STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVI**, Number 3, 2011

# USING IMPACT ANALYSIS BASED KNOWLEDGE FOR VALIDATING REFACTORING STEPS

ISTVÁN BOZÓ, MELINDA TÓTH, MÁTÉ TEJFEL, DÁNIEL HORPÁCSI, RÓBERT KITLEI, JUDIT KŐSZEGI, AND ZOLTÁN HORVÁTH

Abstract. In the life cycle of a large software product the source code often has to be changed to fit to new requirements, which can be aided by refactorings. In order to minimise the possibility of breaking the functionality of the code, various test suites are used. In this paper, we present a method for examining which test cases are affected by doing a refactoring on the code. If we find that the outcome of a particular test case is not changed by the refactoring, the test case does not need to be run, which might make testing more cost effective. Also, we validate the refactoring by checking that the affected functions behave identically pre and post the transformation. This system is based on a language specific model that is designed to capture knowledge about the software product acquired using impact analysis.

## 1. Introduction

The refactoring [5] tools try to preserve the properties of the transformed programs using language specific semantic knowledge about the source code and to ensure the safety of statically performed transformations. In case of industrial sized software and some programming languages the usage of complex static analysis sometimes is not enough to decide whether the transformation is safe, and some rarely used language elements can make it impossible. Therefore it is crucial to retest the system after a major change.

In case of the industrial sized software components the developers prefer to perform the necessary refactorings manually against applying the automatic refactoring, even if the manual refactoring is more error prone. The main reason for this situation is based on the fact that the developer can not follow the changes performed on the source code by the refactoring tool. In order to

Received by the editors: April 10, 2011.

2010 *Mathematics Subject Classification.* 68Q99.

1998 *CR Categories and Descriptors.* [**F.3.2**]: Theory of Computation, Logics and Meanings of Programs, Semantics of Programming Languages – *Program analysis*.

*Key words and phrases.* Erlang, regression test, refactoring validation, property generation.

convince the developer about the reliability of refactorings we have to analyse the impact of the transformations while reducing the cost of the testing process. By using impact analysis we can select the code parts that are affected by a transformation and based on this knowledge we can minimise the cost of regression test.

Our research focuses on Erlang [4], a dynamically typed language. Many of its semantic rules are also dynamic, that makes the static semantic knowledge based analysis hard. In this paper, we examine the various connections between refactorings of the code and test cases. We explore the possibility of leaving some test cases out if we can argue that the refactoring does not change the outcome of the test. We also investigate how we can give further guarantees that the refactoring has successfully transformed the source code using random parameter space testing and metrics based validation.

The paper is structured as follows. In Section 2, we show a motivating example. In Section 3 we present the tool that will store the knowledge we gather about Erlang source code; Section 4 describes change impact analysis based property selection; Section 5 uses the same analysis to find functions that are affected by a code transformation and introduce a validation property. Section 6 discusses related work and Section 7 concludes the paper.

## 2. Motivating example

```
1  -module(exampl).
2  -export([calc/2, prop1/0, prop2/0]).
3
4  calc(X,Y)->
5      Mul = X*Y,
6      Sum = X+Y,
7      {Mul, Sum}.
8
9  prop1()->
10     ?FORALL({A, B},
11             {int(), int()},
12             element(1, calc(A, B)) =:= A*B).
13 prop2()->
14     ?FORALL({A, B},
15             {int(), int()},
16                 element(2, calc(A, B)) =:= A+B).
```

Figure 1. A module with two QuickCheck properties

```
1  calc(X,Y)->
2      Mul = mul(X,Y),
3      Sum = X+Y,
4      {Mul, Sum}.
5
6  mul(X,Y) ->
7      X*Y.
```

FIGURE 2. A part of **calc** is extracted into **mul**

In Figure 1, we see an Erlang module with a function `calc` that takes two parameters. There are two statements `prop1` and `prop2`, formalised as QuickCheck properties [10]. The function itself calculates the sum and the product of its arguments, and returns the pair of the calculated values.

The first property states that the first element of the returned tuple contains the product of the two arguments. For this, it uses a pair of two generators (`int()`) and binds the generated values to `A` and `B` respectively. If QuickCheck is installed, the module test case can be invoked by calling `eqc:quickcheck(exampl:prop1())`, which will use the generators to produce 100 random pairs of integers, and tests the property. The second property, quite similarly, tests the second component of the return value of the function `calc`.

There are several refactorings that can be applied to the function `calc`. As `calc` is not a good name, one idea is to rename the function. Another possibility is to extract a portion of the function `calc` into another function. For example, the production on line 5 may be extracted to a new function `mul` as seen in Figure 2. The use of this refactoring could change the first element of the return value of function `calc` and does not have any affect on the second element. Since property `prop2` uses only the second element of the return value, it does not depend on the changed source code and it is unnecessary to retest it.

## 3. Layers of knowledge for impact analysis

3.1. **Source code representation.** RefactorErl [7] is a source code analyser and transformer tool for Erlang [4]. It builds an abstract syntax tree over the source code, then uses several semantic analysers that enrich it into a Semantic Program Graph (SPG) by adding various other levels of knowledge. Information gathered by the analysers include connections between function applications and definitions (revealing function calls), between definitions and uses of variables etc.

The Semantic Program Graph of RefactorErl is capable of storing and efficiently retrieving information from the represented Erlang source code. While it is possible to directly use the SPG to retrieve the information necessary for impact analysis, directly accessing the SPG is too costly. It is more efficient to build a more compact and focused intermediate representation of the source code first. Therefore, we use the SPG to calculate advanced representations such as control, data and behaviour dependency graphs, and we calculate the impact of a refactoring change on dependency graphs.

Changing the source code (by tool assisted semi-automatic refactoring, or manually editing it) affects those program parts that depend on the changed expressions. We grasp knowledge about such connections by building a *Dependency Graph* where

- a **node** represents an Erlang expression and
- an **edge** represents a possible dependency between two expressions.

A change spreads in the source code with different kinds of dependencies: control, data, behaviour etc. We build several flow graphs to calculate these dependencies: Data-flow Graph [9, 13], Control-flow Graph [14] and Behaviour Dependency Graph [15].

3.2. **Dependency Graph - DG.** Working with flow graphs, in order to determine real dependencies is not efficient, as it requires a number of traversals in the CFG for every expression. Thus we use the well known approach used at compilers, we build a dependency graph that eliminates the unnecessary sequencing and includes only direct dependencies.

In building the DG we follow a compositional approach [12]. First we determine the affected functions by performing a transitive closure on the function call graph, starting from the selected functions. We build the CFG for every function from this set separately. These CFGs are intrafunctional, as these do not follow the function applications and message passing. Then from the obtained CFGs we build postdominator trees (PDT) and control dependency graphs (CDG). The next step is the composition stage of the obtained CDGs. In this stage the function application and message passing edges are resolved. With these steps we obtain the composed control dependency graph.

This graph contains only the control dependency edges, some other useful information is necessary to determine real dependencies among the expressions. We extend this composed CDG with data and behaviour dependency edges, calculated from the former introduced graphs. This compound graph is the DG of selected functions and contains necessary information about data, control and behaviour dependencies of the expressions of the functions [14].

## 4. Change Impact Analysis

In this paper, we focus on impact analysis of refactorings. To find the functions that are affected by a refactoring, we use dependency graph based program slicing [16, 8].

The defined Dependency Graph represents the Erlang expressions as nodes and the dependencies among expressions as edges. A change in an expression in the source code that is changed by a refactoring may affect those expressions that depend on the changed expression. In order to track these changes, we have to traverse the Dependency Graph and gather those expressions that are reachable from the changed expressions. Traversing the Dependency Graph produces a static forward slice of the program. The program slice will contain all expressions that are affected by the refactoring. A function is affected by a refactoring if at least one expression from its body is contained in the slice. We collect all functions that are affected by a refactoring.

Since QuickCheck properties are formalised as Erlang functions, the program slice will contain the affected properties by a refactoring and we can suggest to the user to retest them. In case there is no affected property for an affected function we can check the behaviour equivalency within our refactoring tool.

## 5. Transforming and deriving QuickCheck properties

As mentioned already, change impact analysis exactly identifies program segments that may be affected by an arbitrary change (such as an effect of a refactoring step) in the code. The result of such an analysis process may include code of the application itself as well as parts of test modules. Due to the nature of impact analysis, the former ones basically cover code pieces whose meaning might have been modified by the previously mentioned change, while the latter sort of code fragments include test routines that may have to be changed accordingly and also they are due to be re-checked.

Those QuickCheck [10] properties that are available at the time of the execution of a refactoring step are transformed similarly, according to the impact of the specific refactoring step. More accurately, when a refactoring step altering the public interface of a function referred within a test module gets performed, calls within the test module are also transformed. For instance, when the “reorder function arguments” transformation changes the order of the arguments in a function definition, applications of that function are transformed accordingly, even those that are located within test modules.

Refactoring steps should preserve the semantics and the behaviour of programs. A properly performed refactoring step consequently results in a program that is semantically equivalent to the original one. In this paper, programs are said to be equivalent if they have the same observable behaviour, that is, they return the same values on given input data and have the same side-effects (such as I/O and exceptions). The validation of refactoring steps obviously includes a phase that checks whether user-defined test cases satisfied before the transformation remain satisfied. However, we can involve some additional techniques for checking behaviour preservation, namely, the most vital property to be checked may be that the original and the refactored program are equivalent.

To establish the fact of equivalence, we should execute each function of both the original and the transformed programs on every possible input value and check whether they all produce the same values and side-effects respectively. Obviously, this method is inapplicable in case of complex programs involving many functions operating on a large domains. Fortunately, we can omit the check of all the function that certainly are not affected by the performed code change. As seen before, impact analysis identifies the functions that might have been changed, so thus we only have to check those.

The problem of large input domains is resolved by using random generation. That is, change-affected functions are checked for semantics preservation by a large number of randomly generated values. As Erlang is dynamically typed, we can supply values of any type as arguments to functions, and at the worst case we get runtime exceptions. Arguably, randomly generating lots of improperly typed tests do not help with catching bugs. To avoid test cases failing for reasons of data being ill-typed, types of functions are inferred and then only well-typed input parameters are applied.

The property evaluates both the old and the new versions of affected functions, compares the results and analyses I/O activity and thrown exceptions. In the case they produce the same results, they likely are equivalent and therefore the refactoring step has been performed correctly, as it has preserved the program behaviour. Such a property can be defined within QuickCheck and can be checked just after a refactoring transformation has been made. User-defined properties along with the described equivalence property can efficiently validate refactoring steps.

**Further validation steps.** Beside checking whether the original and the refactored program are semantically equivalent, we might verify refactoring-specific properties as well, ensuring a more accurate validation of the refactoring steps. Such properties can be based on software metrics, calculated and compared on both the old and the new version of the source code.

## 6. Related work

There is a large body of work of software restructuring and refactoring. Accordingly, there exists several state-of-the-art research for making program refactoring safer. In most cases, these works focus on object-oriented programs (for example [2, 6, 11]), but there exists a technique for testing Erlang specific refactoring steps as well [3].

However, in the above researches, randomly generated test data is used and no complex (if any) additional semantic or syntactic information about the program is applied. The method illustrated in this paper uses knowledge based on impact analysis, which makes the generation of more specific and more relevant test cases possible, which may result in a more efficient testing method.

In the case of object-oriented programs there exists also incremental software change supporting tool using impact analysis [1]. Integrating the interactive behaviour of the refered tool and the test case selection method presented in this paper could be a possible direction for future work.

## 7. Conclusions and future plans

The paper demonstrates how knowledge originating from impact analysis can be used for validation of refactoring steps. The applicability of introduced method is illustrated by a case study based on capabilities of RefactorErl, a refactoring tool for functional programming language Erlang. The paper also outlines a method for assembling the required knowledge in the specific case of Erlang.

Since Erlang is a dynamically typed language, to ensure the safety of statically performed transformations is hard and the derived static semantic knowledge is not enough. Therefore we examined various connections among refactorings and test cases. Therefore we recommend to rerun a subset of selected test cases after the refactorings and we introduce new properties to check the behaviour of the changed system. We use impact analysis to select the code parts that are affected by a transformation and based on this knowledge we can minimise the cost of regression test.

In the future, the described method can be extended to make available refactoring steps driven improvement of existing test databases with semi-automatic extension of existing test cases relevant for program slices changed via given refactoring steps.

## Acknowledgement

This work was supported by TECH_08_A2-SZOMIN08, ELTE IKKK and Ericsson Hungary.

## References

[1] Buckner, J., Buchta, J., Petrenko, M., Rajlich, V. JRipples: A Tool for Incremental Software Change IEEE International Workshop on Program Comprehension, 2005, 149 - 152.

[2] Daniel, B., et al., Automated testing of refactoring engines. In ESEC/FSE, pages 185194, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-811-4.

[3] Drienyovszky, D., Horpácsi, D., Thompson, S., QuickChecking Refactoring Tools, Erlang 10: Proceedings of the 2010 ACM SIGPLAN Erlang Workshop, ed.: Scott Lystig Fritchie and Konstantinos Sagonas, pages 75-80 ACM SIGPLAN, September 2010,

[4] Erlang Homepage, *https://www.erlang.org*

[5] Fowler, M. and Beck, K. and Brant, J. and Opdyke, W. and Roberts, D., Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999

[6] Gligoric, M., et al., Test generation through programming in UDITA, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, pages 225-234, New York, NY, USA, 2010, ACM Press

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

[8] Horwitz, S. and Reps, T. and Binkley, D., Interprocedural slicing using dependence graphs., PhD thesis, University of Michigan, Ann Arbor, MI, 1979

[9] Lövei, L.:, Automated module interface upgrade, Erlang '09: Proceedings of the 8th ACM SIGPLAN workshop on Erlang, ISBN 978-1-60558-507-9, pages 11–22, Edinburgh, Scotland, September, 2009

[10] Quviq QuickCheck, http://www.quviq.com/, 2011

[11] Soares, G. Making Program Refactoring Safer. In ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pages 521-522, New York, NY, USA, 2010.

[12] Stafford, J.A., A Formal, Language-Independent, and Compositional Approach to Control Dependence Analysis., PhD thesis, University of Colorado, Boulder, Colorado, USA, 2000

[13] Tóth, M., Bozó, I., Horváth, Z., Tejfel, M.:, $1^{st}$ order flow analysis for Erlang, In Proceedings of 8th Joint Conference on Mathematics and Computer Science, 2010

[14] Tóth, M., Bozó, I.:, Building dependency graph for slicing Erlang programs, Paper submitted to Periodica Politechnica, 2010

[15] Tóth, M., Bozó, I., Horváth, Z., Lövei, L., Tejfel, M. Kozsik, T., Impact analysis of Erlang programs using behaviour dependency graphs., Central European Functional Programming School. Third Summer School. Revised Selected Lectures., 2010

[16] Weiser, M.: Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method., ACM Transactions on Programming Languages and Systems, 12(1):3546, 1990

Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University

*E-mail address*: {bozo_i,toth_m,matej,daniel_h,kitlei,kojqaai,hz}@inf.elte.hu