Verifying properties of RefactorErl-transformations using QuickCheck

E. Deckers
e.deckers@student.ru.nl
Department of Computer Science, Radboud University
Nijmegen the Netherlands

August 24, 2010

Supervisors:
Prof. Dr. Dr.h.c. Ir. Rinus Plasmeijer
Prof. Dr. Ben Dankbaar
Dr. Pieter Koopman (reader)

Research number (RU): 641
Project ID (ELTE): KMOP-1.1.2-08/1-2008-0002

Abstract

At the moment there exists no official tool for refactoring Erlang code. To this end the Programming Languages and Compilers department of the Eötvös Loránd University in Budapest is working on such a tool called RefactorErl. It is of utter importance that this tool is correct, or it might cause defects in critical systems, and should therefore be thoroughly verified (e.g. testing, formal proofs). To this end I have researched how to more strictly define the system's current natural language specification, which decreases room for misinterpretation and enables for new means of testing (e.g. formal proofs). Also, in order to check applicability of model based testing, I defined a few model based tests using QuickCheck, which have revealed a few minor problems. All-in-all the paper gives a structured and thorough testing approach that can be applied in the future to increase the quality of the code.

Contents

List of Tables

List of Figures

List of Snippets

1 Preface

1.1 Thesis structure

This thesis was written as a part of my Computer Science study. Its focus is on the functional programming language Erlang, and the technique of refactoring: changing a source code's structure, without changing its semantics (this will be explained in more detail later).

There is also a smaller management aspect to this thesis, for this is my Minor study. It consists of an introduction to the Ericsson company, the developer of Erlang, and to (the development of) the language itself.

The thesis is divided in two parts, which can be read independently. The first part addresses the management aspect of the research, and is followed by the second part that is focused on the main (Computer Science) research.

1.2 Context of research

I conducted my research at the Eötvös Loránd University in Budapest, where the department of *Programming Languages and Compilers* is working on RefactorErl, a refactoring tool for Erlang.

This research was started in 2006 and is sponsored by Ericsson. To my knowledge, it is one of the three major projects (world-wide) that are trying to achieve the same goal. The others are Wrangler and Tidier, on which I will elaborate more in the Computer Science part of this thesis.

Elroy Jumpertz, a fellow student from the Radboud University accompanied me in the research. But, even though we have both been working on the same subject –testing correctness of RefactorErl– we both had our own way of approaching the subject.[3]

1.3 Thanks

I would like to thank Rinus Plasmeijer and Ben Dankbaar for being my supervisors and advisors; Zoltán Horváth and the RefactorErl project members for having me over and helping me with my research; Bálint Fügi for arranging excellent accommodation; and Pieter Koopman for being the second reader of this thesis.

Also thanks to Emil Megyesi for helping a lot with arranging all the paperwork, and being a great guide; Lajos Kalmár for being my translator on several occasions; and Elroy Jumpertz for our collaboration in the research and for being good company.

Part I
Management Research

2 Ericsson

2.1 Introduction

This first part of the thesis starts off with a short summary of what kind of company Ericsson is today. It then proceeds by explaining how Ericsson became this company, by describing the events in history that significantly influenced the company in some way. Aside from background information on the company itself, the markets it operates in are described to some extent.

A significant part of the introduction to Ericsson is courtesy to the Ericsson History website,[4] especially section 2.3. I will therefore not explicitly refer to this website again.

2.2 Company today

Ericsson is a world-leading provider of telecommunications equipment and related services to mobile and fixed network operators globally. Over 1.000 networks in more than 175 countries utilize Ericsson's network equipment and 40 percent of all mobile calls are made through its systems. It is one of the few companies worldwide that can offer end-to-end solutions for all major mobile communication standards.[5]

These 'end-to-end solutions', are the products and services Ericsson delivers over the entire spectrum of the 'communications' market, which it divides in *networks*, *enabling new media through technology*, *global services* and *mobile communications*. A description of these categories will follow after the explanation of the global organizational structure of Ericsson in the following section.

2.2.1 Organizational structure

Figure 1 shows the current organizational structure of Ericsson. This is divided into several paragraphs, each describing one of the elements displayed in the figure.

Group Functions Group Functions coordinate the Ericsson's strategies, operations and resource allocation and define the necessary directives, processes and organization for the effective governance of the Group. The Group Functions are: Communication, Finance, Human Resources & Organization, Legal Affairs, Sales & Marketing, Strategy & Operational Excellence and Technology. The Group Functions also manage common units like Ericsson Research, IT and Shared Service Centers. The heads of Group Functions report to the CEO.[6]

Business Units Business units are innovators, developers and suppliers of products, services and customer offerings. Business units define business and product strategies. Business units are responsible for the profitable growth and consolidated results within their respective areas. The business units reflect the

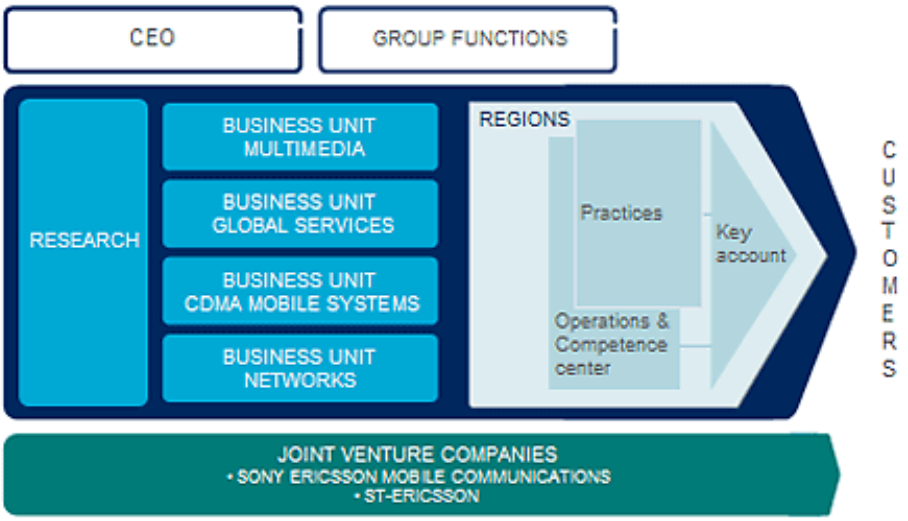


Figure 1: Ericsson's organizational structure

product- and service structure of the business: Networks, Global Services and Multimedia. Business unit heads report to the CEO.[6]

Regions Until 2009 Ericsson used to have 23 Market Units which where reorganized into 10 Regions.[7] They are marketing and sales channels and the company's representatives in the local market environment. Regions define customer strategies. They manage the complete customer relations ranging from marketing to after-sales and support activities. Heads of market units report to the CEO directly or via a selected member of the Group Management Team (i.e. president, CEO and heads of the business units).[6]

Joint ventures In certain areas, Ericsson has chosen to work with joint venture partners. The mobile handset partnership with Sony Corporation in Sony Ericsson Mobile Communications has been in operation since 2001.

In 2009, Ericsson signed a joint venture agreement with STMicroelectronics for mobile platform technology and wireless semiconductors.[6]

2.2.2 Networks

Network infrastructure is what provides the fundamentals so that people can communicate.[8] Its core network products are antennas, transmitters, switching systems, and other gear used to build wireless networks.[9]

2.2.3 Enabling communication using new media

When the company started out in the 19th century as the supplier of telephone equipment, communications basically consisted of voice conversations. Nowadays, communication is no longer limited to voice, but also includes the exchange of media like photos, videos, and IPTV.[8] Ericsson develops products to bring these latest forms of media to their customers.

Ericsson's IPTV-solution for example includes (among others): video processing (including advanced compression), media servers for on-demand content distribution, CA/DRM, QoS monitoring, TV applications, IPTV Home Environment (including set-top boxes).[10]

And Ericsson's *Personalized Greeting Service* (PGS) is an example of how the company innovates with the availability of new media on different platforms. PGS gives users the ability to replace their traditional ringback tone –the tone the calling party hears before the phone is answered or voicemail activates – with a music or video clip, a picture or an advertisement.[11]

2.2.4 Global services

Ericsson not only develops the networks and provides the technology, it also contributes with everything that is needed to get them up and keep them running. This includes consulting, systems integration and education & support services.[8]

2.2.5 Mobile communications (Sony Ericsson)

Sony Ericsson Mobile Communications is a joint venture, combining Ericsson's technology leadership with Sony's consumer electronics expertise.[8] The company was established in 2001, as a provider of multimedia devices such as mobile phones, accessories and PC cards (Section 2.3.8).

As a result of this joint-venture Ericsson and Sony no longer individually produce mobile phones.

2.3 Company's history

2.3.1 Founding Ericsson

Ericsson was founded in Stockholm in 1876 by Lars Magnus Ericsson (LM Ericsson) as a telegraph equipment repair shop, with 5 employees. But in the same year expanded its business into designing, manufacturing and selling its own telephone equipment.

Even though it started out as such a small company, it served very big customers right from the start, such as the Swedish PTT (Telegrafverket, or Televerket as it was later known), the state railways and a number of private railway companies. This was due to the expansion of the railway net and increase of the use of the telegraph at that time.

Ericsson's first step into the manufacturing and selling business was the deliverance of two signal telegraphs of its own design to the state railways, on September 18th of its first year of existence. This also was the start of the continuous expansion of the company, which resulted in Ericsson moving to larger premises before the end of the year.

During this time, 1876 – 1877, the telephone was invented and patented by Abraham Graham Bell. Soon early adopters, who connected their phones directly (i.e. a direct cable rather than through a telephone network) would bring their telephones to Ericsson for repair, which enabled LM to have a peek inside. He would use this knowledge, improve upon it, and sell these phones of his own design for a lower price than the originals. He was allowed to 'copy' Bell's phones, because Bell did not patent the telephone in the Nordic countries; a good fortune which would befall Ericsson more often throughout the years.

The company could however not get the maximum profit out of it, because there was no telephone network in existence in Sweden at that time. And when such network was finally introduced, Bell hastily took over the company that introduced it, and only allowed its own equipment to be used on the network.

To compete with Bell, which now had a very strong position, Ericsson had to stand out somehow. Therefore Ericsson improved upon the design of the current Bell phones, which resulted in the 'helical microphone'. These phones proved to be of vital importance in a fight between Ericsson and Bell for who was to build and operate a telephone network in the city of Gävle.

At the beginning of 1881 Bell offered to build this network, but in response, J.W. Sundberg, a Gävle businessman, proposed the use of Ericsson products rather than Bell's. Therefore, several products of both companies were compared, and the final verdict, as described in the local newspaper was that both systems worked excellently. However Ericsson's telephones were "simpler, more robust and more handsome", and last but not least its products were cheaper. This resulted in the city of Gävle to do business with Ericsson instead of Bell. Not only would Gävle use Ericsson's phones, but also an Ericsson telephone exchange. The word about this decision spread fast, and as a result not only Sweden cities, but also Bergen (Norway) for example, decided to do business with Ericsson rather than with Bell.

2.3.2 Stockholms Allmänna Telefon

In 1881 Henrik Thore Cedergren and LM Ericsson met each other for the first time. Cedergren was a jeweler's store owner with a an engineering degree and 'a nose for business'. He was also a Bell subscriber and worked out that the company was both overcharging and that it would be more successful if it charged less. He contacted the Bell company to propose this change in business approach, but the company was not interested. Cedergren had since then been looking for a way to start competing in the telephone business.

When the two met, Cedergren proposed his ideas to LM Ericsson, who was a little hesitant at the beginning, but nonetheless decided to sign a partnership agreement. This resulted in Cedergren establishing Stockholms Allmänna

Telefonaktiebolag (SAT) in 1883, which offered "public telephone connections at a lower price and the use of Swedish equipment" and "telephone lines in every building and for all the tenants in them". Ericsson was the sole supplier of equipment to SAT, and was not allowed to supply equipment to other Stockholm telephone companies.

This approach worked out very well, and the company would soon gather more subscribers than Bell. Attempts by the latter company to take its position back were in vain, and in 1891 Bell gave up and sold its network to SAT.

2.3.3 Going abroad

In 1896 Cedergren founded AB Telefonfabriken in which SAT was the main shareholder. Its purpose was to compete with Ericsson, and to compete on markets abroad. Therefore Ericsson needed a new partner, but other big competitors already had their own production companies, and hence Ericsson's market declined immensely.

Therefore, LM Ericsson decided that it was time for his company to go abroad as well. He employed several people who would make long journeys abroad in order to expand the company, which resulted in Australia and South-Africa becoming the most important markets for Ericsson for several decades.

Due to new strategy of focusing on abroad markets, Ericsson was considering not only exporting products, but moving production to other countries as well. Russia seemed like the most logical candidate, for Ericsson had been exporting equipment there for years, and a lot of Swedish entrepreneurs were active there at the time. So it happened that in 1896 LM Ericsson bought ground in St. Petersburg for the erection of its first foreign a factory and accommodation. By 1900, exports accounted for about 90 percent of Ericsson's total sales.[12]

In that same year, there was a big fight between several companies for who would become the main supplier of equipment to big Russian cities such as St. Petersburg, Moscow and Warsaw. Ericsson won, which made Cedergren conclude that his factory, even if it expanded, would not have a reasonable chance of supplying all the equipment required for the two cities. This resulted in Ericsson and Cedergren rebuilding their relationship, and the former taking over AB Telefonfabriken.

2.3.4 Lars Magnus resigns

Two weeks before the decisive events in St. Petersburg, Lars Magnus asked the company's board to relieve him of the post of managing director. One of the reasons he offered was his health. The board agreed, but also offered LM the right to change his mind: he could return to his previous post whenever he liked. LM never did change his mind though, but instead resigned as chairman of the board on February 26, 1901.

LM remained the principal shareholder for some time, but eventually sold his shares. In 1903, LM left both the board and the company for good.

2.3.5 World War I and the October Revolution

During World War I Ericsson suffered as hostilities cut off most of its foreign markets. The cost of raw materials was at record-high levels, and the war made the transport of both raw materials and finished products difficult. Exports as a proportion of total sales were decreasing and concentrated to neutral countries plus Russia, while markets in Australia, South Africa, Egypt and China were lost.

Russia alone accounted for 46 percent of the Stockholm plant's sales. In addition, the Russia plant was running at full speed and succeeded in maintaining satisfactory production levels despite problems with raw materials. Production plants had been established in several countries. After the Russian plant, the most important of these was in Great Britain. The plants in France and Austria-Hungary were taken into operation for other purposes than telephone manufacturing. The French plant produced war materials, while the plant in Budapest was used as a military hospital. In Buffalo (US), magnetic ignition equipment was now being produced instead of telephone equipment.

The Russian market dissolved in 1918 due to the October revolution – the new Bolshevik government nationalized Ericsson's Soviet operations, seizing all of its Russian assets, without any compensation. Despite these setbacks, the company's sales continued to rise.[12]

2.3.6 Crisis and World War II

Ericsson's most important markets fell when the full force of the 1930s depression was felt. There was a decline in world demand for telephones and switches, and the company's foreign operations saw their sales almost halved. In order to survive this situation Ericsson sought for other products which could be produced using the existing equipment and personnel.

A diversification of production was implemented and new markets found, primarily within Sweden. This resulted in Ericsson producing among others: electricity meters, measuring instruments, railway signaling equipment, light bulbs and electronic tubes for telegraphy, telephony and radio.

Another reason for the decline in sales was that the war limited Ericsson's exporting capabilities. But then again, it produced new opportunities in the domestic market. In order to equip the Swedish defenses at the height of the war, the Swedish engineering industry was mobilized. Swedish defenses were almost completely equipped with domestic materials, and nearly a quarter of the Swedish engineering facilities' capacity was used directly for the war effort. The figure for Ericsson was approximately 20 percent during the years 1939-1944.

After the war, Ericsson concentrated on its core telephone manufacturing business, and export markets played their traditional leading roles. During the war, domestic orders had accounted for a peak 80 percent of all sales, but this began to decrease steadily in 1946. By 1973, the ratio would be reversed; exports would account for about 75 percent of Ericsson's sales, just as they had during

the early 1920s.[13]

2.3.7 Wireless communication[1]

Wireless communication is an important aspect of Ericsson's business, and it has been since the dawn of the 20th century when, driven by their interest of the relatively new radio technology, Ericsson together with four other companies founded SRA (Svenska Radioaktiebolaget) in 1919. The other most important owners were African Stock Exchanges Association (ASEA), Aktiebolaget Gasaccumulator (AGA) and Marconi. SRA produced radio-receivers under the Radiola brand.

The field kept on evolving, resulting in the emergence of mobile transmitters in the 1930s and Edwin Howard Armstrong demonstrating frequency modulation (FM), which enabled digital data-transmission over radio. In the 1940s the first interconnection of mobile users to public switched telephone network (PSTN) took place. And in 1956 the world's first fully automatic mobile phone system, called Mobile Telephony A (MTA), was developed and introduced by Ericsson. Requiring no manual control of any kind, users only had to dial numbers on a telephone to make a call.[15]

After the user-base of mobile communications grew over 1.4 million people in the 1960s awareness grew at SRA that the company should focus on communications radio. Ericsson, which was the principal owner since 1927, had established a strategy that focused on telephony and communications. Therefore radio and television production were sold to AGA, which also acquired the Radiola brand.

During the 1970s, in Chicago, Bell Labs installed the first commercial cellular network. A few years later, in 1981, Ericsson launched the first fully-automatic mobile telephony system, Nordic Mobile Telephony (NMT). Ericsson was also the driver of the GSM global standard, which was introduced in 1991 as a Pan-European mobile system and later evolved into a global system.[15]

2.3.8 Sony Ericsson

Ericsson and Nokia (alongside Motorola) were the biggest mobile phone equipment manufacturers of the nineties (Table 4), and competed fiercely. Both had a contract with a Philips plant in Mexico, that added together counted for 40% of its production.[16]

On March 17th, 2000 a lightning bolt caused a fire at a clean-room in the Philips plant, which made millions of chips unusable. Philips told both Nokia and Ericsson that it would take only a week before production could be taken up again, and that they would get priority over other customers.

Unlike Ericsson, Nokia did not have faith that Philips would be able to make such a quick recovery and immediately started to arrange other sources for the production of these particular chips, and even modified some phone designs to work with different chips.

[1] The bigger part of this section was taken from[14]

It was the end of March when Ericsson started to realize the consequences this disaster would have. And on July 20, 2000, Ericsson reported that the fire and component shortages had caused a second-quarter operating loss of $200 million in its mobile phone division. Six months later, it reported divisional annual losses of $1.68 billion, a 3% loss of market share, and corporate operating losses of $167 million. This resulted in Ericsson outsourcing of cellphone manufacturing to Flextronics; Several thousand jobs were eliminated.[16]

After the reports in July Ericsson secretly started negotiating with Sony about a joint-venture to make mobile phones. Sony had been trying to obtain a market share in the mobile phone industry for several years, but remained a marginal player with its share of about 1 percent in 2000. Therefore, in October 2001, Sony and Ericsson started their joint-venture and seized their individual production of mobile phones.

2.4 Products and services throughout the years

2.4.1 Servicing and manufacturing equipment

Ericsson started out in the first year of its existence as an electronics repair shop, but soon moved on to develop its own equipment. This was the result of both the introduction of the telephone and inherent networks, and expansion of the railway system. The first products manufactured by the company were two telegraphs, which were sold to the state railways.

2.4.2 Switchboards and telephones

From 1884, Ericsson produced switching equipment, alarm telegraphs, wall- and desk phones. Many of Ericsson's products became very known for their design, like switchboard phones innovatively equipped with a receiver that included both the microphone and the ear piece in a single unit.

Famous is also the Ericofon (figure 2, or "Cobra" as it was more commonly known due to its 'snake-like' appearance; it created demand Ericsson could not fulfill. And yet another Ericsson product known for its design is the Dialog telephone (figure 3), which was sold over several decades to millions of people from its introduction in 1962.

2.4.3 Computer-controlled switching systems

In 1970 Ellemtel was founded, a Research & Development subsidiary of Ericsson and Televerket. Its core assignment at that moment was to develop the AXE (most sources agree that this means Automatic eXperimental Electronic) switch, which would be the digital successor of the AKE analogue telephone exchange.

Both Ericsson and Televerket had different interests in the system. The latter party desired a switch that was specially designed for and adapted to the Swedish telephone network's characteristics. Ericsson was more interested in export markets, which demanded the system to suit as many international

Figure 2: Ericofon (or Cobra)

telephone systems as possible. This resulted in a very modular system, which was quite innovative for the time.

AXE was finished in 1978, and was used all around the world. It was the most advanced and flexible switching system of the 1980s (mostly due to its modularity), doubled the company's market share and facilitated the company's entry into the important US market.

Moreover, it made Ericsson a major player in mobile telephony at an early stage (nowadays mobile communications still are Ericsson's core business). This resulted in Ericsson by 1992 having received orders for nearly 400 AXE switching stations for mobile telephony, corresponding to 40 percent of the global market.

2.5 Most significant products nowadays

This section explains what Ericsson's most signification products are nowadays. The section is split-up in several sections of which each addresses one of these signification products.

2.5.1 Networks

Figure 4 describes the products Ericsson offers in the networks-segment. This section is divided into the categories displayed in this figure.

Figure 3: Ericsson Dialog with number disc

Customized services The Customized Services part of the network architecture contains functions that provide services for consumer and business users, based on a provider's network and IT capabilities.

Standardized services Standardized Services enable a provider to offer standard user services such as telephony, SMS and MMS as well as IMS-based services such as presence, messaging and multimedia telephony.

Multi-access edge Enables the ability to access all kinds of services on any terminal, across multiple different fixed and mobile access networks (such as, for example, video-telephony sessions between fixed and mobile terminals).[17]

Radio access/Mobile broadband By *broadband* in this respect, the use of 3rd generation networks (3G) is referred to (e.g. UMTS,WCMDA/HSPA). These networks contain a much broader frequency band than earlier generations like GSM (2G) and NMT (1G), which most importantly increases data transfer speed. About 50 percent of the operators with commercial mobile broadband networks –of which Ericsson is the world's principal supplier– use Ericsson products. The introduction of mobile broadband in the networks has substantially enhanced the end-user experience, which explains the current massive uptake of mobile broadband services around the world.

However, the majority of today's mobile users are still served by GSM and every month several million people join the GSM community. Ericsson is one of the few vendors continuing to develop new technology for GSM. Fortunately, Ericsson GSM and WCDMA/HSPA (3G) networks share a common core network, indicating that previous investments are protected as operators shift from voice- to multimedia centric networks.[18]

Wireline access/Fixed broadband Besides mobile networks, Ericsson targets the fixed broadband market. The acquisitions of Marconi, Redback Net-

Figure 4: Ericsson network products and services

works and Entrisphere strengthened its offering in access, transport and Internet Protocol (IP) networks.[18]

Transport Mobile transport solutions collectively provide comprehensive IP (Internet Protocol) connectivity for all mobile traffic, from the radio base station to the mobile interconnection with external networks.

Operation and business support Support for a provider's operations and business in the areas of network and resource management, service offerings, revenue streams, and customers and partnerships.

Market In the 1980s large deliveries of mobile systems were made in the US. Paging systems were another profitable product area in which several new products were introduced. Back then its market share was about 45%.

With a share of 31.1% Ericsson still plays a prominent role in this market. Its most important competitors in this market are Huawei Technologies Co., Ltd. (Huawei), Nokia Siemens and ZTE Corporation (ZTE), which respectively account for 27%, 16% and 15% of the market (Figure 5).[2]

Networks account for about two-thirds of Ericsson's net sales. Ericsson's network solutions include radio access network, (radio base stations for GSM, WCDMA, HSPA, LTE) core network solutions, (like softswitch, IP infrastructure, IMS, media gateways), transport solutions (like microwave radio and optical fiber solutions) and fixed access solutions for copper and fibre.[18]

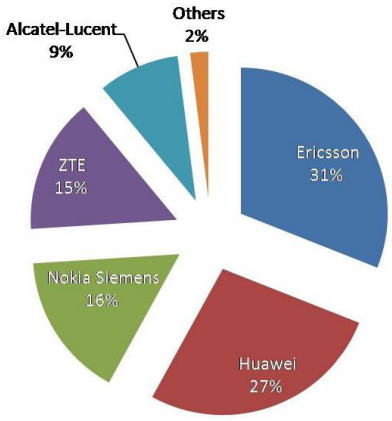


Figure 5: GSM global market shares 2009[2]

2.5.2 Services

Sales of Ericsson's services represent more than 22 percent of net sales. These services include network roll-out, consulting and education, systems integration and customer support.

Ericsson is the industry leader in managed services, managing networks that serve more than 120 million subscribers.[18] These services include activities such as designing, building, planning, operating and managing day-to-day operations on behalf of a customer.[19]

2.5.3 Multimedia

Products in this category include service layer products, revenue management systems, enterprise solutions and mobile platforms. Sales of multimedia represent about 8 percent of net sales.[18]

3 Erlang

3.1 What is Erlang?

Erlang is a functional language with *strict evaluation*, *single assignment*, and *dynamic typing*. The first version was developed by Joe Armstrong in 1986, and it was designed for writing concurrent programs that "run forever.";[20, 21] It uses *concurrent processes* to structure the program in compliance with the *Actor model*.

3.1.1 Basic language characteristics

Functional language There exist several programming paradigms which a programming language can support. Well-known are the procedural and object-oriented programming (OOP) paradigms; the first is a *imperative programming* paradigm and the second is a *structured programming* paradigm.

Another, less-common, programming paradigm is the functional programming paradigm, which in turn is a type of *declarative programming* paradigm. In languages that support the declarative programming paradigm the programmer describes *what* the computer should do. This opposes the imperative programing paradigm, in which the programmer tells the computer *how* to do something.

As an illustration imagine an algorithm which picks a particular fruit from a virtual basket. In an imperative language a programmer would instruct the computer what to do step-by-step:

- pick one fruit from *fruit* basket, if it is an apple put it in our *apple* basket, otherwise discard the fruit; repeat until *fruit* basket is empty;
- pick apple from *apple* basket, if it is yellow, put it in our *yellow apple* basket, otherwise discard; repeat until *apple* basket is empty.

The declarative variant doing the same thing would rather tell the computer *what* to do:

- transfer all apples from *fruit* basket into *apple* basket;
- put all apples from *apple* basket that are yellow in the *yellow apple* basket

That is the general idea of declarative programming, of which functional programming is one type. It treats computation as the evaluation of mathematical functions –hence the name– and avoids state and mutable data. By avoiding state a functional language guarantees that a every function always gives the same output given a particular input (except functions with side-effects i.e. input from the environment). And the immutable data property ensures that a variable can only be assigned once (single-assignment). These properties makes it easier to verify, and parallelize programs for example.

Strict evaluation This property means that arguments to a function are always evaluated completely before the function is applied, which happens in most programming languages.

Haskell, another functional programming language, is known for its *lazy* evaluation, which is the opposite of strict evaluation.

Lazy evaluation would cause problems with Erlang though, because with lazy evaluation the compiler (the program translating the Erlang-code to machine-code) determines the best evaluation order of Erlang expressions, regardless of in what order the programmer wrote them. Because Erlang functions can have side-effects (i.e. can influence other functions) reordering Erlang functions can result in undesired results though.

3.1.2 Concurrent programming

What is concurrent programming? In a concurrent program, several streams of operations may execute concurrently.[22] Each stream of operations executes as it would in a sequential program. This enables the programmer to split op large problems into smaller ones and to compute them simultaneously on multiple processors, cores or pc's for example.

Threads One common way of concurrent programming is called threading. A single process might contain multiple threads; all threads within a process share the same state and same memory space, and can communicate with each other directly, because they share the same variables.[23]

Because threads can interfere so easily with each other, they can easily cause a lot of difficult issues in a program.

Processes Processes are independent execution units that contain their own state information, use their own address spaces, and only interact with each other via interprocess communication mechanisms. They can be regarded as separate programs.

Each separate process can compute a part of the larger problem at hand, and they cannot (directly) interfere with each other due to the mentioned characteristics.

Processes in Erlang conform to the Actor model, in which each process is an *actor*. This is an entity that has a mailbox and a behavior. Messages can be exchanged between actors, which will be buffered in the mailbox. Upon receiving a message, the behavior of the actor is executed, upon which the actor can: send a number of messages to other actors, create a number of actors and assume new behavior for the next message to be received.[24]

3.1.3 Hot code swapping

Erlang allows a programmer to change the code of a program as it runs. This enables bug fixing in a system without it having any down-time. In contrast,

the majority of other languages requires a running program to be terminated before it can be updated.

3.2 Why was Erlang developed?

Over the years Ericsson has developed several programming languages, among others PLEX, a high-level programming language specifically designed for AXE telephone systems.[21] It was based on Eriplex, a language previously developed by Ericsson for its AKE switch. However, PLEX was designed to build applications specifically for the AXE switch, and it did not meet all demanded criteria required in the 1980s (which will be described shortly).

Therefore, Ericsson started looking for alternatives, which resulted in the initial development of Erlang in 1986 at the Ericsson Computer Science Laboratory. Erlang was designed with a specific objective in mind: "to provide a better way of programming telephony applications." At the time telephony applications were atypical of the kind of problems that conventional programming languages were designed to solve.[21]

Telephony applications are by their nature highly concurrent: a single switch must handle tens or hundreds of thousands of simultaneous transactions. Such transactions are intrinsically distributed and the software is expected to be highly fault-tolerant.

Telephony software must also be changed "on the fly," that is, without loss of service occurring in the application as code upgrade operations occur. Telephony software must also operate in the "soft real-time" domain, with stringent timing requirements for some operations, but with a more relaxed view of timing for other classes of operation.

3.3 Development of the language

In order to solve the requirements mentioned in the previous section, the company experimented with > 20 languages through the years 1982 – 1985. The conclusion was that the language must be a very high level symbolic language in order to achieve productivity gains. This left only: Lisp, Prolog and Parlog.[25]

Through 1985 – 1986 Ericsson experimented with these languages, and concluded that the language must contain primitives for concurrency and error recovery, and the execution model must not have back-tracking, which ruled out Lisp and Prolog. It must also have a granularity of concurrency such that one asynchronous telephony process is represented by one process in the language, which ruled out Parlog. Therefore Ericsson had to develop its own language with the desirable features of Lisp, Prolog and Parlog, but with concurrency and error recovery built-in.[25]

In 1987 the first experiments were conducted with Erlang, after which it was tried on a real-world problem in the ACS/Dunder project. This consisted of the prototype construction of PABX functionality by external users.[21, 25]

The ACS/Dunder project produced its final report in 1989. The conclusion of this report was that, even though Erlang increased programming productivity

by a factor 10 in comparison to PLEX (section 3.2), it lacked performance and needed to execute 40 times as fast, to be useful in practice.[21, 25]

In 1991, due to some clever rewritings of the virtual machine, a faster implementation of Erlang is released to users, which also has more functionality. This new version was 70 times as fast as the initial Erlang prototype, easily bypassing the 40 times requirement. However, this requirement turned out to be a miscalculation: it should have been 280 times as fast.[21, 25]

As a result Turbo Erlang (christened BEAM) was developed and released in 1993, increasing the speed of the 1991 Erlang implementation about 3 to 10 times, practically complying with the speed requirement.[21]

Erlang was then used for a few years, but in 1998 banned certainly within Ericsson Radio AB for new product development. A few months afterward, Erlang was made Open Source by Ericsson. Following the ban the 60 Erlang developers resigned from Ericsson to form the new company Bluetail AB.[21, 25] This company was taken-over a few years later by Alteon Web sytems, which was in turn taken-over by Nortel Networks.

3.4 Opening-up the source

The reason for banning Erlang for usage in new products was that: "[Using] a proprietary language [implied] a continued effort to maintain and further develop the support and the development environment. It further [implied] that [Ericsson could not] easily benefit from, and find synergy with, the evolution following the large scale deployment of globally used languages."

Following the Erlang ban, interest shifted to the use of Erlang outside Ericsson. It was concluded however that, in these times of vast amounts of freely available software, no-one would be interesting in buying Erlang. Therefore, it was released as open source to the public, which stimulated the long-term growth of Erlang.[21]

Ericsson makes money from selling a commercial variant of the language, which is essentially the same as Open Source Erlang. Differences are that Commercial Erlang comes with a support agreement and is more stable because it is tested more thoroughly and all experimental code is excluded.[26]

3.5 Companies using Erlang

[27] The largest user of Erlang is Ericsson. Ericsson uses it to write software used in telecommunications systems. Many (dozens) projects have used it, a particularly large one is the extremely scalable AXD301 ATM switch. AXD301 has several hundred people working on it and the code volume has reached about 850 KLOC of Erlang (and 1 Mloc of C/C++).

Some of the other companies using Erlang are:

- *Bluetail/Alteon/Nortel*: Distributed, fault tolerant email system, SSL accelerator;
- *Facebook*: Chat backend;

- *Finnish Meteorological Institute*: Data acquisition and real-time monitoring;
- *T-Mobile*: Advanced call control services.

4 Tables

	1992	1993	1994	1995	1996	1997	1998	1999	2000
Nokia									
Sales ($ mil)	3,451	4,079	6368	8,400	8,446	9,702	15,553	19,954	28,608
Net Income ($ mil)	(78)	132	632	509	745	1,154	2,043	2,601	3,709
Employees	26,700	25,800	28,600	31,948	31,723	36,647	44,543	51,777	60,000
Ericsson									
Sales ($ mil)	6,644	7,622	11,342	14,902	18,291	21,219	22,760	25,267	29,026
Net Income ($ mil)	68	340	531	813	1,033	1,511	1,609	1,423	2,230
Employees	66,232	69,597	76,144	84,513	93,949	100,774	103,667	103,290	105,129
Motorola									
Sales ($ mil)	13,303	16,963	22,245	27,037	27,973	29,794	29,398	30,931	37,580
Net Income ($ mil)	453	1,022	1,560	1,781	1,154	1,180	(962)	817	1,318
Employees	107,000	120,000	132,000	142,000	139,000	150,000	133,000	121,000	147,000

Table 1: Financial overview, major mobile telecommunication manufacturers during the 1990s (Excerpt from[1])

Part II
Computer Science research

5 Introduction

5.1 Subject and structure

This introduction section will give a quick impression of the main topics of research and tools involved, as will it elaborate on the core problem this thesis addresses, and the strategy applied to diminish it.

5.1.1 Core subject

Prevailing subject of this thesis is RefactorErl,[28] an instrument being developed by the department of *Programming Languages and Compilers* at the Eötvös Loránd University, for refactoring *Erlang/OTP*[29] (hereafter referred to as 'Erlang') source code.

An essential aspect of any refactoring tool is its correctness, which must be proved as exhaustively as possible. Conventional means of achieving this include: *formal proofs*, *unit testing* and *Model Based Testing* (MBT).

RefactorErl is being tested by means of unit tests, but this is considered to be too concise by the project's members. Hence, other means of testing are required in order to achieve a more thorough confidence in the system's correctness and its refactoring transformations in particular.

To this end I explored the possibilities of the applicability of mentioned MBT method in my research, in which the formalization of some of the transformation-specifications of RefactorErl also has an important role. The desired result being a possible approach for testing the system more thoroughly in the future.

Moreover, appendix A describes how to compare syntax of RefactorErl parse trees to Erlang parse trees, which I researched as an introduction to the subjects of refactoring and RefactorErl. This could form a basis for further research, but is not an integral part of the thesis.

5.1.2 Research structure

My research is divided 3 closely related parts: specification formalization, MBT and selecting interesting complex constructs.

Formalizing Foremost reason for formalizing the specifications, aside from all other advantages it brings along (on which I will elaborate in section 7.3), is that a test can only be as good as its specification. Hence, it is of utter importance to have these specifications defined in an unambiguous manner. Additionally it is desired –but not necessary– to formalize these specifications using an independent notation, which the bigger part of the audience is familiar with, in order to eliminate the overhead of learning a new notation. Therefore, I considered QuickCheck notation not to be a suitable candidate for formalizing specifications, if that is the only goal.

Alas, there did not exist such formalizations yet; only natural language (*edoc*) comments. So, in order to get the best out of testing, formalization was my first step in the research.

MBT After specifying three of RefactorErl's implemented refactoring's specifications, I defined some model-based properties, a concept which is discussed in more detail in section 5.6.2. Using QuickCheck (also abbreviated as QC in this thesis), an MBT instrument for Erlang, I tested these properties of RefactorErl thoroughly.

Semi-random input MBT requires input to test against, for example boundary values. I would like to test the implementation against semi-random data though in order to generate obscure test cases, which require actual Erlang-code as input. Yearning to both demonstrate the usefulness of defined properties and to uncover as many defects as possible –given a time-window too small for writing a decent code-generator– I gathered a few hard-to-refactor constructs by hand, to test against.

The intention of this structure is to demonstrate *a* possible, useful, approach for MBT-ing all transformations of RefactorErl.

5.1.3 Document structure

IMRaD This part of the thesis elaborates on aforementioned matter somewhat along the lines of the *IMRaD*[30] (Introduction, Methods, Results, Discussion) structure conventions, though I have added a Background Information section between the Introduction and Methods section where I explain about the tools used in this research. The thesis abstains from in-depth background information on Erlang and RefactorErl, which is documented extensively in the Programming Erlang book[31] and the RefactorErl tutorial respectively.[32]

Upcoming sections The method-section that follows section with background information explains in detail the methods, and reasoning behind their application in this research. The results of the application of said methods in this research, follow in the result-section that succeeds the method section.

The thesis ends with a discussion- and a conclusion section, which provide some introspection on the research.

5.2 What is refactoring?

Introduction Since this thesis focuses on refactoring, a decent introduction to the subject is desired, which is the purpose of this section.

To comprehend the subject of refactoring, it is important to realize that the quality of a computer program can be measured –among a lot of aspects– by the readability and maintainability of its source.[33] In order to write a good program in terms of these measures, it is of utter importance to keep the code well-structured (e.g. intuitive, concise, transparent). This makes the code more readable, which reduces the chance of introducing new defects,[28] because it is easier to understand the source.

The problem refactoring addresses However, when writing code, especially at the beginning of a project when code-structure tends to evolve a lot (in the case of agile software development methodologies like XP[34] it is even supposed to evolve *all the time*), and a lot of ad-hoc code will be written, the program's source can easily become very messy.

Failing to rewrite this disorganized code will most probably result in a final revision that contains severe amounts of 'spaghetti code', which the original author cannot even understand anymore after one year.

Some of the common abominations that cause this problem include: rudimentary functions (i.e. were used before, but are not referred to anymore); multiple functions that basically perform the same trick (i.e. repetitive code, which conflicts with the *DRY* (Don't Repeat Yourself)[35] principle for example); the situation where it makes more sense for a certain group of functions to be grouped in a single separate module in stead of being scattered over multiple modules; variables and functions that are assigned names which do not describe their meaning or purpose sufficiently, or at all.

It is nearly inevitable that these problems are introduced, given that as the project develops, programmers will learn and gain new insights all the time, which will make them discover room for improvement. But these improvements might conflict with, or overlap existing parts of the code, which can easily be overlooked (e.g. a lot of large projects tend to have multiple developers, which makes it impossible for a single programmer to know all the code). These conflicting parts in turn will provide room for improvement as well. This illustrates how easily inefficient or conflicting code is introduced, and the inherent importance of continually improving ones code.

Definition of refactoring Whenever a point, where a programmer finds room for improvement, is reached (e.g. simplification of a complex function, restructuring of modules), he might consider *refactoring* the code: "the process of changing a software system in such a way that it does not change the external behavior of the code yet improves its internal structure".[36]

An example of a refactoring is when one renames a function or variable and all its references: the code is changed, but its semantics remain the same. Even though this act does not change anything to the behavior, or performance of the code, it may increase its readability and therefore improve its quality. More concrete, a variable named `LastName` is much more clear than `X`, because it is descriptive of its meaning.

A lot of refactorings are very commonly applied, which is why people developed a lot of refactoring tools for all kinds of programming languages that aid in applying them. So, in previous example, instead of having to identify each and every occurrence of a particular variable by hand, these tools will do that for you and apply the refactoring.

Usefulness controversy In addition to the definition of refactoring stated in the previous paragraph I would like to mention here that it is actually a

significant point of discussion whether or not refactoring necessarily improves code. First of all this issue relies heavily on the skills of the person operating the refactoring tool. But mostly, it is a matter of opinion that it is well documented on numerous Internet forums,[37–39] so I will not delve too deep into it and just name a few common-heard arguments against refactoring and refute them.

Probably the most prominently heard argument against refactoring is 'if it ain't broke, don't fix it', but this argument only holds from a functionality perspective of 'broken'. What about quality perspectives, like: readability, modularity and style? If these do not comply with the project's requirements, then the code might be considered 'broken' too and therefore needs fixing, to which end refactoring tools can be of great help.

Another common heard argument is: 'the source code of the application to be refactored is too big, and it would be to big an effort to make it noticeably better'.[40] This problem can however easily be overcome by applying divide-and-conquer –for example by functionality–, which is pretty much the way the application was probably written in the first place.

These are just two of a long list of the arguments against refactoring, which can be found on lots of forums and blogs on the Internet, and demonstrate that the 'improvement' part of the definition of refactoring is rather subjective. I therefore will not delve into this discussion any further, and consider the definition of refactoring in this thesis as presented earlier.

5.3 Refactoring tools for Erlang

The generic nature of refactorings makes them very tedious and error-prone to apply by hand,[41] but also very apt for automation. For this reason and because refactoring is such common-practice for programmers, a lot of tools for all kinds of programming languages are being developed for this sole purpose and often even integrated into IDE's.[28, 42–46]

Unfortunately though, mature refactoring tools are quite rare in the world of functional programming languages.[43] Likewise for Erlang, which lacks any official refactoring tool as for now. To my knowledge there currently exist three official projects[44, 47] though that address this subject, one of them being RefactorErl, core subject of this thesis.

In the Wrangler project QuickCheck was also used as a means to verify the correctness of refactoring transformations, which led to the revealing of several bugs.[48] This is also very convenient for RefactorErl, because several of the ideas and properties used to check Wrangler can be incorporated indirectly in the testing of RefactorErl (section 7.4.1). For Tidier I could not find any information on whether and how the tool is tested for correctness.

The rest of this section concisely discusses the capabilities and differences of RefactorErl, Wrangler and Tidier.

5.3.1 RefactorErl

The RefactorErl research group has been working on this open source refactorer, which will be IDE-independent, since 2006.[43] In their approach they use their own Erlang parser, and the *Mnesia* DBMS for storing parse graphs. This enables them to easily query and modify the graphs, and to write them back to the original file preserving its formatting.[49]

RefactorErl currently counts 22 refactorings,[28] ranging from renaming nodes (e.g. modules, functions and variables) to heavy restructuring (e.g. moving and generalizing functions, eliminating variables and merging expressions). And it is currently integrated with the Emacs and Eclipse IDE's.[50]

Side-effects One Erlang-specific property that requires special attention in the context of developing a refactoring tool for this language, is *side-effect* handling. As opposed to a pure functional language like Haskell for example, which secludes side-effects to *monads*, Erlang is an impure functional programming language that allows every function to have side-effects. This design makes it harder to detect if a certain function has side-effects than in pure languages, and for this reason is harder to determine whether or not a particular node in a function's scope is apt for a certain refactoring.

Consider for example snippets 1 and 2 (before and after eliminating a variable respectively) which presents a situation where a problem would occur as a result of the side-effects of `random:uniform/0`, would the refactoring tool actually allow variable `Y` to be eliminated. Hence, a refactoring tool should prohibit the elimination of variables with side-effects, which is relatively easy in Haskell because all side-effect variables are 'tagged' with a particular type (e.g. `IO`) by the monads.[51] This forces the a Haskell programmer to explicitly detect and handle side-effects.

Erlang however does not entail such luxury and hence the presence of side-effects has to be deduced by other means. Because Erlang handles side-effects by secluding them to *BIFs* (Built-in Functions), RefactorErl detects whether or not a function is dirty by searching the bodies of all functions it refers to for calls to dirty BIF's or the sending of *messages* (Erlang handles concurrent programming through message-passing, which will be explained later in this section). When such call is found, the function being analyzed is marked 'dirty' as well. And finally, calls to the function being analyzed will be analyzed as well, because these calls may pass dirty functions (effectively making the analyzed function dirty as well).

It will be no surprise that this approach of detecting the presence of side-effects brings a few problems along, one of the most obvious being that it cannot be predicted if and when new dirty BIF's will be added to the language (or non-dirty BIF's become dirty), which will render RefactorErl unreliable for builds where dirty BIF's have been introduced, removed or renamed by Ericsson.

The significance of this problem is demonstrated by snippet 1, for *RefactorErl* actually allowed the elimination of the `Y` variable as a result of the `random:uniform/0` function falsely not being marked as being a dirty BIF by

the *function_analyzer*, and thus changing the code's semantics.

It should be noted that at the time of writing the side-effect analysis was not fully implemented and tested, and hence bugs were to be expected at this point. Nonetheless this example proves that marking a non-built-in function as being dirty depends on several human factors (e.g. correct marking of dirty BIFs by the RefactorErl team), which is more error-prone than Haskell's monad-approach for example.

```
1 f() ->
2     %generate a random number (side-effect)
3     Y = random:uniform(),
4     X = Y,
5     {X,Y}.
```

Snippet 1: Before eliminating variable Y

```
1 f() ->
2     X = random:uniform(),
3     {X,random:uniform()}.
```

Snippet 2: After eliminating variable Y

Concurrent programming Another point of interest is *concurrent programming*[52] which Erlang handles in an uncommon manner: rather than invoking threads the programmer spawns concurrent processes whom Erlang communicates with through message-passing, much in the way a mail server and client communicate.

This has some advantages over threads, for example because there is no shared memory, and because it is easier to have concurrent processes running simultaneously on different processors than threaded programs.

RefactorErl could incorporate refactorings that turn sequential processes into parallel processes and vice-versa. It could for example automatically split up *maps* of complex functions into smaller parts (smaller lists) which can then be computed in their own separate and parallel processes.

Dynamic typing Yet another interesting aspect of Erlang in this context is its dynamic typing system, which makes refactoring more troublesome, since it cannot be verified if the refactored code makes sense from a typing-perspective.

An example of this issue is the renaming of a function `f/1` as described in snippet 3.

```
1  f(X) when is_integer(X) ->
2      X*42;
3  f(X) when is_list(X) ->
4      X ++ "Hello World!";
5  f(_) ->
6      false;
7
8  % When applying the 'rename function' refactoring to
9  % 'f' (in function g), it cannot be detected which
10 % clause is referred to, because the type of Y is
11 % unknown. Therefore the entire 'f'-definition is
12 % renamed (i.e. all clauses), rather than a particular
13 % clause of 'f'. In a typed language it is possible to
14 % only rename the desired clause, whereas in Erlang
15 % this is impossible without requesting extra
16 % information from the user.
17
18 g(Y) ->
19     f(Y).
```

Snippet 3: Dynamic typing and refactoring problem example

For a typed system it would be clear to what type was referred since the type of `X` would be known at compile-time. Therefore it could, for example, rename only the `f/1` function clause which applies to *integers* on application of the *rename_function* transformation. In Erlang type information is not available before runtime though, and hence RefactorErl transforms the entire function (another option would be to prohibit the transformation in this situation). The same issue applies to the *inline_function* transformation (i.e. substitutes the selected application with the corresponding function body and executes compensations[32]), which can introduce a lot of code repetition.

And as a final example RefactorErl will never be able to apply certain coding conventions. A good example is Hungarian notation,[53] which requires prefixing variable-names with an abbreviation of the type they represent. This convention is relatively common and useful in untyped languages. Though, due to the lack of typing, RefactorErl cannot determine the desired name, and hence cannot apply this convention. This is in contradiction with refactoring tools for typed languages.

5.3.2 Wrangler

This open source refactoring instrument is being developed at the university of Kent. It aims to serve the same purpose as RefactorErl, but it has a different implementation: instead of using a database and an own parser, it works directly on the syntax tree that is parsed by Erlang's own parser.

A few of the inherent advantages of the 'Wrangler-approach' are that there exists only *one* syntax tree at every moment, it therefore needs no extra parser

and hence the possible number of points on which the system can fail is reduced. It is however less trivial for this approach to roll-back a refactoring, since that requires storage of the previous tree in some form, something that RefactorErl already does by design.[43] This means that Wrangler has to overcome this problem in another way than RefactorErl does – for example by keeping track of different files using a revision-control system (e.g. Subversion, CVS, custom-made)–, which inherently introduces new possibilities for the introduction of defects and might slow-down the system. This weakens the advantage the tool has over RefactorErl somewhat.

5.3.3 Tidier

The last tool I mentioned is Tidier, which distinguishes itself from the aforementioned tools by its (semi-)*automatic* approach. Rather than letting the user identify a part of code which is apt for refactoring, it does so automatically. One advantage is that it can identify refactorable code which the programmer failed to do, something that is bound to happen in large modules (needle – haystack). Snippet 4, which I copied from a paper[44] about Tidier, nicely illustrates this.

Snippet 4 is the original code, which comes from the reasonably big module *egd_render* written by the Erlang developers, and the code in snippet 5 after Tidier had its way with it. A seasoned Erlang programmer (which I think is safe to assume that Erlang programmers (mainly Ericsson employees) are) will immediately see this code could be easily simplified. Nonetheless this particular author and possible code reviewer(s) missed it.

```
1 if
2     Yp =:= Y -> true;
3     true -> false
4 end
```

Snippet 4: Automatic update example before Tidier

```
1 Yp =:= Y
```

Snippet 5: Automatic update example after Tidier

Hence, Tidier can be of great help in identifying refactoring possibilities (despite its experimental state, it is already being used extensively in the development of Erlang according to *Git*'s commit comments[54]), but it will never be able to fully replace the manual identification of refactorable code, which makes it a nice extension to the mentioned *manual* refactoring tools, rather than a replacement.

5.4 Refactoring other languages

As a comparison of the status of refactoring tools for Erlang, to that of other programming languages, this section mentions a few of the latter along with their most interesting properties. Note that it is infeasible to write about all

available refactoring tools here, so I just mention two popular refactoring tools here as an illustration (one Object Oriented and the other Functional).

5.4.1 ReSharper

ReSharper is a 3rd party commercial tool containing (among others) a refactorer which incorporates both the manual- (RefactorErl/Wrangler) and semi-automatic (Tidier) approach, for the object oriented C# language and integrates to the Visual Studio IDE. It is well out of the experimental phase and handles over 30 refactorings.[45]

5.4.2 HaRe

Haskell **Re**factorer[46] is an open source manual refactoring tool developed by the university of Kent, which covers the lazy, strongly-typed functional programming language Haskell 98, and is integrated with two development environments: Vim and (X)Emacs.

The tool is developed by Simon Thompson (Investigator) and Huiqing Li (Research Assistant) among others; also the main developers of Wrangler (supported by John Derrick and others of the university of Sheffield).

By the third release of HaRe, it supports 24 refactorings, and also exposes an API for defining refactorings or more general program transformations.[55]

5.5 Detailed problem description

An important property of a refactoring tool is that it should be very reliable, for it is unacceptable that it will introduce defects to totally correct code, or to any code for that matter. The problem here is that RefactorErl, at the moment of writing, consists of a little under 33 KLOC[2]. This implies that according to *LOC-vs-defects* estimates,[56] there exist between 2 to 70 errors per KLOC, resulting in 66 errors in the best scenario and 2310 in the worst. This is a problem, because it contradicts the *reliability* requirement of the tool.

Of course, these *LOC-vs-defects* approximations are to be considered merely an indication, since the complexer the project is, the higher the probability of defect-introduction is. This property is not incorporated very well in the LOC-vs-defects approximations though. The code of a compiler for example is many times more complex, and hence many times more likely to contain defects, than a desktop application which shows a straightforward slide-show of the photos of your latest family get-together.

Since RefactorErl involves a lot of parsing and rewriting of syntax-trees –which leaves no room for mistakes, because that would alter a program's semantics– and given that RefactorErl must be able to handle *every* syntactically correct Erlang program, this system can be classified as rather complex.

[2] determined using (on version 4009 of the tool's root directory): grep -v '^[[:space:]]*\($\|%\)' $(find . -name "*rl") | wc -l

Therefore it probably contains a number of defects closer to (or even over) 2310 than to 66.

5.6 Plan of attack

The goal of my research is to propose an approach in diminishing the aforementioned problem. This approach largely comes down to *model based testing*, because there already exist a lot of (regular) unit tests for the project, and I therefore considered a more thorough approach to be desired. Alternately I could have decided to prove code correctness, which I decided not to do because the definition of any interesting proofs would not fit in the time-window of this research. I split-up my approach into multiple tightly connected steps, which I will describe in this section. I defer the explanation of my motivations for the use of particular tools to aid me in achieving my goal to section 7 though.

5.6.1 Formalization of transformation-specifications

Motivation for formalization Since MBT relies wholly on the quality of the specification of the SUT, it is of utter importance to have it defined as clear and concise as possible – without any room for misinterpretation. Given that on embarking on this research the specifications were only defined in natural language, which leaves lots of room for misinterpretation, formalizing them was a logical first step. Additionally, a decent formal specification has the advantage of having multiple other practical appliances, e.g.: automatic derivation of models / test-generation and theorem-proving.

Z-notation I consider convenient notation for a formalization, to be first of all a notation that is easily understood by a great group of people. I found Z-notation to be very apt for this goal, for it basically comes down to Zermelo-Fraenkel set-theory, which is taught at universities world-wide.

Furthermore, Z-notation was designed for the purpose of defining specifications of (software) systems, and has been applied successfully in a variety of (critical) systems in such fields as cryptography,[57] control mechanisms and telecoms industry.[58]

A welcome additional benefit is that a lot of tools, mostly for checking Z specifications in several ways, are freely available. These include type-checkers, animators (which 'execute' the specifications against some test-data), and abstract test-generators.

Its 'familiar' syntax and semantics, successful applications on several large projects, and the rich set of tools available, convinced me that Z is an excellent candidate for the specification of RefactorErl's transformations.

Other means Another tactic of formalizing code would be to design a language that is tailored to fit RefactorErl, much like the developers of HaRe

did when they defined the λ_{Letrec} calculus to formalize some of their refactorings[46, 55] and statically prove them to be correct in a very generic fashion (i.e. focus on refactorings in general, rather than their implementations in HaRe).

As an illustration the formalization below was taken from the original paper.[55] It shows the specification of the *generalization of a definition* transformation in the λ_{Letrec} calculus (definitions 5 and 6). The actual specification is preceded by a few definitions that are used in the specification. These definitions use = for semantic, and ≡ for syntactic equivalence.

Definition 1 *Given two expressions E and E', E' is a sub-expression of E (notation $E' \subseteq E$), if $E' \in sub(E)$, where sub(E), the collection of sub-expressions of E, is defined inductively as follows:*

$sub(x) = \{x\}$
$sub(\lambda\, x.E) = \{\lambda\, x.E\} \cup sub(E)$
$sub(E_1\, E_2) = \{E_1\, E_2\} \cup sub(E_1) \cup sub(E_2)$
$sub(letrec\, x_1 = E_1, ..., x_n = E_n\, in\, E) =$
$\{letrec\, x_1 = E_1, ..., x_n = E_n\, in\, E\} \cup sub(E) \cup sub(E_1) \cup ... \cup sub(E_n)$

Definition 2 *$FV(E)$ is the set of free variables in expression E, and is defined as:*

$FV(x) = \{x\}$
$FV(\lambda\, x.E) = FV(E) - \{x\}$
$FV(E_1 E_2) = FV(E_1) \cup FV(E_2)$
$FV(letrec\, x_1 = E_1, ..., x_n = E_n\, in\, E) =$
$(FV(E_1) \cup ... \cup FV(E_n) \cup FV(E)) - \{x_1, ..., x_n\}$

Definition 3 *A Context $C[\,]$ is an expression with one hole in it; e.g. $C[E]$ means that C is an expression in which sub-expression E occurs exactly once.*

Definition 4 *Given an expression E and a context $C[\,]$, we define $sub(E, C)$ as those sub-expressions of $C[E]$ which contain the hole filled with the expression E, that is: $e \in sub(E, C)$ iff $\exists\, C_1[\,], C_2[\,]$, such that $e \equiv C_2[E] \wedge C[\,] \equiv C_1[C_2[\,]]$. Another way of explaining it would be: all sub-expressions of C which contain the expression E.*

Definition 5 *Given an expression $letrec\, x_1 = E_1, ..., x_i = E_i, ..., x_n = E_n in E_0$ Assume E' is a sub-expression of E_i, and $E_i \equiv C[E']$. Then the condition for generalizing the definition $x_i = E_i$ on E' is:*

$x_i \notin FV(E') \wedge \forall\, x, e : (x \in FV(E') \wedge e \in sub(E_i, C) \Rightarrow x \in FV(e))$

Definition 6 *After generalization, the original expression becomes:*

$letrec\, x_1 = E_1[x_i := x_i E'], ..., x_i = \lambda\, z.(C[z][x_i := x_i z]), ...,$
$x_n = E_n[x_i := x_i E']\, in\, E_0[x_i := x_i E']$, *where z is a fresh variable.*

Definition 5 can be regarded as a property that should hold before the transformation: all free variables in the context of E', the part for which E_i is generalized, should be free in all its sub-expressions as well. Otherwise the transformation would change semantics of these sub-expressions. Also, E' cannot contain x_i as a free variable, because x_i changes its semantics in this refactoring (it becomes a lambda-expression), so the x_i in E' would have a different meaning after the transformation.

And definition 6 shows what actually happens when the preconditions are satisfied: x_i becomes a lambda expression, with one free variable z, that returns $C[z]$. All calls to x_i are then replaced by $x_i E'$, because the meaning of x_i has changed from $C[E']$ to $\lambda\, z.C[z]$. Therefore $x_i E'$ will now result in the former $C[E']$, and hence semantics are preserved.

I could have applied this approach as well, but I found Z to have many advantages over defining my own notation or using λ_{letrec}. Mainly I find Z to be more readable; there is a huge collection of tools available for Z; many people are 'familiar' with set-builder notation and hence Z; Z has been applied in several projects successfully.

Specification correctness In an effort to gain confidence in the correctness of these specifications I defined using Z-notation I applied type-checking, and animation using *FUZZ* and *JAZA*[3] respectively, tools on which I will elaborate in section 7.3.

Generating test cases Additionally, as an illustration of the usefulness of Z-specifications I tried and failed, to derive abstract MBT-cases from the specifications using *Fastest*.[59] The problem was that the tool started working, but seemed to stop responding. I could not find any documentation about the problem, and decided to abandon the plan of generating test cases.

The authors of Fastest claim that the program can generate abstract test-cases from Z specifications however, and that it can even compile them to C-code. Also, compilers for other languages can be written. So, if the tool would work, it demonstrates a very powerful advantage over natural language of defining specifications in Z-notation, because QC tests could be automatically generated from it for example.

5.6.2 Model based testing using QuickCheck

After formalizing the refactoring specifications I used them to define some actual Model based tests[60, 61] in Erlang using QuickCheck, which is a MBT framework for Erlang (among others).

[3] I used version 1.1, the stable version at the time, which contained a minor bug which caused it not to compile against `ghc 6.10.3`. I fixed it, and sent Mark Utting (the author) a patch which he will apply and publish as soon as possible.

Model based testing This is a more thorough approach than unit-testing a system, because not only its boundary-values, but its entire model is checked.

This model mimics the desired behavior of the system-under-test (SUT). On a given input it calculates the expected output of the system-under-test when fed the a particular input. It can either manually or automatically derived from a system's specification. Since, to my knowledge, there do not exist tools at the moment that can do this automatically for Erlang, I have derived the models manually.

QuickCheck The defined models are embedded in several properties (each property contains its own model), which test the actual implementation against the appropriate model. These properties are then tested by and against inputs that are generated by QuickCheck.[62] To illustrate this, one property of the standard list reversal function is defined in snippet 6:

```
1 prop_reverse() ->
2   ?FORALL({Xs,Ys},
3     {list(int()),list(int())},
4     lists:reverse(Xs++Ys) == lists:reverse(Ys) ++
          lists:reverse(Xs)).
```

Snippet 6: Reversal function in QC

Which is the QuickCheck's equivalent of this set-builder notation (lists:reverse using sets rather than lists):

$\forall\, Xs \subset \mathbb{Z}, Ys \subset \mathbb{Z}.$

$lists : reverse(Xs \cup Ys) = lists : reverse(Ys) \cup lists : reverse(Xs)$

As you might notice, a dedicated model is absent here. Instead the implementation itself is used as a model to test itself for certain behavior. A test with an actual –example– model is displayed in snippet 7. In practice both approaches are applied intertwined in MBT-ing RefactorErl's transformations.

```
1 reverse_model([]) -> [];
2 reverse_model([X]) -> [X];
3 reverse_model([X:XS]) ->
4       reverse_model(XS) ++ [X].
5
6 prop_reverse_em() ->
7   ?FORALL({Xs,Ys},
8     {list(int()),list(int())},
9     reverse_model(Xs++Ys) == lists:reverse(Ys) ++
          lists:reverse(Xs)).
```

Snippet 7: Reversal function and example model in QC

Generators and shrinking QC uses generators to generate input for the test. In the example of snippet 7, `list(int())` tells QC to generate a random list of integers.

It is also possible to define your own generators, of which the *correct function name*-generator created by István Bozó – one of RefactorErl's team members – is a practical example (snippet 8).

```
1 %% Function generates 'NumNames' function names
2 %% length between 1 and 7 chars [a..zA..Z_].
3 %% Could be extended with support for numbers.
4 generate_funnames(NumNames) ->
5     Abc = lists:seq($a, $z) ++ lists:seq($A, $Z) ++ ?
          Underscore,
6     [io_lib:write(list_to_atom(refqc_common:get_n(
          random:uniform(7), Abc)))
7     || _ <- lists:seq(0,NumNames)].
```

Snippet 8: QC Function name generator

When a defect is encountered, QuickCheck can be instructed to find the smallest possible value on which the defect appears, which is called *shrinking*.

Validity dependant of implementation The validity of the properties I defined depends a lot on RefactorErl's implementation, since I heavily use parts of it in calculating the output of a model on a particular input. Hence, it plays a key role in the defined models.

The bigger part of the functions of the implementation on which the properties rely, is defined in the Querying-layer (figure 7.2). This layer is used in the models to look-up subsets of corresponding nodes in the syntax-tree, on a given input (as described in snippet 10 of section 6.2.1). The result is then used to calculate the desired output of the model.

My properties assume this model to be correct, and compare the result of the aspect they are testing to it, which results in the verdict of the test. This verdict might either reveal an issue in the implementation *or* the model, which are both desired outputs. It is up to the tester to trace which of both causes the issue.

Of course, the assumption of this model which relies so much on the implementation to be correct – even though the implementation is exactly the subject of the MBT – will raise some questions (e.g. how does this affect the validity of the test results?). I will elaborate on this approach in section 7.4.

5.6.3 Selecting hard-to-refactor Erlang constructs

Model based testing requires input, which I would like to generate semi-randomly: I would like to test at least some data that I classify as 'interesting', and complement it with some randomly generated input.

This generation of interesting 'random' data is possible,[63] but not too straight-forward in RefactorErl's case, because actual Erlang code is required as input to the tests. Horpácsi Dániel –member of the research-group– is currently working on such a generator.

Since there exists no good solution for generating semi-random Erlang modules at the moment, the next best thing to do is to select some complex constructs in an educated fashion (e.g.: nested expressions; 'uncommon' function patterns; 'special' constructs of Erlang that are hardly used in a certain context) in an effort to reveal defects using the defined tests. A set of modules can be created from these complex constructs as well, which can eventually help building a strong *regression test-set*, and to identify defects in both RefacterErl's and the defined tests' code.

6 Background information on tools used

6.1 Introduction to research basics

This part of the thesis gives a little background information on some of the most important aspects of the thesis, e.g.: how to work with Z (and related tools) and RefactorErl. To get a good understanding of the rest of the thesis, it is important that the reader is familiar with the knowledge presented in this section.

6.2 Instruments used in research

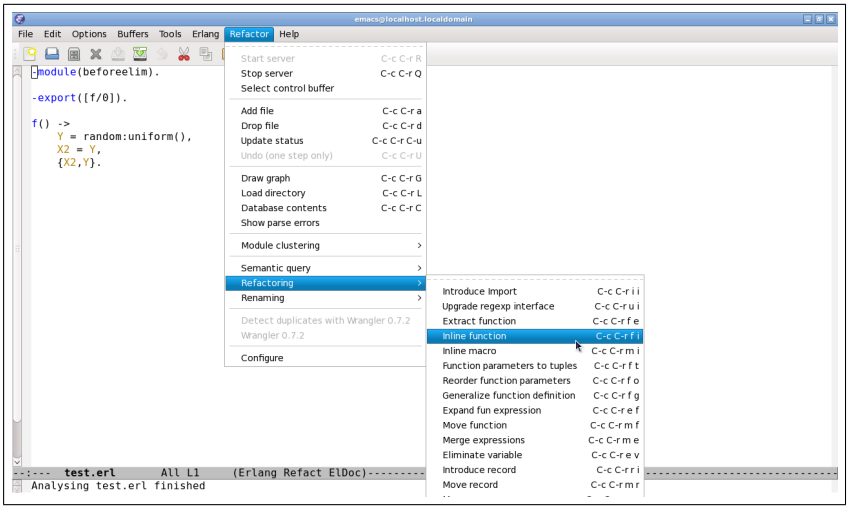
6.2.1 RefactorErl

Introduction The goal of this project is to develop a refactoring tool for the Erlang functional programming language. The group is investigating implementation possibilities of refactoring using its experiences from previous research, and building a refactoring framework that can be integrated with any development environment.[28]

There is a manual on the RefactorErl website, explaining how to install the environment on Emacs. This then displayed as a new menu (figure 6.2.1) in the editor. When a transformation is selected, and the cursor is on a corresponding node in the source code, RefactorErl asks for input required for the transformation (e.g. new variable name or argument order).

RefactorErl is written in Erlang, and has its own Erlang parser. The parsed source code is stored as a syntax/semantics graph in an MNesia database (this database comes with Erlang). It enables easy lookup of nodes, and makes it possible to support the ability to roll-back changes. The general structure of the system is displayed in figure 7.2.

An example of a visual graph representation generated by RefactorErl (using the *referl_draw_graph* module), is shown in figure 6, which represents the code in snippet 9; the identity function. The hexagons represent the semantic nodes of the code, whereas the rounded squares are syntactical. The former are nodes



that contain the actual meaning of a variable or function for example (e.g. its name, and arity for example). The syntactical nodes represent the occurrences of a variable or function, and contain information like token position and whether it is a reference or a definition.

Even though the example in snippet 9 only consists of a few lines of code, the graph is rather big, but –even when considering a serious snippet of code– the graph visualization can be very useful nevertheless in manually verifying the correctness of relations between nodes.

```
1  -module(example).
2
3  id(X) ->
4      X.
```

Snippet 9: Identity function in an Erlang module

Querying the graph To provide easy access to the nodes of the parsed source code in the database RefactorErl contains its own querying methods. If one would like to retrieve a list of all variables in the function *id* of snippet 9 for example, it can be seen in figure 6 that first the *example.erl* file needs to be queried, followed by its 2nd *form*, after which the *function clause* should be requested.

This query will return a *clause* node, from which a list of all its child *variable* nodes can be queried. The code would look like something along the lines of snippet 10.

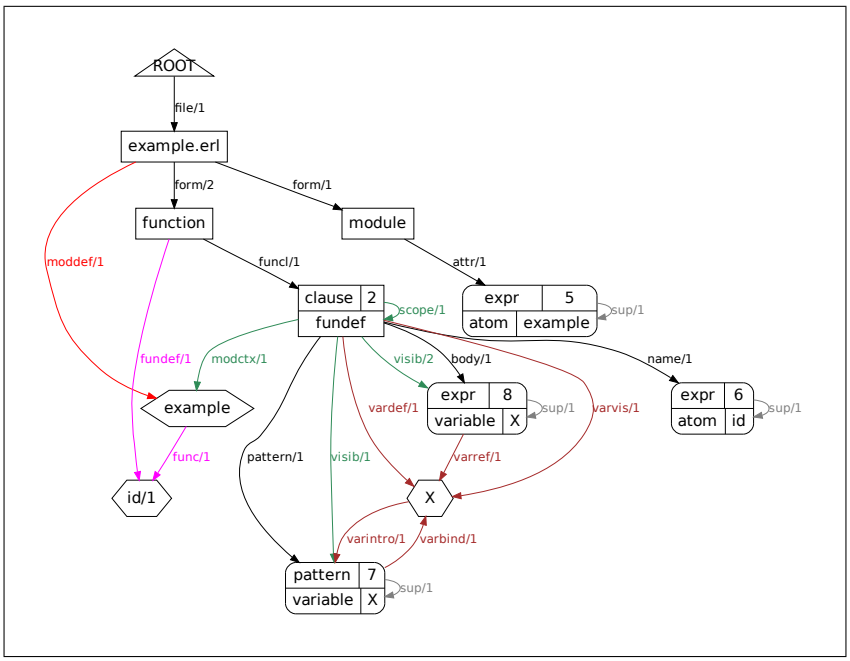


Figure 6: Syntactic / Semantic graph of Example

```
1 FileNode = ?Query:exec(?File:find("example.erl")),
2 FunNode = ?Query:exec(FileNode,?Query:seq([
3     ?File:form(2),?Form:clause(1)])),
4 ?Query:exec(FunNode,?Clause:variables()).
```

Snippet 10: Example of a basic RefactorErl query

There are also querying actions available that can update nodes in the graph in a similar fashion, which is how the system performs its refactorings. After a refactoring the tree is read from the database, and written to the particular file, keeping the actual file and Mnesia tree in sync. This subject was part of the introductory research I did familiarize myself with Erlang and RefactorErl; it is documented in section A.

6.2.2 QuickCheck

Quviq QuickCheck is a commercial Erlang-based library for the model based testing of Erlang source code. It is inspired on a tool initially developed for testing Haskell source code.[61]

In a nutshell this system is able to take a model, and feed it data, based on information provided by the tester. When an error is found, the system will automatically shrinks (that is: it finds the smallest possible input for which an error occurs) the solution that results in an error.

To use the system, a tester has to write an Erlang module, which will call the `quickcheck` function (provided with a test-property) of the `eqc` library. Depending on the model that the tester made, he will have to provide a *generator*, a function that generates data that is accepted by the model.

A concise example of a QuickCheck property, and how to test it is displayed in snippet 11.[64]

```
1 % definition of a property that tests if a member of
2 % a list is correctly deleted.
3 prop_delete() ->
4     ?FORALL({I,L},
5         {int(),list(int())},
6         not lists:member(I,
7         lists:delete(I,L))).
8
9 eqc:quickcheck(examples:prop_delete()).
```

Snippet 11: QC property test

More information on using QuickCheck can be found in several examples and a manual that are available online. I found that extensive documentation to the system is not *very* easily available though, but armed with the manual and some time for trial and error, it is possible to understand the system pretty well.

6.2.3 Z-notation

In this thesis Z notation is used to specify transformations of RefactorErl. It is to be used with LaTeX, and its syntax is quite comparable to that of the `math` environment. Using this notation, which is based on Zermelo-Fraenkel set theory, it is possible to define a system using sets and predicates.

Systems are specified using *schemas*, which represent states of the system with signatures of predicates and functions, along with their definitions (all defined in an adaptation of set-notation).

Several examples are included in this thesis, along with their explanation in natural language, and I will therefore not elaborate further on it here. Better, and more thorough examples are available though. A very good example is the *Birthday Book specification* as included in the Z-Book.[65]

6.2.4 Fuzz

When compiling the specifications with LaTeX though, they are only checked if they comply with the LaTeX syntax. Because the goal is to write very precise specifications, more thorough verification is needed. It is important that the types of the system are correct for example, or it will not make any sense.

This can be checked using *fuzz*, a command-line tool that was designed for this purpose. After installation of fuzz, a specification can be tested on type-correctness by entering the following command in the CLI:

`zfuzz <filename.tex>`.

When an error is found, its location in the file and a description of the problem is shown in CLI. When no problems are encountered, nothing shows up.

6.2.5 JAZA

JAZA is an animator for Z specifications (i.e. this program makes it possible to 'execute' specifications). The goal of this system is to enable the user to test the specification for (un)desired behavior.

7 Approach

7.1 Introduction

This section starts off with a detailed description of the scope of my research, followed by the methods I applied and my motivations for using them.

7.2 What is the scope of testing?

Model based testing My main approach in testing the implementation of several refactorings in RefactorErl, will be to model based test them using QuickCheck (section 7.4).

Models, which are very important to this type of testing, need to be defined. Because of this significance, I will discuss my approach of defining my models and issues inherent to this approach in the upcoming paragraphs.

Flawed models My goal is to write tests that are as complete and aim to reveal defects throughout the entire system. Unfortunately this goal is somewhat complicated to achieve, because I use quite some functions of the implementation of RefactorErl to calculate outputs for my models, as I already mentioned in the introduction section.

The obvious problem this introduces, is that my models cannot be considered correct, after all I suspect that the implementation contains errors and hence it's output and therefore the model's output is not *fully* reliable. One result of this, is that false negatives will go unnoticed.

Creating models which do not use the implementation at all can theoretically solve this problem, but that will require lots of lines of code, which most probably also contain defects, and hence are not *fully* reliable as well.

Results' worth Given this information, what is the value of the test results then? Given the unreliability of models, the results cannot be relied on as well. Well, the results will be either: positives, negatives, false negatives or false positives. Finding (false) negatives does not mean anything; this proves neither the absence nor presence of defects, because the models cannot be fully relied on

and the negative will have to be fully analyzed to determine whether it is a real or false negative. The (false) positives are the true value of the results, these ensure that there either is a defect in the defined property, in the model or the implementation. A tester will have to analyze the results in order to determine the exact whereabouts of the defect.

Also notice that there will be no completely false positives, for when an issue is detected in a test, this is always the result of a flaw in the implementation or the model. And when the model is fully correct, no false positives can occur at all, because each positive is the result of the implementation conflicting with its model and hence its specification.

To wrap it up: the properties, as thorough as they may be, might not return all the flaws in the implementation, but neither will they return completely false positives. Hence, each issue found by the property is a real issue, which is what makes these tests valuable.

Increasing results value To improve upon these tests, it is of foremost importance to notice that the implementation of RefactorErl consists of several layers which fully rely on each other(figure 7.2). Layers are groups of libraries

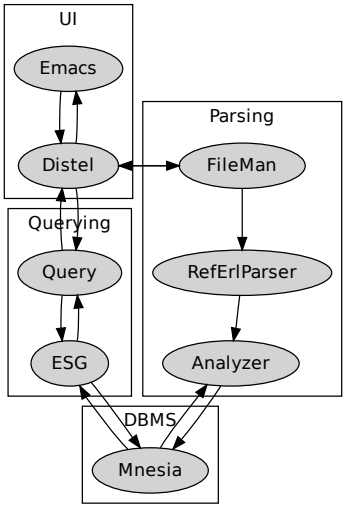


Figure 7: RefactorErl architecture

which together have a dedicated purpose, like querying the syntax graph, or parsing the source code of a file for example. These layers depend on each other for information gathering purposes (e.g. the Querying layer requires information

it receives from the UI and DBMS layers, as can be seen in figure 7.2. Defects can occur in each of them, and since I refrain in my tests to using only the upper layer (querying-layer), defects on layers it is dependent of can be missed.

To illustrate this, consider the situation where one of the lower layers (a layer that another layer is dependent of) consistently provides wrong information (e.g. too many, too few, or the wrong nodes of a parsing-tree) to the querying-layer. This results in both the model and the implementation outputting the same incorrect information. From the perspective of the property, the implementation conforms to the model, and hence it passes the tests: a false negative.

A concrete example of this situation is the model-testing of the *function_rename* transformation. In this particular case, which I will describe in more detail in section 7.4.1, it is crucial that before and after the transformation all nodes that refer to the particular function's semantic node are correctly updated.

Now consider that as a result of a defect in the querying-layer, it misses some of the function's references and hence returns just a subset of the references that exist in the module that is being tested. Notice that this is the information that is output by the model to represent the desired output of the implementation. And after the transformation the querying-layer returns the exact same, but correctly updated, subset of function references. In this case the test will pass, since after the transformation, the implementation returns the desired output as represented by the model. This is a false negative.

This situation actually occurred to me (one time that I know of), and is described in section 8.6. The other way around it works the same: one cannot expect the upper layers to be correct because tests of the layers it depends on all pass.

The only way to locate and eliminate these defects (with the approach currently discussed) is to independently test each of these layers as well.

Why not dedicated models? Clearly, the approach described has some issues, and hence it is useful to review the alternative: writing for each property a fully independent, dedicated model, which does not use any of the implementation's information to mimic the desired behavior of the system. For example, the model could compute desired output based on a certain input by parsing the Erlang syntax tree.

This approach comes at the cost of writing a lot of extra lines of code for this purpose alone, which gives rise to new defects that will probably result in false negatives as well. Hence, I consider this approach to be more useful in the scenario where development of the implementation and MBT go hand-in-hand from the start.

The method mentioned earlier, the one I actually applied, is more interesting when a substantial implementation is already at hand, for: it allows the tester to quickly build MBT's focused on a high level of the system which are likely to reveal a lot of errors at all layers; it requires fewer lines of code, since a lot of required model-information can be gathered using the implementation; the

tester can choose whether or not he trusts the models enough, and decide if he wants to write MBT's for lower layers to strengthen this belief. The labor of having to manually write extensive models does not seem to weigh op to these reasons for using the first approach.

What layer to use in models? So, the question to be answered here is: 'which layer should be used for in the models used in the tests?'

When for example, tests are written for verification of functions defined in the bottom layer, defects will be revealed on that level, whereas defects in higher levels will go unnoticed. By testing functions in the higher layers on the other hand, defects in all lower layers might be revealed as well, including those found in the tests for the bottom layer functions. This 'high-level testing' renders the latter redundant.

On the other hand, I explained how using information provided by the top-layer in models could result in false negatives, because defects in the lower layers might result in the top-layer returning inadequate information.

I decided, based on these considerations, to only use the top-most layer in my models, the foremost reason being the fact that it saves me the trouble of writing a lot of code that has already been written by the project group. As an effect of this decision I will have to write significantly fewer lines of code for my tests, which will save time, but more importantly, will result in a drop in the number of defects introduced in the tests in conformance with *LOC-vs-defects* estimates.

The only way to identify and eliminate as much as possible false negatives that result from this approach, is to separately MBT each layer (from the top-most to the bottom layer), thus to divide-and-conquer rather than using a big-bang approach. This engenders several advantages, e.g.: it will be easier to divide the testing of code in portions that are actually feasible for one person to handle in a reasonable time; and make it possible to have multiple people working on related parts of the code that are spread over different layers, at the same time.

Actually this approach has already been incorporated slightly, since Máté Tejfel –a member of the research group– has developed a QC testing module for verification of some properties that should hold for the *variable analyzer* (figure 7.2), which resides in the parsing-layer.

7.3 Step 1: Formal specification of properties in Z

Since tests can only be as good as the specification they test, my first step toward MBT-ing RefactorErl, consist of formalizing the specifications of transformations, using *Z-notation*. Before this moment (for RefactorErl) these were only defined in terms of 'properties': statements that should always hold for a certain transformation, in natural language. Similar specifications have been defined on a more abstract level[55] (the paper addresses the actual transformations, whereas my focus is more RefactorErl / code specific) for *HaRe*.[46] I considered specifying all the functions that I use in my Z specifications too

much work to fit in my thesis though, and decided to define function signatures in Z that are to represent the equivalent existing functions in RefactorErl. These functions can in the future be specified in Z as well, making the total specification more complete.

Formalizing the specifications might seem a bit superfluous and even redundant, since I might as well directly transform the currently defined natural-language properties to *QuickCheck* code (it is a formalization of properties after all), but there are some reasons for doing this nonetheless that I will elaborate on.

The main reason for formalizing the specifications in the first place is the problem that resides in definitions in natural language: they are context dependent. Hence, people might interpret them in different ways; not a very comforting thought when writing *reliable* code is one of the most important goals of a project. By formalizing the specifications using Z, the context dependence is resolved, and hence all readers should interpret them the same. Since it is derived from a natural language specification, designers of the Z specification will have to discuss what the final meaning of the Z specification should be though.

But why not just formalize the specifications using QC then? I am planning to test the specifications using QC anyway, and hence I could eliminate a formalization step. First of all because QC is no common notation, hence people that are not familiar with QC can have a hard time understanding its semantics. Z notation on the contrary, is understood by a much larger group of people (I referring to the final, compiled specification), because it is basically set-notation which many computer scientists are familiar with.

Secondly, QC is designed for the purpose of MBT: it has no other application at all, whereas Z-notation can be used to for multiple purposes like generating models out of specifications. Furthermore, unlike Z-notation, QC is language dependent; there are tools available for language independent type-checking and animation of specifications defined using Z-notation; Use of formalized specifications using Z is not limited to MBT, but can be used for all testing purposes in the future. Thus, an independent notation such as Z that was designed for the purpose of formalizing specifications of systems, has several advantages over a very specific 'language' like QC, which was designed for the sole purpose of MBT.

Additionally, once the formal specifications are written down, they can be used for many purposes in future work, like formally proving the correctness of a model using a (automated) theorem prover.[66] Natural language specifications do not facilitate this possibility.

And the third reason is that formal specifications present you the natural language specifications from a different perspective (e.g. set-wise, rather than a series of restrictions; forces you to truly understand them, as opposed to natural language definitions which will make you easily miss/read over things). So, formalizing might aid one in seeing that tests are incomplete because of the different perspective.

7.3.1 Z-notation

The formal specification notation Z (pronounced "`zed`"), useful for describing computer-based systems, is based on *Zermelo-Fraenkel* set theory and first order predicate logic. It has been developed by the Programming Research Group (PRG) at the Oxford University Computing Laboratory (OUCL) and elsewhere since the late 1970s, [...] Z is now defined by an ISO standard and is public domain.[67]

This notation seems to be the most convenient tool for the job, since I was looking for a way to write specifications that are as tool-independent as possible. But at the same time I wanted to have a tool available which can verify the specification's syntax, so as to avoid writing specifications that have are not formally compliant with the used notation.

The Z-notation conforms perfectly to these requirements, since any text-processor will do as a tool for writing specifications in Z; it has a syntax verification tool available; and last but not least the Z-notation is based on an internationally well-known notation (Zermelo-Fraenkel set theory) as stated in the first paragraph of this section.

7.3.2 Tools for Z

Fuzz is a collection of tools that help you to format and print Z specifications and check them for compliance with the Z scope and type rules.[68]

It is a command-line tool which I used to verify the syntax of all my specifications. Basically it searches through a LaTeX source for Z specifications and verifies their syntactic correctness. If it detects incorrect syntax or typing issues, it outputs on what line and character position the defect is to be found.

Jaza is an 'animator' for the Z formal specification language. It is intended to help one validate specifications via: evaluating Z expressions, testing Z schema's against example data values and executing Z specifications (but not all specifications are executable).[69]

Jaza can 'execute' a specification, given some input. This will result in an output, and the designer of the specification use this result to verify if it is the desired output.

7.4 Step 2: MBT transformations using QuickCheck

Currently RefactorErl testing is approached by means of unit- and regression testing. Good unit tests usually perform boundary checks, and try to achieve decision- and code coverage. But even if all code and decisions were covered, that does not imply that the code will work for all inputs, since not all input is checked.

Model based testing on the other hand will more thoroughly test the code, by feeding functions randomly generated data, or when available, data based on heuristics; which is less naive than the random approach. Theoretically, this

approach will test every case (i.e.: if it ran to infinity, and the model is complete) as opposed to the unit tests which will only test a small subset of inputs.

Besides that, the model based testing approach separates the properties being tested and the test data, whereas in unit tests these are defined together in one test. Which makes it easier to change test data and properties apart from each other. Also, model based testing can be configured to test more thoroughly by changing one parameter, which is not the case with unit testing.

Given these advantages, it is interesting to incorporate model based testing as an extension to the current methods of testing RefactorErl.

This actual testing part of my thesis is also the part that is affected the most with my models using information supplied by the implementation of RefactorErl (section 7.2). Even though the models only access information through the querying-layer (the upper-most layer) the tests are able to detect defects throughout all the layers of the system, as opposed to the situation where I would use one of the other layers 'below' the querying-layer. For example, most issues that are detected in the top layer actually originate from defects residing in the underlying layers, given that this is a much larger code base and it is where the more complex computations are performed. When an issue is found, it is up to the tester to find out in which layer the defect causing this issue is located.

And finally, by applying this method, the false negatives in the upper layer tests are naturally eliminated when bugs are fixed in lower layers. So, as a result of the project's maturing process, false negatives will become positives.

7.4.1 Refactoring 1: Rename function

The tests I defined for the *rename_function* transformation are an extension to an incomplete QuickCheck module that was written by Elroy Jumpertz.[3] The extension consists of two properties, which verify correctness of the binding structure of both function parameters and functions.

These properties are defined in a paper[48] written by Huiqing Li and Simon Thompson of the Wrangler project. They state that the binding structure of a function: "refers to the association of uses of identifiers with their definitions in a program, and is determined by the scope of the identifiers.". This basically comes down to associating the scopes of all functions throughout a set of modules with their definitions.

To verify the correctness of the binding structure after the *rename_function* transformation –which renames a function and all its occurrences loaded in the database–, I collect all nodes (i.e. definition and references) that belong to a certain function, and determine their token position in the syntax tree (i.e. in the database, not in the source file, because these might change when a function name changes length). Before and after the renaming of a function, these token positions should remain unaltered –this also means that the binding structure is still intact–, and the function name should have changed.

Of course from the sole perspective of correctness, this property does not fully hold. Consider snippets 12 and 13. If the function renaming would refactor

the code like this, the semantics of the module would remain intact. Hence, the refactoring is correct with respect to the specification, but contradicts the property I defined, because the token positions in the syntax tree have changed.

However, another important property that holds for all transformations, is that the overall structure of the code should remain recognizable after each transformation (i.e. change as little as possible). This means that when moving a function across modules for example, the unaffected code should remain in place, and the moved function being added to the beginning or the ending of the target module. And in this case of merely renaming a function, its token positions (definition and references) in the syntax tree should remain unaltered.

```
1 f() ->
2     42.
3
4 g() ->
5     f().
```

Snippet 12: Before renaming f/0 to c/0

```
1 g() ->
2     c().
3
4 c() ->
5     42.
```

Snippet 13: After renaming (semantically equivalent)

7.4.2 Refactoring 2: Reorder function parameters

After finishing the *rename_fun* test module, the most logical step was to define tests for the *reorder_fun_args* transformation, since functions for analyzing variable and function binding structures are already at hand as a result of the *rename_fun* test module and earlier work by István Bozó, which are exactly the functionalities needed to verify properties for the *reorder_fun_args* module.

In general, the test module for this transformation performs the same trick that was applied in the *rename_fun* test module.

The two properties I implemented are the a binding structure comparison of both function scope and the arguments of the function, and the 'reverse' property.

The latter verifies that if the arguments of a function are reversed, and the same transformation is performed on the result, the binding structure after the transformation is equal to the one before it. Like I stated before: *theoretically* verification of binding structures of randomly ordered arguments (i.e. the 'normal' binding structure test) will test this case too, but one can only be sure of that if the test was ran to infinity. However, by forcing the verification of this reverse-property, the reordering of the same variables consecutively is checked, which might reveal errors like snippet 14.

7.5 Step 3: Regression testing of QC tests

While writing the model-tests and searching for *complicated constructs* (section 7.6) to test them against and vice-versa, I decided to write a regression-test module which takes some code and tests it against one or several specified model-tests (rather than having to manually test them one by one). But apart from easing the testing-process there are some other reasons for writing such a module.

First of all, *complex constructs* (section 7.6) might cause trouble in multiple refactorings, and therefore should be tested against every applicable model after each change to RefactorErl's source. This requires several steps to be taken by the tester, that can easily be automated by the proposed regression testing module. Also, in the future, it might be useful for the project to include the QC regression tests in the already existing regression test, in order to quickly detect the (re)introduction of bugs.

The second reason is that each time perform a test was performed the test-set had to be reinitialized(i.e. keep a copy of the original files and copy them over the modified test files), as a result of the refactorings being performed directly on the original test code. The regression module can handle this fully automatically.

7.6 Step 4: Selecting interesting complex constructs

I searched for some 'complex' constructs that assist in revealing defects in RefactorErl, as well as in my own test-modules. This was achieved by studying the Erlang manual thoroughly looking for 'rare' constructs; by checking code from existing projects for bugs using the model-tests; and using common sense (e.g.: nested constructs probably cause trouble more easily than plain variable assignments, since programmers tend to mainly test only the more obvious cases when coding).

By this selecting these constructs a set of modules –that can be used as useful input to the MBT's– is created. These modules can be classified by purpose.

For example, testing the *rename_variable* refactoring against a module that does not contain any variables is (most probably) not too interesting. So, in principle, this is a less naive approach than just feeding model based tests fully-randomly generated data.

This is not to say that the generation of semi-random data as input for model based tests is useless. If one would write an Erlang code generator which accepts parameters that define what types of constructs the emitted code should contain (e.g. lots of list comprehensions, deeply nested functions, etc.), then this might result in interesting and more 'creative' code than an Erlang programmer would come up with.

The result of this part of the research is a set of interesting modules, which will form the basis for a useful QC regression test-set.

8 Results

This section shows the results of my research, grouped by the methods (as described in section 7 from which they resulted. The results of the MBT, and Complex Construct sections sketch concrete situations that revealed defects in RefactorErl. The Regression Test section is more of a description of the use of an Erlang module I wrote. And the Z-notation part is the simplest specification I formalized, with an explanation of its meaning.

8.1 Model based testing

8.1.1 'Rename function'-refactoring

The *binding structure*-property of this transformation revealed one major bug, which was fixed by the group before I reported it though.

Consider a piece of code (snippet 14) containing two function definitions `f/0` and `g/1`. Rename `f/0` to `g/0` and then rename it back to its original name `f/0`.

```
1 % the bug shows after renaming 'f/0' to 'g/0', and
      then back to 'f/0' again
2 f() ->
3     g(7).
4
5 g(X) ->
6     6*X.
```

Snippet 14: Binding structure violation bug

This action broke the entire function binding structure, causing symptoms like: the *export* attribute not being updated accordingly, 'phantom' function nodes (i.e. when renaming `f/0` back to `g/0` one more time, the system would complain that this function already existed), and other function nodes not being rename-able anymore.

Even though this bug was solved before I reported it, its discovery by this test is an important result, since it demonstrates that the test's ability to reveal bugs it was designed to find.

Additionally, the test is able to detect the *rename_fun_record* bug (snippet 19), which emphasizes this conclusion.

8.2 Regression testing

8.2.1 Merely a convenience module

The result of the regression testing should be regarded merely in terms of a convenience-module addition to the QC modules that were written, and that are to be written in the future; and as an extension to the project's current regression tests, rather than another means of revealing bugs.

To illustrate this: writing this module resulted from the inconvenience of having to supply a new test-set after each test; the calling of inconsistent function names on the different testing modules to initiate testing; and not being able to execute a batch of different model test on a test-set. These are just a few examples of things that result in very tedious, repetitive and sometimes inconsistent work (e.g. no convention for function names for each testing module), which is eliminated by a single regression-test module.

Hence, this part's result is a module that helps to get more out of the potential of the model tests, the gathered complex constructs, and the testing of the project in general.

8.2.2 Concept of the module

The module is an extension to the MBT research, and its concept is quite simple. It is a module that can be used to batch-test a set of QC modules against a set of regression-test files.

The module is easily extensible, meaning that whenever a new QC testing module is written, it can be integrated in the regression tester without having to modify a single line of code of the regression module itself.

This is achieved by adding a `regtest/2` function to the particular QC module, which on invocation is supposed to call the properties that are defined in the module. How this function is implemented, is entirely up to the author of the module. As for extending the regression test-set (the modules which are being tested), this is as trivial as copying a file to a directory where all test-files reside.

I implemented this concept in the `regression` module that contains a function `go/2` which accepts two parameters, the first one being a path to a directory which contains a set of regression test files, and the second being a list of options.

When this function is called, the module (which is assumed to reside in the same directory as all QC testing modules) compiles all Erlang files residing in its own directory, which are presumably all QC testing modules.

This this testing of modules that do and do not compile is important (and filtering out those that do not compile), since it is very plausible that a tester or programmer has some work-in-progress code in this directory when he executes the test, which would interrupt the testing as a result of the erroneous modules not being filtered out.

Modules that compiled without throwing errors are then scanned for a function called `regtest/2`, which results in a list of modules which are supposed to be included in regression testing. For each of them a separate *result directory* is created.

In this newly created directory, for all QC modules that have a `regtest/2` function, a separate directory is created, where all modules from the specified regression test-set are copied. This ensures that the original regression test-set remains intact, and that each test will have its own result set in their dedicated result directories.

Finally, on each of the QC modules the `regtest/2` function is invoked, which should invoke the testing of all the properties that are specified in that particular module. To include his module in the regression testing, all the programmer has to do is to implement this `regtest/2` function.

8.3 Specification formalization

8.3.1 Introduction

Specifications of two transformations were formalized in my research using Z-notation: *variable_rename* and *function_reorder_arg*. Both are contained in this section and in this order.

8.3.2 Notation

Table 2 gives a short description of the notations that are not too straightforward, or Z-specific.

Notation	Description
$\nrightarrow$	Partial function
$\mathbb{P}_1\, S$	$\mathbb{P}\, S \backslash \emptyset$
`X?`	Input variable X
`Y!`	Output variable Y
dom `R`	Domain of a relation $\underline{R}$
ran `R`	Range of a relation $\underline{R}$
$S \oplus T$	Replace (or add if does not exist) element T in S
$A \lhd R$	Domain restriction of a relation $\underline{R}\ \{x : X;\ y : Y \mid x \underline{R} y \land x \in A \bullet x \mapsto y\}$
ΔS	State S might be modified
ΞS	State S will remain unchanged

Table 2: Z-specific set-notation

8.3.3 General definitions

Nodes This section describes some general Z definitions, that are used throughout the specifications of all the transformations.

- *VARNAME* set of all valid variable names;
- *VARSYNNODE* set of all syntactic variable nodes;
- *CLAUSENODE* set of all clause nodes;
- *FUNSEMNODE* set of all semantic function nodes;

- *PATTERNNODE* set of all syntactic pattern (or function-argument) nodes.

I described the mentioned sets below in Z-notation. As you can see, I abstracted from RefactorErl by using ranges of Z-identifiers, rather than realistic representations of the elements in the sets since the latter would require me to write Z-specifications for all elements. In the light of the time-window of this thesis, and because identifiers suffice to specify the desired behavior of the transformations (behavior is about what is *done* to the elements, not what they *are*; not in this context anyway), I have abstracted from that.

$$CLAUSENODE == cl_0, ..., cl_n$$

$$VARSYNNODE == vsyn_0, ..., vsyn_n$$

$$VARNAME == vname_0, ..., vname_n$$

$$FUNSEMNODE == fun_0, ..., fun_n$$

$$PATTERNNODE == arg_0, ..., arg_n$$

As was shortly explained earlier in section 6.2.1, there exists a difference between semantic and syntactic nodes in RefactorErl. The semantic nodes contain information about a node like its value, arity (when the node describes a function), bindings and references. The syntactic nodes represent the source code, they have information about token positions in the file, and whether this token represents the definition or a reference to a variable or function. The syntactic nodes are automatically related to their respective semantic nodes, and a semantic node only exists when it is related to at least one syntactic node.

Clause nodes were not explained earlier. RefactorErl maintains syntactic nodes that represent function clauses in the code. Such function clause consists of a clause head and a clause body, separated by $\rightarrow$. A clause head consists of the function name, an argument list, and an optional guard sequence beginning with the keyword *when*. A clause body consists of a sequence of expressions separated by comma (,).[70]

Z-specific definitions Aside from Z representations of RefactorErl-nodes, some definitions are required in order to be able to write down complete specifications. The definition of `REPORT` and the `Success` schema are required to handle inputs to specifications that violate the preconditions.

$$REPORT ::= ok \mid already\_exists \mid invalid\_arg\_number$$

Success

$$result! : REPORT$$

$$result! = ok$$

8.3.4 The initial state of the specifications

The initial state `InitState` of the specification in this thesis is a set of basic functions. Some of them have equivalents in the RefactorErl library (e.g.: `?Var:occurrences()`, `?Var:scopes()`), whereas the others are just helpers, which result from the abstraction from the system. But all of them have an existing equivalent in RefactorErl, which makes it possible to rather easily write tests which can verify whether or not the system conforms to the specification.

The Z-schema shown below displays the definition of the assumed initial state before either of both specified transformations:

InitState

$$
\begin{aligned}
&\mathit{occurrences} : \mathit{VARSYNNODE} \pfun \mathrm{seq}_1\ \mathit{VARSYNNODE} \\
&\mathit{scopes} : \mathit{VARSYNNODE} \pfun \mathrm{seq}_1\ \mathit{CLAUSENODE} \\
&\mathit{scope\_variables} : \mathit{CLAUSENODE} \pfun \mathrm{seq}\ \mathit{VARSYNNODE} \\
&\mathit{var\_name} : \mathit{VARSYNNODE} \pfun \mathit{VARNAME} \\
&\mathit{fun\_clauses} : \mathit{FUNSEMNODE} \pfun \mathrm{seq}\ \mathit{CLAUSENODE} \\
&\mathit{clause\_patterns} : \mathit{CLAUSENODE} \pfun \mathrm{seq}\ \mathit{PATTERNNODE} \\
&\mathit{fun\_arg\_count} : \mathit{FUNSEMNODE} \pfun \mathbb{N}
\end{aligned}
$$

Notice that the schema only defines the signatures of the functions that are used and it lacks their actual implementation. This is due to the limited time to finish this thesis; a certain abstraction is therefore mandatory.

This abstraction requires some of the definitions of these functions to be described in natural language. But given their quite intuitive character they can be described in one sentence, which should minimize the context sensitivity. Additionally, the functions presented here are representations of functions that are actually available in RefactorErl's implementation, in order to keep the specifications related to the project as closely as possible.

- *occurrences:* provided a *variable* syntactic node σ, this function will return a list τ of nodes that refer to σ, where $\tau = \mathrm{seq}_1\ \mathit{VARSYNNODE}$ (analogous to RefactorErl's `?Var:occurrences()`);
- *scopes:* when fed a *variable* syntactic node σ, this function will return a list of scopes (i.e. *clause* nodes), in which σ is referred to. (analogous to `?Var:scopes()`);
- *scope_variables:* handing this function a *clause* node σ results in a list of variables that occur in σ. (analogous to `?Clause:variables()`);
- *var_name:* on input of a *variable*-syntactic node this function returns a string representing its name. (analogous to `?Var:name()`).
- *fun_clauses:* When given a function's semantic node, it will return all its clauses (analogous to `?Fun:clauses()`).
- *clause_patterns:* Returns all patterns (or 'arguments') belonging to the supplied clause-node. (analogous to `?Clause:patterns()`).

- *fun_arg_count:* Returns the arity of the function belonging to the supplied semantic function-node. (analogous to `?Fun:arity()`).

8.3.5 'Variable rename'-refactoring specification

Trivial transformation This refactoring –which renames all occurrences of a particular variable based on an old and new variable name provided by the user– is one of the most trivial transformations of the RefactorErl project, which is the main reason I use it here to explain the meaning of the Z-results of this thesis.

This 'Variable rename'-specification is followed by the 'Function Reorder Parameters'-specification, which is a little more complicated.

Schema The *OkRenameVar* schema tries to update `InitState` schema in such a way that all occurrences of a variable
var_node? are renamed to *new_name?* (input-variables are followed by a question-mark, whereas output-variables are followed by an exclamation-mark). This can be read from the $\Delta InitState$ line, which says that the initial state of this schema is described in the `InitState` schema mentioned earlier. The Δ sign tells that the `InitState` might be modified by this transformation.

Furthermore, only when none of the scopes where the variable occurs already contains a variable with name *new_name?*, `InitState` will be updated.

Originally Zermelo-Fraenkel set-theory defines an axiom ρ[71] for this purpose (i.e. replacing an occurrence of a particular node with a new one), which Z lacks. Instead the specification –inspired by a use-case from the [65, Zbook]– now manually finds all occurrences of a particular variable-node and binds them to the new variable name. The *new_nodes* function has an important role in this, which will be explained in detail in the following paragraph, which focuses on the *definition*-part (lower-part) of the schema.

OkRenameVar

$$
\begin{array}{l}
\Delta InitState \\
new\_nodes : VARSYNNODE \pfun VARNAME \\
var\_node? : VARSYNNODE \\
new\_name? : VARNAME \\
\hline
\forall x : \mathrm{ran}(scopes(var\_node?)) \bullet new\_name? \notin \\
\mathrm{ran}(\mathrm{ran}(scope\_variables(x)) \rhd var\_name) \\
\\
new\_nodes = (\mathrm{ran}(occurrences(var\_node?))) \rhd var\_name \\
\\
(var\_name' = var\_name) \land \\
(\forall x : (\mathrm{ran}(occurrences(var\_node?))) \bullet \\
var\_name'(x) = new\_name?) \\
\\
scopes' = scopes \\
\\
occurrences' = occurrences
\end{array}
$$

This specification tells what to do when a the input conforms to a set of propositions, like: the supplied variable name does not exist yet. But it doe s not tell what to do when it does *not* exist, and I therefore specified the `VarExists`-schema, which handles exactly this situation.

VarExists

$$
\begin{array}{l}
\Xi InitState \\
var\_node? : VARSYNNODE \\
new\_name? : VARNAME \\
result! : REPORT \\
\hline
\exists\, x : \mathrm{ran}(scopes(var\_node?)) \bullet new\_name? \in \\
\mathrm{ran}(\mathrm{ran}(scope\_variables(x)) \lhd var\_name) \\
\\
result! = already\_exists
\end{array}
$$

Now there are two schema's that both handle the same action, but on different situations (i.e. variable does not exist yet, and variable does exist yet respectively). And even more of these schema's can be added until the specification is complete.

These schema's can be combined into one big schema which specifies the entire desired behavior of the *variable_rename* function, like so:

$$RenameVar \mathrel{\widehat{=}} (OkRenameVar \land Success) \lor VarExists.$$

Definitions of OkRenameVar This paragraph explains, in short, the intuitive meaning of the OkRenameVar specification.

$$
\begin{array}{|c|}
\hline
\forall\, x : \mathrm{ran}(scopes(var\_node?)) \bullet new\_name? \notin \\
\mathrm{ran}(\mathrm{ran}(scope\_variables(x)) \lhd var\_name) \\
\hline
\end{array}
$$

Explanation:

- Determine all scopes where *var_node?* is referred to (or defined), and get all the variable nodes from these scopes. Get their corresponding variable names, and make sure that none of them equals *new_name?* as to avoid name clashes.
- More literally: restrict the domain of *var_name* to the variable nodes which occur in all scopes *var_node?* is part of and make sure none of the nodes is named *new_name?*.

$$\boxed{\begin{array}{c}(var\_name' = var\_name) \land \\ (\forall\, x : (\mathrm{ran}(occurrences(var\_node?))) \bullet var\_name'(x) = new\_name?)\end{array}}$$

Explanation:

- all nodes in *new_nodes* are assigned new name (*new_name?*). (i.e. each x in the domain of *new_nodes* is mapped to a new value).

$$\boxed{scopes' = scopes}$$
$$\boxed{occurrences' = occurrences}$$

Explanation:

- *scopes* and *occurrences* must remain unaltered.

Definitions of VarExists This paragraph explains, in short, the intuitive meaning of the VarExists specification. This specification describes how to handle the situation where a user hands an already existing name for a variable.

$$\boxed{\begin{array}{c}\exists\, x : \mathrm{ran}(scopes(var\_node?)) \bullet new\_name? \in \\ \mathrm{ran}(\mathrm{ran}(scope\_variables(x)) \lhd var\_name)\end{array}}$$

Explanation:

- There exist at least one a variable named *new_name?* in any of the affected scopes.

$$\boxed{result! = already\_exists}$$

Explanation:

- There already is a variable that goes by the name *new_name?*, so return *already_exists*, and leave *InitState* unaltered (as described by Ξ).

Combining schema's Now there exist two schema's which handle two separate situations, these are combined in the *RenameVar*-schema.

$$RenameVar \mathrel{\widehat=} (OkRenameVar \land Success) \lor VarExists.$$

Explanation:

- If the supplied variable name does not conflict with that of any existing variables, then *OkRenameVar* will hold, *Success* always holds, and returns *ok*.
- When the supplied variable *does* clash, it *OkRenameVar* will not hold, but *VarExists* does. This results in *already_exists*, and nothing will be changed.

8.4 Function Reorder Arguments Specification

8.4.1 What does this specification describe?

This is a partial Z-specification of the already available natural language specification of the Function Reorder Arguments transformation in RefactorErl. This transformation reorders the arguments of a given function, to a given order. It not tested very thoroughly (only on type and syntactic correctness using *fuzz* and *jaza*), and is mainly added for illustration purposes.

8.4.2 Initial state

The implementation of the transformation in RefactorErl, uses already existing RefactorErl functions. *fun_args* and *fun_args_count* in the *InitState* schema represent `[?Query:exec( <querytofunction>, ?Clause:patterns())]` and `?Fun:arity` respectively. The *fun_args_count* function returns the number of arguments function represented by the supplied semantical node has (this is defined below the line).

8.4.3 Reordering function parameters

The *ReorderFunPar* schema defines the requirements of this transformation, and how the reordering works. It should be noted that it is assumed that *before* the action, the graph was correct.

Input to the transformation consists of a function's semantical node, and a list of integers, which represent the new order of arguments (e.g. [3,1,2], [1,4,3,2]).

The first requirement is that the supplied order is correct, so the specifications requires that the number of items in the supplied list is equal to the actual number of arguments to the function. After that, it is checked that all the items in the list are a unique number between 1 and the arity of the supplied function.

When the preconditions are met, the functions arguments (i.e. all patterns of all clauses) are reordered with respect to the supplied list.

OkReorderFunPar

$\Delta InitState$
$what\_fun? : FUNSEMNODE$
$apply\_order? : \operatorname{seq} \mathbb{N}$

$\operatorname{ran} apply\_order? = (1 \,..\, fun\_arg\_count(what\_fun?))$

$\forall\, x : \operatorname{dom} apply\_order?;\ y : \operatorname{ran}(fun\_clauses(what\_fun?)) \bullet$
$(clause\_patterns'(y)(x) = clause\_patterns(y)(apply\_order?(x)))$

As already was the case with the *rename_var* transformation, situations can occur in which the preconditions for the *function_rename*-transformation are not met. These situations require different schema's that can handle them. *InvalidArgNumber* handles the situations where the supplied list does not have the correct length, or does not contain the right numbers.

InvalidArgNumber

$\Xi InitState$
$what\_fun? : FUNSEMNODE$
$apply\_order? : \operatorname{seq} \mathbb{N}$
$result! : REPORT$

$\operatorname{ran} apply\_order? \neq (1 \,..\, fun\_arg\_count(what\_fun?))$

$result! = invalid\_arg\_number$

And finally, the *ReorderFunPar* schema is defined, which combines the several schema's presented before.

$$ReorderFunPar \mathrel{\widehat{=}} (OkReorderFunPar \land Success) \lor InvalidArgNumber.$$

OkReorderFunPar This paragraph contains a short explanation for the specification of the *OkReorderFunPar*-schema.

$$\boxed{\operatorname{ran} apply\_order? = (1 \,..\, fun\_arg\_count(what\_fun?)).}$$

Explanation:

- The list with the new order *apply_order?* must contain exactly the numbers [1 .. arity]. These numbers can have any random order.

$$\forall x : \text{dom } apply\_order?;\ y : \text{ran}(fun\_clauses(what\_fun?)) \bullet$$
$$(clause\_patterns'(y)(x) = clause\_patterns(y)(apply\_order?(x)))$$

Explanation:

- All clauses of *what_fun?* are gathered, and for each one its patterns are swapped with the desired pattern (as described by *apply_order?*).

InvalidArgNumber This paragraph contains a short description of how the *InvalidArgNumber*-schema works.

$$\text{ran } apply\_order? \neq (1 \,..\, fun\_arg\_count(what\_fun?)).$$

Explanation:

- The list with the new order *apply_order?* does not contain exactly the numbers [1 .. arity] in any random order.

$$result! = invalid\_arg\_number$$

Explanation:

- When the precondition was met, it means *apply_order?* has an incorrect value, and hence *invalid_arg_number* is returned, and everything remains unchanged (note the Ξ before *InitState*).

Combining schema's Like with the *rename_var* transformation, the schema's need to be combined, which results in the *ReorderFunPar*-schema.

$$ReorderFunPar \mathrel{\widehat{=}} (OkReorderFunPar \land Success) \lor InvalidArgNumber.$$

8.5 From specifications to QuickCheck

A tester can use Erlang to manually code a model along the lines of the Z-specifications. The tester will have to write code that loads and generates test data (e.g. source code, nodes, function names), and then he can check if the separate properties hold, by defining them using QC syntax (section 6.2.2).

Another approach can be to write or use a tool which generates a model or (abstract) test cases from the specifications. Of course, writing such tool is a lot of work in itself, and might turn out not being worth the effort. Besides,

this tool will need thorough testing as well, which comes down to the same problem you are trying to solve. Furthermore, I did only find one third party tool which claims to generate abstract test cases, but I could not get it to work (section 5.6.1). Therefore, it seems that one can only manually translate the specifications into QC code for now.

8.6 Interesting constructs selection

8.6.1 'Dynamic function call'-bug

This section elaborates on a 'bug' I found, by walking through the Erlang manual in a quest for little-known or -used, and hence interesting constructs. I quote 'bug', because it turned out not to be a bug as much as it is a deliberately unsupported construct (one of the main reasons being that this construct is marked deprecated in the Erlang documentation). It is, however, a perfect example of a 'false positive' that is not detected by my model tests, resulting from my assumption of the lower layers being correct.

The construct I found is a 'dynamic function call': a function call, in the form of a tuple. For example, snippet 15 contains two lines of code that mean the same thing, the first line being a regular function call, and the second representing its dynamic notation.

```
1 modulename:f().
2 {modulename,f}().
```

Snippet 15: Regular vs. dynamic function call

When considering the code in snippet 16 below, and one renames function `g/1` to `h/1`, all the calls but the dynamic call will be updated, violating the number 1 property for all refactorings: when code compiles before, it must compile afterward, which it does not, because there exists no function `g/1` anymore.

```
1  -module(fun_rename_dynfunbug).
2
3  -export([f/0, g/1]).
4
5  f() ->
6      {fun_rename_dynfunbug,g}(6).
7
8  % when 'g' is renamed, the dynamic function call to 'g
       ' in 'f' is not updated.
9  g(Number) ->
10     Number * 7.
```

Snippet 16: Dynamic function rename bug example

Then why is this problem not revealed by my QC test? Well, that is because in my model I use RefactorErl's upper layer for querying the database. So, when renaming function `f/0` to `g/1`, I use this layer extensively to collect all the occurrences of function `f/0` before and function `g/0` after, which will then

be compared on their binding structures, which should be the same (i.e. the binding structure remained unchanged).

But since the lower layers, and hence the upper layer, do not recognize dynamic function calls as being actual function calls, the test will never include them in the comparison in the first place, and therefore the test does not detect that the dynamic call was not updated.

However, should dynamic function calls in the future be supported by the lower layers, the test will –without modification– include the dynamic function calls in its comparison, and as a result reveal potential defects concerning dynamic calls.

8.6.2 'Function rename record'-bug

Erlang contains a `record` construct, which is described in the Erlang Reference Manual[70] as follows: a record is a data structure for storing a fixed number of elements. It has named fields and is similar to a *struct* in C. A record definition consists of the name of the record, followed by the field names of the record. Each field *can be given an optional default value.*

```
1 -record(Name, {Field1 [= Value1],
2                ...
3                FieldN [= ValueN]}).
```

Snippet 17: Example of a record definition

The following expression creates a new *Name* record where the value of each field *FieldI* is the value of evaluating the corresponding expression *ExprI*:

```
1 #Name{Field1=Expr1,...,FieldK=ExprK}
```

Snippet 18: Example of creating a record

As stated earlier, I consider nested constructs to be the most plausible location for finding defects in the implementation of RefactorErl, so I defined a record *ftr* in a module, a construct which I noticed is not used very often in the RefactorErl project, and assigned as the default value of its *a*-field a function `g/0` that is defined in its module, like so:

```
 1 -module(fun_rename_recordbug).
 2
 3 -export([f/0]).
 4
 5 -record(ftr,{a=g()}).
 6
 7 f() ->
 8     g(),
 9     Z = #ftr{},
10     Z#ftr.a.
11
12 %when 'g' is renamed, the #ftr.a field is not updated.
13 g() ->
14     42.
```

Snippet 19: Function rename record bug example

When the rename function transformation is applied to function `g/0`, all nodes are updated accordingly, with exception of `#ftr.a`'s default value.

The reason that this bug was introduced most probably resides in the fact that records are usually defined in import modules as 'global-constructs', rather than in a 'normal' module (and hence no one spotted the problem).

An interesting detail is that in the 'import module'-situation the assignment of a certain function is ambiguous. For example, when there are two modules `x` and `y` that both contain a function `g/0`, and a header file that is included in both modules that defines `#ftr` in the same way as snippet 19 does, then `#ftr.a` refers to both `x:g/0` and `y:g/0`. So, it is impossible for RefactorErl to update the field to the right value, for such value does not exist. It would for example be better to throw an exception informing the user of this problem, rather than just ignoring this construct though.

The situation described in snippet 19, where the record is used in just a single module, is unambiguous though. Therefore it *is* a bug.

9 Discussion

The main results of my research is a working approach of testing the transformations of RefactorErl using QuickCheck. This approach consists of several steps: specification, Model Based Testing, gathering complex constructs as input to MBT. Each of these steps has its own results, which are both an explanation and a proof that this approach is useful for testing RefactorErl.

9.1 MBT vs Unit testing

Prior to this research, the RefactorErl project was tested by means of unit tests. This is a quite structured approach of testing, and is often used to test boundary

values and gain certain code coverage percentage. Also, unit testing is designed to test small parts of a system, like separate classes or modules for example.

To achieve a closer-to-testing-completeness result, it is useful to apply multiple testing techniques with different philosophies. This makes MBT a suitable candidate, because it takes a different approach in testing: rather than testing boundaries and defining static tests, it uses generated inputs which may test many more cases.

The usefulness of complementary testing techniques is demonstrated by the defects that went undetected by unit tests, but were revealed by MBT.

9.2 Z-specifications

The specifications in this thesis are merely an example of how Z-specifications can be applied in the RefactorErl project. As I mentioned several times in the text already, I am convinced that it is important to define the current natural language specifications in some formal-notation, and I think Z is a very suitable notation to use to this end.

One of the merits of having a formal specification, is that model based tests can easily be derived from them. This can even be done with the simplified specifications that I defined in this thesis. When all functions that are used by the specifications are specified as well, the currently existing specifications can be extended to be more accurate. Properties can then be specified that are related to more detailed information like particular node-properties for example.

When everything, or all 'crucial' parts are specified, the result is a tool which can be useful for several researches in the future (e.g. partial formal proof of the system; expanding the MBT-set and making it more thorough; deriving models).

9.3 Selected constructs

Resulting from the lack of an Erlang code generator, for the inputs to QC MBT, I gathered a few constructs that caused problems with RefactorErl by browsing through Erlang's documentation and using some common-sense (e.g. nested operations are more likely to cause problems than a simple arithmetic operation). The result is a small set of modules that demonstrate the simplest occurrence of a particular problem. I have discarded the constructs that I suspected to be troublesome, but did not reveal any problems, because the value of these results cannot be tested.

The resulting set of modules demonstrates problems with transformations of obscure constructs that are either marked for removal in some future version of Erlang (e.g.: dynamic functions); not likely to be used in the practice (e.g.: function call in definition of a record); or still under construction in RefactorErl (e.g.: checking for side-effects in functions).

These results are not too compelling on themselves, but they are useful to demonstrate that defined MBT's are able to spot any defects at all. It is also

useful to add them to the regression test-set, after defects have been solved, to ensure that these defects are abandoned in the future.

9.4 Improving the results

As I have stated before, all my tests most probably return some false negatives, which, as I explained, can only be solved by either MBT-ing each layer or creating an implementation independent model.

Therefore, the results gain more value by extending the focus of testing to underlying layers as well, or by creating fully implementation-independent models. It should be noted that the latter might be both unfeasible and a more error-prone approach, though.

9.4.1 Fully random input to MBT

The results of MBT (defects detected) were gathered by supplying 'interesting' code to the tests, which results in more test-cases that actually make sense. Also, this is a more sophisticated approach than just 'brute-force' testing by feeding the MBT random data.

This means however, that some test-cases that reveal defects might be omitted. For example: testing a *rename_variable* refactoring to a module which contains no variables at all might result in an error, but this will never be detected by the 'sophisticated' approach however, because it does not make enough sense to test it.

Hence, to strengthen the result-set, it is useful to use fully randomly generated Erlang code (i.e. unbiased code) as input to the MBT on a regular basis. MBT is perfect for this purpose, because it costs no effort to leave such test to work unattended for a long time-span (e.g. several days).

10 Conclusion

10.1 Main goal

The main part of my research consisted of model based testing the implementation of RefactorErl using QuickCheck. My goal was to propose an approach of testing the transformations defined RefactorErl, and to prove that this approach is successful.

10.2 Proposed approach

The proposed approach consists of formalizing the specifications of transformations, which are currently only defined in natural language; create models based on these specifications; collect as many as possible code snippets that show undesired behavior under a particular transformation (and simplify them as far as possible). And finally use all these ingredients to test the implementation of

RefactorErl, by testing implementation against models and code snippets using QuickCheck.

10.2.1 Formalizing specifications

I have successfully specified several of these transformations based on their natural language definitions. That is: I used Z-notation to define them, the way I interpreted them, and tested them using a type-checker and an animator. This does not mean that these specifications are correct, for I could have misinterpreted the natural language definitions.

Therefore, reviewing is required before they can be used for non-experimental testing-purposes. But once a transformation in specified, it can be used for all testing-purposes in the future; the specifications can be used for theorem proving; are interpretable by computers, and can therefore be used for such purposes as generating models.

10.2.2 MBT using QuickCheck

Based on the specifications I have defined some QuickCheck tests. I have revealed one defect this way, and the tests are able to detect several others which were revealed in another step of the proposed approach.

Therefore both these tests and the proposed approach have proved their use, albeit not too compelling yet; I expect a lot more defects to be revealed when all layers are tested.

On a final note, MBT is by no means a (total) replacement for other types of testing. It might be too much effort for example to write models for the bigger part of the system. Due to its shortcomings it cannot completely test a system on itself; it should be complemented by the already existing unit tests for example, which has its own shortcomings.

10.2.3 Complex constructs

This part of the research revealed several defects in the system. I have used these results to verify whether or not my MBT's detect them as well.

It has proved to be quite useful as a surrogate for an actual code generator. And it is also a good means to prove the defined tests are (at least partially) correct; and a good extension to the set of regression-test modules.

10.3 Overall conclusion

From this research I conclude that applying MBT to RefactorErl is useful. I also consider the formalization of transformation-specifications an interesting addition to the project, for they are a simple means for improving on all tests, models and implementations that will be written in the future. Also, they make way for new approaches in testing.

The overall approach I proposed seems to work out well, although the MBT-part can be improved on, as I mentioned several times.

11 Acknowledgments

Most of the code in this thesis came from my hand, but in some parts I extensively made use of pre-existing code written by István Bozó, either by incorporating modified parts of it to my own code, or by moving particular (very generic) functions to shared libraries which I would then call from my own code.

Furthermore, the idea for comparing the Erlang syntax tree to its RefactorErl equivalent in order to ensure that these are equal at every moment, was coined by László lövei.

A Syntax-tree comparison

In order to familiarize myself with both Erlang and RefactorErl, I did some introductory research. This consisted of comparing the syntax tree of RefactorErl to the code in the actual file, which should at all times be the same. It is very well possible that there reside defects to be detected in this part of the system, because RefactorErl has its *own* Erlang parser, which is used to store Erlang source code in an MNesia database.

Rather than writing my own parser, I used *syntax_tools*, which is included with Erlang. This system uses the actual Erlang parser (which I assume to be correct) the parse a file and store it in a tree. It also contains a lot of functions that can be used to access this tree.

The general approach I applied was to take a file, and load it with RefactorErl and just walk from the root of the three through all the nodes. With every step in the RefactorErl tree I then make the same step in the tree as presented by the *syntax_tools*. If this succeeds, then the RefactorErl tree must at least be a sub tree of the *syntax_tools* tree. But to be sure that they are one and the same tree, the trees also need to be walked the other way around (i.e. walk the RefactorErl tree based on the *syntax_tools* tree.

Unfortunately though, due to my lack of a thorough understanding of Erlang and RefactorErl it took quite some time to implement the one-way test. Given the limited time-window for my entire research, I did not get to implement the test the other way around.

It might be useful to further/reimplement this idea though, because it seems like a feasible thing to do. And due to the different parsers, defects seem almost inevitable.

Glossary of Terms

BIF	A function that is part of the Erlang core, a Built-In Function	26
DRY	Don't Repeat Yourself	24
edoc	EDoc is the Erlang program documentation generator. [It] lets you write the documentation of an Erlang program as comments in the source code itself.	22
Erlang/OTP	The open telecom platform (OTP) is a development system platform for building telecommunications applications, and a control system platform for running them. The platform, whose aim is to reduce time to market, enables designers to build - from standard, commercially available computer platforms - a highly-productive development environment that is based on the programming language Erlang.[72]	22
Git	Source Code Management system	29
IMRaD	Introduction, Methods, Results and Discussion	23
Model Based Testing	Testing the implementation of a software system against a model.	22
monad	Code construct for which the definition of 'building-block for workflows' suffices in the context of this thesis. Haskell uses them to preserve its purity, banning all side-effect operations to monads.	26
side-effect	Change in state caused by a function	26
struct	Structured (record) type that aggregates a fixed set of labelled objects, possibly of different types, into a single object.	61

References

[1] Örian Sölvell & Michael E. Porter. Finland and nokia, 2002. http://www.exchange.unisg.ch/org/lehre/exchange.nsf/1176ad62df2ddb13c12568f000482b94/5d58eda8f1f3dc90c1256fa3005ebc52/$FILE/10.%20Finland%20and%20Nokia%20(9-702-427).pdf (retrieved June 1st, 2010).

[2] Frost & Sullivan. Global gsm market analysis, 2009. http://www.frost.com/prod/servlet/cio/180573216 (retrieved May 29th, 2010).

[3] Elroy Jumpertz. Using quickcheck and semantic analysis to verify correctness of erlang refactoring transformations. Master's thesis, Radboud University Nijmegen, 2010.

[4] Ericsson. History of ericsson, 2010. http://www.ericssonhistory.com (retrieved May 13th, 2010).

[5] Ericsson. Corporate information, 2010. http://www.ericsson.com/ericsson/corpinfo/ (retrieved May 13th, 2010).

[6] Ericsson. Company structure and organization, 2008. http://www.ericsson.com/thecompany/investors/financial_reports/2008/annual08/company-structure-and-organization.html (retrieved August 14th, 2010).

[7] Ericsson. Company structure and organization, 2009. http://www.ericsson.com/thecompany/investors/financial_reports/2009/annual09/sites/default/files/downloads/en/Complete_annual_report_2009_EN.pdf (retrieved August 14th, 2010).

[8] Telefonaktiebolaget LM Ericsson. This is ericsson, 2008. http://www.ericsson.com/ericsson/corpinfo/doc/this-is-ericsson.pdf (retrieved May 31st, 2010).

[9] Hoover's Inc. Ericsson inc. — company profile from hoover's, 2010. http://hoovers.com/company/Ericsson_Inc/cyxyxi-1.html (retrieved June 1st, 2010).

[10] Ericsson. Ericsson iptv solution - enabling a new kind of iptv, 2009. http://www.ericsson.com/ericsson/press/facts_figures/doc/iptv.pdf (retrieved August 6th, 2010).

[11] Ericsson. Personalized greeting service (pgs), 2009. http://www.ericsson.com/ourportfolio/products/personalized-greeting-service-pgs (retrieved August 6th, 2010).

[12] International Directory of Company Histories. Telefonaktiebolaget lm ericsson, 2002. http://www.fundinguniverse.com/company-histories/Telefonaktiebolaget-LM-Ericsson-Company-History.html (retrieved June 3rd, 2010).

[13] Funding Universe. Telefonaktiebolaget lm ericsson – company history, 2004. http://www.fundinguniverse.com/company-histories/Telefonaktiebolaget-LM-Ericsson-Company-History.html (retrieved August 6th, 2010).

[14] John M. Shea. From the birth of telecommunications to the modern era of cellular communications and wireless local area networks, 2010. http://wireless.ece.ufl.edu/jshea/wireless_history.html (retrieved August 6th, 2010).

[15] Ericsson. Ericsson celebrates 50 years of mobile telephony, 2006. http://www.ericsson.com/news/1081237 (retrieved August 6th, 2010).

[16] Amit S. Mukherjee. *Spider's Strategy, The: Creating Networks to Avert Crisis, Create Change, and Really Get Ahead*, chapter 1. 2009. http://www.ftpress.com/articles/article.aspx?p=1244469 (retrieved May 20th, 2010).

[17] Ericsson. The multi-access ip edge: Key to delivering fmc, 2008. http://archive.ericsson.net/service/internet/picov/get?DocNo=4/28701-FGB101504&Lang=EN&HighestFree=Y (retrieved August 13th, 2010).

[18] Ericsson. Our business offerings, 2008. http://www.ericsson.com/ericsson/corpinfo/compfacts/offering.shtml (retrieved August 6th, 2010).

[19] Ericsson. Managed services, 2010. http://www.ericsson.com/ourportfolio/services/managed-services (retrieved August 13th, 2010).

[20] Ericsson. Erlang: the movie, 1990. http://www.archive.org/details/ErlangTheMovie (retrieved June 1st, 2010).

[21] Joe Armstrong. History of erlang, 2007. http://www.cs.chalmers.se/Cs/Grundutb/Kurser/ppxt/VT2009/general/languages/armstrong-erlang_history.pdf (retrieved June 1st, 2010).

[22] Corcky Cartwright. What is concurrent programming?, 2000. http://www.cs.rice.edu/~cork/book/node96.html (retrieved August 17th, 2010).

[23] Elliotte Rusty Harold. Threads vs. processes, 2005. http://www.cafeaulait.org/course/week11/02.html (retrieved August 17th, 2010).

[24] Ruben Vermeersch. Concurrency in erlang & scala: The actor model, 2009. http://ruben.savanne.be/articles/concurrency-in-erlang-scala (retrieved August 6th, 2010).

[25] Ericsson. History of erlang, 2001. http://www.erlang.org/course/history.html (retrieved June 2nd, 2010).

[26] Ericsson. Implementation and ports of erlang, 2010. http://www.erlang.org/faq/implementations.html (retrieved August 17th, 2010).

[27] Ericsson. What is erlang, 2010. http://www.erlang.org/faq/introduction.html#id49852 (retrieved August 17th, 2010).

[28] ELTE PLC. Erlang refactoring, 2010. http://plc.inf.elte.hu/erlang/.

[29] Joe Armstrong, Bjarne D äcker, Thomas Lindgren, Håkan Millroth, and Ericsson production team. Erlang white paper, 2009. http://erlang.org/white_paper.html.

[30] Pete Ellertsen. Imrad tip sheet, 2010. http://www.sci.edu/faculty/ellertsen/imrad.html.

[31] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, first edition, 2007.

[32] ELTE PLC. Refactorerl 0.6.2 tutorial, 2009. http://plc.inf.elte.hu/erlang/dl/tutorial-0.6.2.pdf.

[33] Raymond P.L. Buse and Westley R. Weimer. A metric for software readability. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130. ACM, 2008. http://www.cs.virginia.edu/~weimer/p/weimer-issta2008-readability.pdf.

[34] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison-Wesley Professional, first edition, 2000.

[35] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison Wesley, first edition, 1999. http://www.pragprog.com/titles/tpp/the-pragmatic-programmer.

[36] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[37] Danijel Arsenovski. Debunking common refactoring misconceptions, 2008. http://www.infoq.com/articles/RefactoringMyths.

[38] Jeff Grigg. If it is working dont change, 2008. http://c2.com/cgi/wiki?IfItIsWorkingDontChange.

[39] Gabor Szabo. What does "if it ain't broke, don't fix it" really mean?, 2009. http://szabgab.com/blog/2009/11/1259431123.html.

[40] Manuel Antonio Aldana. Bashing the refactoring criticism, 2008. http://www.aldana-online.de/2008/04/17/bashing-the-refactoring-fears-practice-refactorings/.

[41] Huiquing Li, Simon Thompson, László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Vg, and Tamás Nagy. Refactoring erlang programs. In *Proceedings of the 12th International Erlang/OTP User Conference, EUC 2006*, November 2006. 10 pages.

[42] László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Vg, and Tamás Nagy. Refactoring erlang programs. Conference poster at High Speed Networking Workshop, May 2006.

[43] Tamás Nagy and Anikó Vg. Erlang refactoring with relational database. Presentation in University of Kent Functional Program Group Meeting, Canterbury, UK, December 2006. 25 slides,`http://plc.inf.elte.hu/erlang/pub/refac_db_kent.ppt`.

[44] Konstantinos Sagonas and Thanassis Avgerinos. Automatic refactoring of erlang programs. In *PPDP '09: Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 13–24, New York, NY, USA, 2009. ACM. `http://dx.doi.org/10.1145/1599410.1599414`.

[45] Inc. etBrains. Resharper:: The most intelligent add-in to visual studio 2005, 2008, and 2010 - c# 3.0, linq, vb.net, asp.net, xml, xaml, build scripts. best-of-breed tools for code analysis, code cleanup, code generation, and unit testing, plus multiple refactorings and code templates., 2010. `http://www.jetbrains.com/resharper/`.

[46] University of Kent. Hare – the haskell refactorer, 2009. `https://www.cs.kent.ac.uk/projects/refactor-fp/hare.html`.

[47] Tamás Nagy and Anikó Nagyné Víg. Erlang testing and tools survey, 2008. `http://www.erlang-solutions.com/erlangworkshop08/erlang08m-nagy.pdf`.

[48] Huiqing Li and Simon Thompson. Testing Erlang Refactorings with QuickCheck. In *the 19th International Symposium on Implementation and Application of Functional Languages, IFL 2007, LNCS*, Freiburg, Germany, September 2007. `http://www.cs.kent.ac.uk/pubs/2007/2648`.

[49] Melinda Tóth. Erlang Refactoring: Extract Function. Bachelor thesis, ELTE, Budapest, Hungary, 2008.

[50] Istvn Boz and Melinda Tth. Restructuring Erlang programs using function related refactorings. In *Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, pages 162–176, Tampere, Finland, August 2009.

[51] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to haskell: Io, 2000. `http://www.haskell.org/tutorial/io.html`.

[52] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall PTR, second edition, 1996. http://www.erlang.org/download/erlang-book-part1.pdf.

[53] Charles Simonyi. Hungarian notation, 2009. http://msdn.microsoft.com/en-us/library/aa260976(VS.60).aspx (retrieved August 2nd, 2010).

[54] Erlang/OTP community. Erlang's otp at dev, 2010. http://github.com/erlang/otp.

[55] Huiqing Li and Simon Thompson. Formalisation of haskell refactorings. In *Trends in Functional Programming*, pages 95–110, 2005.

[56] Steve McConnel. *Code complete*. Microsoft Press, second edition, 2004.

[57] Thales e Security. Secure generic sub-system (sgss), 2005.

[58] Alistair A. McEwan and J. C. P. Woodcock. A verified implementation of a control system, 2005.

[59] Flowgate Security Consulting. Fastest, 2010. http://formalmethods.wikia.com/wiki/Z_notation#Fastest.

[60] Maximiliano Cristi'a, Valdivino Santiago, and N. L. Vijaykumar. On comparing and complementing two mbt approaches, 2009. http://www.fceia.unr.edu.ar/~mcristia/publicaciones/inpe-cifasis.pdf.

[61] Quviq AB. Quviq quickcheck for erlang, 2009. http://www.cs.chalmers.se/~rjmh/ErlangQC/.

[62] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck, 2006. http://www.quviq.com/documents/erlang001-arts.pdf.

[63] Pieter Koopman and Rinus Plasmeijer. Systematic synthesis of λ-terms, 2007.

[64] Quviq. Software testing with quickcheck, 2009. http://www.protest-project.eu/upload/tutorials/TestingToolsWs/Software_Testing_with_QuickCheck1of3.pdf (retrieved June 23rd, 2010).

[65] Jonathan Bowen. Formal specification and documentation using z: A case study approach, 2003. http://sites.google.com/site/jpbowen/zbook.pdf.

[66] S.G.K. Murthy and K. Raja Sekharam. Software reliability through theorem proving. *Defence Science Journal*, 59(3):314–317, 2009.

[67] Z User Group. Z user group, 2010. http://www.zuser.org/.

[68] Mike Spivey. The fuzz manual, 2000. http://spivey.oriel.ox.ac.uk/mike/fuzz/fuzzman.pdf.

[69] Mark Utting. The jaza animator, 2010. http://www.cs.waikato.ac.nz/~marku/jaza/.

[70] Ericsson AB. Erlang reference manual user's guide, 2009. http://www3.erlang.org/doc/reference_manual/users_guide.html.

[71] Stanford Encyclopedia of Philosophy. Zermelo-fraenkel set theory, 2010. http://plato.stanford.edu/entries/set-theory/ZF.html.

[72] Seved Torstendahl. Open telecom platform. *Ericsson Review*, (1), 1997. http://www.erlang.se/publications/ericsson_review_otp_1997012.pdf.