiredSupport)$
        $or$ $(confidence + remainingEntities < requiredConfidence)$
        $or$ $(membership + remainingEntities < requiredMembership)$ then
        $IsInteresting \leftarrow false$
      EndIf
      If ($support \geq requiredSupport$) $and$ ($confidence \geq requiredConfidence$)
        $and$ ($membership \geq requiredMembership$) then
        $IsInteresting \leftarrow true$
      EndIf
    EndFor
    $IsInteresting \leftarrow false$
  EndFunction
```

The method that follows presents the update of support, confidence and membership of a *GRAR* when considering a current data instance.

```
  Subalgorithm $UpdateSuppConfM$($gRule, instance, supp, conf, membership$)
Input:   $gRule$ - the gradual relational association rule whose support, confidence
    and membership will be updated considering the $instance$ data entity
       $instance$ - the data instance on which the rule $gRule$ is evaluated so as to
    update the $supp$, $conf$ and $membership$ values
       $supp$, $conf$ and $membership$ - the current support, confidence and mem-
    bership for $gRule$ which are required to be updated through also considering
    $instance$.
Output:   $supp'$, $conf'$ and $membership'$ - the support, confidence and member-
    ship of $gRule$ are updated as a result of evaluating $gRule$ on $instance$
    If @$instance$ $has$ $non-missing$ $values$ $for$ $all$ $attributes$ $in$ $gRule$ then
      $supp \leftarrow supp + 1$
      $m \leftarrow min$(@ $the$ $memberships$ $of$ $the$ $fuzzy$ $relations$ $in$ $gRule$ $on$ $the$
      $instance$ $data$ $entity$)
      If $m > 0$ then
        $conf \leftarrow conf + 1$
      EndIf
      $membership \leftarrow membership + m$
    EndIf
  EndSubalgorithm
```

So, *AGRARM*, the proposed method, starts by performing an initial pass over the extended data set $\mathcal{E}^{ext}$ so as to identify the interesting binary rules in addition to the 2-length rules from *GRules*. In every subsequent iteration, the set of interesting rules of length $k > 2$ will be mined. This set will obviously include the k-length rules from the set $\mathcal{G}Rules$. But there is an alternative to obtain a k-length interesting rule. The alternative consists in generating a new candidate rule by joining two $(k-1)$-length rules from $\mathcal{G}Rules^{ext}$ such that at least one of the two rules contains at least one newly added attribute. The candidate rules generation is followed by the verification of minimum support, confidence and membership compliance. At the end of each iteration, all the k-length interesting rules will be included in the set $\mathcal{G}Rules^{ext}$. The mining process stops when no new interesting rules have been discovered in the latest iteration.

We present in the following the method of generating candidate rules.

```
Function $GenCandidates(\mathcal{G}Rules_k)$
Input:  $\mathcal{G}Rules_k$ - the interesting $GRARs$ of length $k$
Output:  $\mathcal{G}Rules_{k+1}$ - the candidate $GRARs$ of length $k+1$ which were obtained
    through joining pairs of rules in $\mathcal{G}Rules_k$
    $\mathcal{G}Rules_{k+1} \leftarrow \emptyset$
    $n \leftarrow |\mathcal{G}Rules_k|$
    For $i \leftarrow 1$ to $n-1$ do
      For $j \leftarrow i+1$ to $n$ do
        $gRule_i \leftarrow$ the $i-th$ rule from $\mathcal{G}Rules_k$
        $gRule_j \leftarrow$ the $j-th$ rule from $\mathcal{G}Rules_k$
        If @$gRule_i$ or $gRule_j$ contain at least one newly added attribute (i.e.
          in the set $\{a_{m+1}, a_{m+2}, ..., a_{m+s}\}$) then
          If @$gRule_i$ matches for join with $gRule_j$ in one of the cases (1)
            −(4) from Figure 2 then
            $resultingRule \leftarrow$ @ the rule obtained by joining $gRule_i$ and $gRule_j$
            $\mathcal{G}Rules_{k+1} \leftarrow \mathcal{G}Rules_{k+1} \cup \{resultingRule\}$
          EndIf
        EndIf
      EndFor
    EndFor
    $GenCandidates \leftarrow \mathcal{G}Rules_{k+1}$
EndFunction
```

In Figure 2, we present the four rules according to which *GenCandidates* proposes new candidate rules.

4. Results and discussion

We present in the following the experiments we performed in order to comparatively evaluate *AGRARM* against *GRANUM* applied from scratch, the comparison being performed in the context in which the data of interest is extended with a number of new attributes.

In these comparative experiments, we considered three different data sets, various possibilities of extending their attribute sets and multiple values for the minimum support, confidence and membership thresholds.

The three data sets we have considered in our experiments are publicly available in $tera-PROMISE$ repository [17]. They are *Tomcat*, *Ar* and *JM*1. The *Tomcat* data set consists of the values for 20 Chidamber and Kemerer (CK) software metrics, computed for the 858 classes in Apache Tomcat software, version 6.0. The *Ar* data set is composed of 29 static code attributes (McCabe, Halstead and LOC software measures), for 745 modules in Ar, which is an embedded software implemented in C. The third data set, *JM*1, consists of 7782 instances, corresponding to modules in *JM*1 software, each being characterized by 21 attributes (5 different lines of code measures, 3 *McCabe* metrics, 4 base *Halstead* measures, 8 derived *Halstead* measures and a

$$\begin{gathered}
gRule_1 \equiv (a^1\mathcal{G}^1 a_{i_1}\mathcal{G}_1 a_{i_2}\dots\mathcal{G}_{k-3}a_{i_{k-2}}),\\
gRule_2 \equiv (a_{i_1}\mathcal{G}_1 a_{i_2}\dots\mathcal{G}_{k-3}a_{i_{k-2}}\mathcal{G}^2 a^2),\\
\Rightarrow resultingRule \equiv (a^1\mathcal{G}^1 a_{i_1}\mathcal{G}_1 a_{i_2}\dots\mathcal{G}_{k-3}a_{i_{k-2}}\mathcal{G}^2 a^2),
\end{gathered} \tag{1}$$

or

$$\begin{gathered}
gRule_1 \equiv (a_{i_1}\mathcal{G}_1 a_{i_2}\dots\mathcal{G}_{k-3}a_{i_{k-2}}\mathcal{G}^1 a^1),\\
gRule_2 \equiv (a^2\mathcal{G}^2 a_{i_1}\mathcal{G}_1 a_{i_2}\dots\mathcal{G}_{k-3}a_{i_{k-2}}),\\
\Rightarrow resultingRule \equiv (a^2\mathcal{G}^2 a_{i_1}\mathcal{G}_1 a_{i_2}\dots\mathcal{G}_{k-3}a_{i_{k-2}}\mathcal{G}^1 a^1),
\end{gathered} \tag{2}$$

or

$$\begin{gathered}
gRule_1 \equiv (a^1\mathcal{G}^1 a_{i_1}\mathcal{G}_1 a_{i_2}\dots\mathcal{G}_{k-3}a_{i_{k-2}}),\\
gRule_2 \equiv (a^2\mathcal{G}^2 a_{i_{k-2}}\mathcal{G}_{k-3}^{-1}\dots a_{i_2}\mathcal{G}_1^{-1}a_{i_1}),\\
\Rightarrow resultingRule \equiv (a^1\mathcal{G}^1 a_{i_1}\mathcal{G}_1 a_{i_2}\dots\mathcal{G}_{k-3}a_{i_{k-2}}(\mathcal{G}^2)^{-1} a^2),
\end{gathered} \tag{3}$$

or

$$\begin{gathered}
gRule_1 \equiv (a_{i_1}\mathcal{G}_1 a_{i_2}\dots\mathcal{G}_{k-3}a_{i_{k-2}}\mathcal{G}^1 a^1),\\
gRule_2 \equiv (a_{i_{k-2}}\mathcal{G}_{k-3}^{-1}\dots a_{i_2}\mathcal{G}_1{}^{-1}a_{i_1}\mathcal{G}^2 a^2),\\
\Rightarrow resultingRule \equiv (a^2(\mathcal{G}^2)^{-1} a_{i_1}\mathcal{G}_1 a_{i_2}\dots\mathcal{G}_{k-3}a_{i_{k-2}}\mathcal{G}^1 a^1).
\end{gathered} \tag{4}$$

FIGURE 2. The joining rules considered by the candidate generation process in the *AGRARM* algorithm

branch-count). We mention that, prior to the mining phase, the data have been preprocessed in the sense that the values have been scaled using the *Min-Max* scaling method.

In each of the experiments, the interesting *GRARs* on the extended $(m+s)$-dimensional instances have been mined in the following two ways: (1) by applying *GRANUM* from scratch on the extended data and (2) by applying *AGRARM* so as to adapt the rules mined before extension. Certainly, the interesting *GRARs* mined were the same regardless of the mining method applied (i.e. (1) or (2)). But we will compare the time required by the two methods in order to test our expectation that *AGRARM* is faster than *GRANUM* applied from scratch, at least if the data set is expanded with a relatively small number of attributes.

We considered, in the mining processes, the following set of fuzzy binary relations: $\mathcal{F} = \{\approx$ (*approximately equal*), $\lesssim$ (*fuzzy less*), $\gtrsim$ (*fuzzy greater*), $\sim\ll$ (*fuzzy much less*), $\sim\gg$ (*fuzzy much greater*)$\}$. The $\approx$ relation has been defined using the asymmetric Gaussian membership function, while the rest of the fuzzy relations have been defined through S-shaped membership functions, which have been parameterized, of course, so that the following inequalities occur: $\lesssim(x,y) \geq \sim\ll(x,y)$ and $\gtrsim(x,y) \geq \sim\gg(x,y)$.

We mention that the experiments have been carried out on a PC with an Intel Core i7 Processor at 2.40 GHz, with 8 GB of RAM.

We depict in Table 2 the results obtained by applying *AGRARM* versus *GRANUM* from scratch on *Tomcat* data set, when considering the minimum support threshold

m	**s**	**Rules on $\mathcal{E}$**	**Time *GRANUM* (ms)**	**Time *AGRARM* (ms)**	**Time reduction**
2	18	0	273.67	269.66	0.014
3	17	0	275.08	275	0.0002
4	16	0	274.14	273.8	0.001
5	15	0	274.55	272.58	0.007
6	14	9	275.42	264.03	0.041
7	13	9	274.68	261.28	0.048
8	12	9	274.49	259.55	0.054
9	11	32	274.12	237.93	0.132
10	10	32	274.04	232.9	0.150
11	9	32	274.99	230.38	0.162
12	8	32	273.77	226.75	0.172
13	7	32	273.97	223.12	0.186
14	6	32	273.83	219.78	0.197
15	5	99	274.87	119.13	0.567
16	4	100	275.9	115.27	0.582
17	3	117	275.52	88.92	0.677
18	2	171	276.52	14.29	0.948
19	1	171	275.5	8.22	0.970

TABLE 2. Experimental results obtained on *Tomcat* data set for $s_{min} = 1$, $c_{min} = 0.97$ and $m_{min} = 0.5$

$s_{min} = 1$, the minimum confidence threshold $c_{min} = 0.97$ and the minimum membership threshold $m_{min} = 0.5$. Here, m gives the number of initial attributes, while s gives the number of newly added attributes.

In Table 2, we give, on the first column, the number m of initial attributes, on the second column, the number s of newly added attributes, on the third column, the number of interesting *GRAR*s mined before extension, on the fourth and fifth columns the mining time for *GRANUM* and *AGRARM*, respectively, and, on the last column, the time reduction obtained by applying *AGRARM* to the detriment of *GRANUM* applied from scratch. The reduction in mining time has been computed as the ratio between the gained time (i.e. the difference between the time required by *GRANUM* and the time required by *AGRARM*) and the time consumed through resuming the mining process (i.e. applying *GRANUM* from scratch).

We observe from the table that the time reduction becomes significant when the newly added attributes count no more than one third of the number of initial attributes. For instance, when $\frac{s}{m} = \frac{1}{3}$ (i.e. $s = 5$ and $m = 15$), the mining time is reduced by more than 56%. The most substantial reduction, namely 97%, is obtained when the data set is extended with only one attribute.

Figures 3 and 4 illustrate how the time reduction evolves, depending on the number s of new attributes, for additional case studies on the *Tomcat* data set.

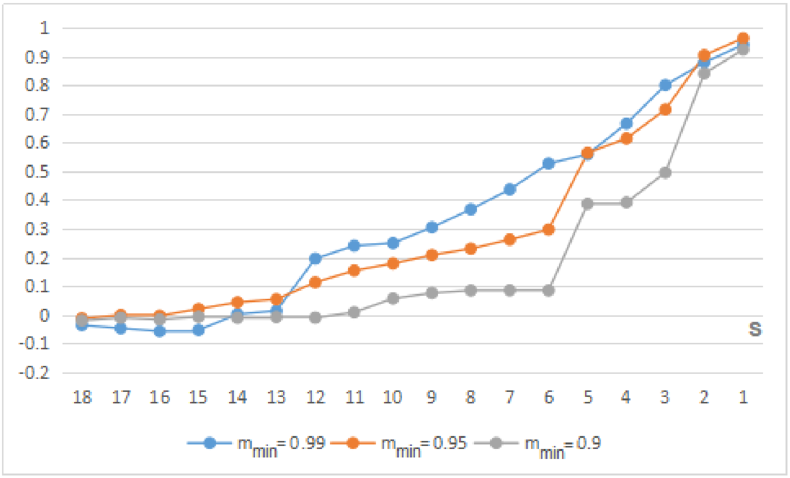


FIGURE 3. The reduction in total mining time when applying *AGRARM* on Tomcat and considering $s_{min} = 1$, $c_{min} = 0$ and $m_{min} \in \{0.99, \ 9.95, \ 0.9\}$

The results illustrated in Figure 3 have been obtained by imposing, besides the condition of a minimum support threshold $s_{min} = 1$, minimum membership thresholds, thus renouncing at also using a minimum confidence threshold to condition the *interestingness* of a *GRAR* (i.e. c_{min} has been set as 0). We successively initialized the minimum membership threshold with the following values: 0.99, 0.95 and 0.9.

In Figure 4 we give the reductions obtained by considering the minimum support threshold $s_{min} = 1$ and by varying both the minimum confidence and membership thresholds. We successively considered $c_{min} = 0.99$ and $m_{min} = 0.95$, $c_{min} = 0.95$ and $m_{min} = 0.9$ and, as a third setting, $c_{min} = 0.97$ and $s_{min} = 0.5$.

From both figures we can deduce that the time required by *AGRARM* decreases as the number s of newly added attributes decreases. Consequently, the adaptive algorithm we propose proves to be significantly more efficient than *GRANUM* applied from scratch when s is relatively small.

In order to strengthen the finding according to which *AGRARM* really makes the mining process more efficient when data is enlarged with relatively few new attributes, we comparatively tested it on two more data sets.

We present in Figure 5 the time reductions obtained on *Ar* data set when considering $s_{min} = 1$ and various values for c_{min} and m_{min}.

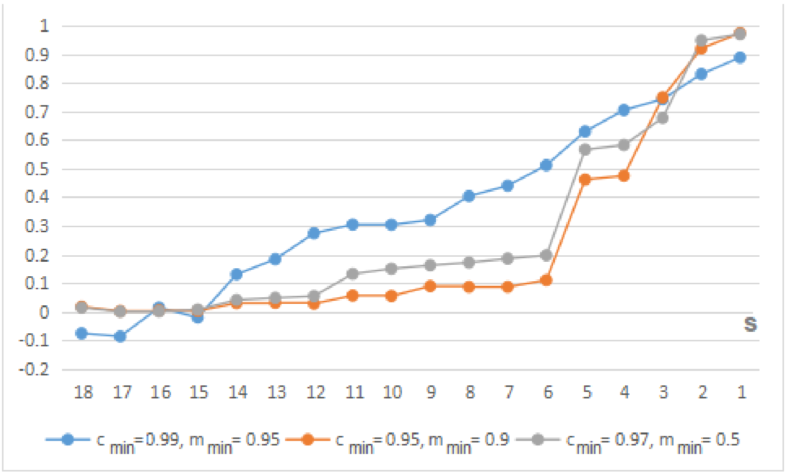


FIGURE 4. The reduction in total mining time when applying *AGRARM* on *Tomcat* and considering $s_{min} = 1$ and $(c_{min}, s_{min}) \in \{(0.99, 0.95), (0.95, 0.9), (0.97, 0.5)\}$

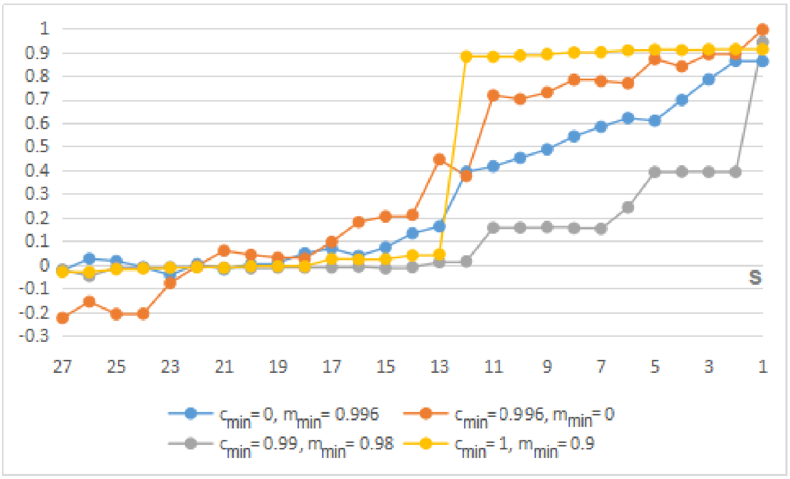


FIGURE 5. The reduction in total mining time when applying *AGRARM* on *Ar* and considering $s_{min} = 1$ and $(c_{min}, s_{min}) \in \{(0, 0.996), (0.996, 0), (0.99, 0.98), (1, 0.9)\}$

Figure 6 illustrates how the total mining time is reduced when applying, on *JM1* data set, *AGRARM* instead of *GRANUM* from scratch. In the experiments performed on *JM1* we also set the minimum support threshold, s_{min}, to 1, while varying the values for the minimum confidence and membership.

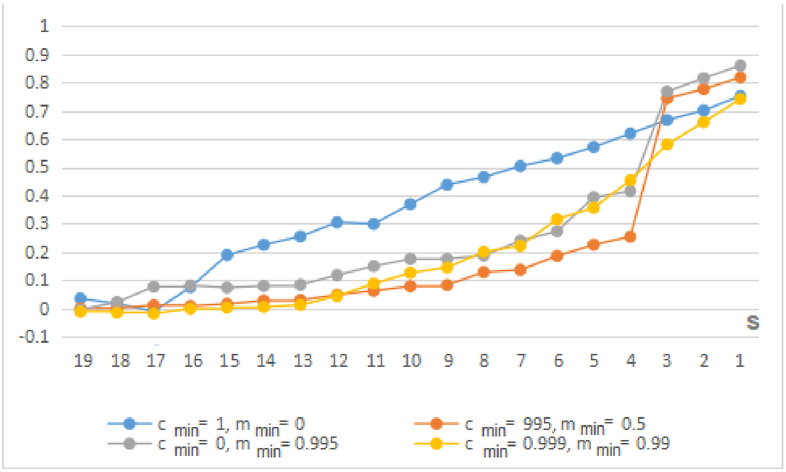


FIGURE 6. The reduction in total mining time when applying *AGRARM* on $JM1$ and considering $s_{min} = 1$ and $(c_{min}, s_{min}) \in \{(1, 0), (0.995, 0.5), (0, 0.995), (0.999, 0.99)\}$

As we can see in Figures 5 and 6, the reduction in total mining time becomes substantial when the newly added attributes are relatively few. Consequently, the results of the experiments performed on *Ar* and *JM1* also confirm the effectiveness of *AGRARM*, the algorithm we propose for adapting the interesting *GRARs* mined before extension, so as to avoid applying *GRANUM* from scratch on the extended data.

4.1. **Comparison to related work.** *AGRARM*, the adaptive mining approach introduced in Section 3, is new in the data mining literature. The existing approaches consider *non-relational* Association Rules and their adaptability refers to other aspects, except for *ARARM*, which handles *non-gradual* Relational Association Rules.

AGRARM is an adaptation of *ARARM* [8] so as to additionally consider the degree to which the rules are satisfied. This implies that the rules *AGRARM* discovers as interesting are also filtered according to a given minimum *membership* threshold (see Function *isInteresting* in Section 3) in addition to support and confidence minimum thresholds.

Apart from *ARARM*, the perspectives of the other incremental mining approaches are quite different. Still, we will briefly present several recent approaches that are somehow related to our approach, since they focus on mining dynamic data. They are incremental in the sense that the dynamics of data refers to adding new instances and not new features to the existing instances.

Nath et al. [15] provides a survey on association rule mining, insisting on the situation in which the data set is not static. The authors have highlighted the important issues and challenges of mining dynamic data, including: the multiple passes over the data set, the high number of generated candidates and the incremental behaviour of the data set.

Dhanabhakyam and Punithavalli [11] have proposed an efficient Market Basket Analysis mining method, called *Adaptive Association Rule Mining with Faster Rule Generation Algorithm* ($FRG - AARM$). The adaptability of the method refers to regulating the minimum support threshold during mining so as to attain a suitable number of rules.

Ogunde et al. [16] have introduced an *Adaptive Incremental Mining Algorithm* ($AIMA$). $AIMA$ has been designed to adapt the existing rules to the changes in the distributed databases, by mining, with the help of mobile agents, only the incremental database updates, in order to improve the response time and communication overhead.

A different incremental data mining algorithm has been proposed by Chang et al. [4]. The proposed method is based on FP-Growth and uses the concept of heap tree for incrementally updating the frequent itemsets.

A similar approach has been proposed by Yu-Dong et al [19]. The incremental association rule mining algorithm is called $PVSIFP - Growth$. The authors have incorporated in their proposal the *Improved FP-Growth* ($VSIFP-Growth$) and parallel computing based on $MapReduce$. $PVSIFP - Growth$ can discover association rules when both database increase or decrease and minimum support changes.

Li et al. [13] have proposed a *three-way decision update pattern approach* ($TDUP$) combined with a synchronization mechanism for efficiently updating and maintaining the frequent itemsets. It is based on using an additional support-based measure, so as to classify all possible itemsets into positive, boundary, and negative regions.

So, the existing adaptive approaches either rely on adapting the mining parameters [11] so as the discovered rules to be relevant, or aim the adaptation of the rules in the case of a dynamic data set, but which is extended vertically, not horizontally (i.e. by adding new data instances to it rather than adding new attributes to the existing instances) [19, 4, 13].

5. Conclusions and further work

We have proposed in the current paper $AGRARM$, a complete approach for adaptively uncovering the interesting Gradual Relational Association Rules within a dynamic data set that is extended by adding new features to it. Multiple experiments have been performed in order to comparatively evaluate $AGRARM$'s time performance. The evaluation results confirm that $AGRARM$ provided the interesting GRARs within the enlarged data more rapidly than resuming the mining algorithm $GRANUM$, i.e. applying it from scratch on the updated data set.

A first direction of further work is to further improve the efficiency of the adaptive mining process. To this effect, we aim to study possible algorithmic improvements of $AGRARM$ (like trying to generate a new candidate rule only from relevant pairs of rules, i.e. when at least one rule in the pair contains at least one newly added attribute) and also to develop a distributed version of it. We also plan to apply

AGRARM in concrete data mining tasks including incremental software defect prediction.

As an additional direction for further work, we plan to propose an adaptive-incremental approach for discovering interesting Gradual Relational Associations Rules within a dynamic data set to which both new features and new objects are added.

References

[1] Abdelhamid Boudane, Said Jabbour, Lakhdar Sais, and Yakoub Salhi. A SAT-based approach for mining association rules. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 2472–2478. AAAI Press, 2016.

[2] Alina Câmpan, Gabriela Şerban, and Andrian Marcus. Relational association rules and error detection. *Studia Universitatis Babes-Bolyai Informatica*, LI(1):31–36, 2006.

[3] Alina Campan, Gabriela Şerban, Traian Marius Truta, and Andrian Marcus. An algorithm for the discovery of arbitrary length ordinal association rules. In *DMIN*, pages 107–113, 2006.

[4] H. Y. Chang, J. C. Lin, M. L. Cheng, and S. C. Huang. A novel incremental data mining algorithm based on FP-growth for big data. In *2016 International Conference on Networking and Network Applications (NaNA)*, pages 375–378, July 2016.

[5] Gabriela Şerban, Istvan Gergely Czibula, and Alina Câmpan. Medical diagnosis prediction using relational association rules. In *Proceedings of the International Conference on Theory and Applications of Mathematics and Informatics (ICTAMI'07)*, pages 339–352, 2008.

[6] Gabriela Czibula, Maria-Iuliana Bocicor, and Istvan Gergely Czibula. Promoter sequences prediction using relational association rule mining. *Evolutionary Bioinformatics*, 8:181–196, 04 2012.

[7] Gabriela Czibula, Istvan Gergely Czibula, and Diana-Lucia Miholca. Enhancing relational association rules with gradualness. *International Journal of Innovative Computing, Communication and Control*, 13(1):289–305, 2017.

[8] Gabriela Czibula, Istvan Gergely Czibula, Adela-Maria Sîrbu, and Ioan-Gabriel Mircea. A novel approach to adaptive relational association rule mining. *Appl. Soft Comput.*, 36(C):519–533, November 2015.

[9] Gabriela Czibula, Zsuzsanna Marian, and István Gergely Czibula. Software defect prediction using relational association rule mining. *Inf. Sci.*, 264:260–278, 2014.

[10] Gabriela Czibula, Zsuzsanna Marian, and Istvan Gergely Czibula. Detecting software design defects using relational association rule mining. *Knowledge and Information Systems*, 42(3):545–577, Mar 2015.

[11] M. Dhanabhakyam and Punithavalli M. An efficient market basket analysis based on adaptive association rule mining with faster rule generation algorithm. *The SIJ Transactions on Computer Science Engineering & its Applications (CSEA)*, 1(3), 2013.

[12] David J. Hand, Fergus Daly, K. McConway, D. Lunn, and E. Ostrowski. *A Handbook of Small Data Sets*, volume 1. CRC Press, 1993.

[13] Yao Li, Zhi-Heng Zhang, Wen-Bin Chen, and Fan Min. TDUP: an approach to incremental mining of frequent itemsets with three-way-decision pattern updating. *International Journal of Machine Learning and Cybernetics*, 8(2):441–453, Apr 2017.

[14] Diana-Lucia Miholca, Gabriela Czibula, and Istvan Gergely Czibula. A novel approach for software defect prediction through hybridizing gradual relational association rules with artificial neural networks. *Information Sciences*, 441:152 – 170, 2018.

[15] B. Nath, D. K. Bhattacharyya, and A. Ghosh. Incremental association rule mining: A survey. *Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(3):157–169, 2013.

[16] Adewale O. Ogunde, Olusegun Folorunso, and Adesina S. Sodiya. The design of an adaptive incremental association rule mining system. In *Proceedings of the World Congress on Engineering 2015 - Volume I*, London, UK, 2015.

[17] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[18] Gabriela Serban, Alina Câmpan, and Istvan Gergely Czibula. A programming interface for finding relational association rules. *IJCCC*, I(S.):439–444, June 2006.

[19] Guo Yu-Dong, Li Sheng-Lin, Li Yong-Zhi, Wang Zhao-Xia, and Zeng Li. Large-scale dataset incremental association rules mining model and optimization algorithm. *International Journal of Database Theory and Application*, 9(4):195–208, 2016.

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Kogălniceanu 1, Cluj-Napoca, 400084, Romania
Email address: diana@cs.ubbcluj.ro