STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIX**, Special Issue 1, 2014
10th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, May 21-25, 2014

REDUCTION OF REGRESSION TESTS FOR ERLANG BASED ON IMPACT ANALYSIS

ISTVÁN BOZÓ, MELINDA TÓTH, AND ZOLTÁN HORVÁTH

Abstract. Legacy codes are changed in software maintenance processes to introduce new functionality, modify existing features, eliminate bugs etc. or by refactorings while the main original properties and the behaviour of the system should be preserved. Developers apply regression testing with highest degree of code coverage to be sure about it, and thus they retest the software after some modifications. Our research focuses on impact analysis of changes in applications written in the dynamically typed functional programming language, Erlang. To calculate the affected program parts, we use dependence graph based program slicing, therefore we have defined the Dependence Graphs with respect to the semantics of Erlang. Applying the results, we may shrink the set of test cases selected for regression testing for ones which are affected by the changes.

1. Introduction

Impact analysis is a mechanism to find those source code parts that are affected by a change on the source code, therefore it could help in test case selection for regression testing.

Erlang [9] is a dynamically typed functional programming language that was designed for building concurrent, reliable, robust, fault tolerant, distributed systems with soft real-time characteristic like telecommunication applications. The language has become widespread in industrial applications in the last decade.

Our research focuses on selecting the test cases of the Erlang applications that are affected by a change on the source code. In other words, we want to calculate the impact of a source code modification. To calculate the affected program parts, we use dependence graph based program slicing [24, 14], therefore we have to define the Dependence Graphs for Erlang.

Received by the editors: May 1, 2014.

2010 *Mathematics Subject Classification.* 68N18, 68Q99.

1998 *CR Categories and Descriptors.* F.3.2 [**Theory of Computation**]: Logics and Meaning of Program – *Semantics of Programming Languages: Program Analysis*.

Key words and phrases. static analysis, impact analysis, Erlang, test case selection, slicing.

Refactoring [10] is the process of changing and improving the quality of the source code without altering its external behaviour. Refactoring can be done manually or using a refactoring tool. We have been developing a refactoring tool for Erlang, called RefactorErl [6]. RefactorErl is a source code analysis and transformation tool [3]. It provides 24 refactoring steps for Erlang developers, such as moving, renaming different language entities, altering the interface of functions or the structure of expressions, parallelisation, etc. Besides transformations, RefactorErl has different features to support code comprehension [22].

RefactorErl checks the correctness of transformations using complex static source code analysis and accurate transformations. On the other hand, due to the semantics of dynamic languages such as Erlang, the accuracy of checking the side-conditions by static analysis is limited, which means a regression test is needed even for refactorings.

Erlang applications are often tested with the property based testing tool QuickCheck [2]: the tool checks some properties given by the developers with random generated test inputs. Therefore we want to select those QuickCheck properties that should be retested after the source code is modified.

The rest of this paper is structured as follows: Section 2 presents our motivation through an example; Section 3 introduces the used intermediate source code representations and Section 4 describes how we build the Dependence Graph based on the Control-Flow and Data-Flow Graphs; Section 5 describes the used program slicing technique for test case selection; Section 6 describes an extension of the presented techniques; Section 7 compares the results with the use of dynamic analysis; Section 8 presents related work; and finally, Section 9 concludes the paper and contains some future work.

2. Motivating Example

In this section we demonstrate a small example showing how we can select affected test cases after a modification of the source code. For the sake of simplicity we use an easy to understand transformation, a refactoring step in this example.

The following module (`test`) contains the function `add_mul/2` that adds and multiplies two numbers and returns the results in a tuple. We introduce two QuickCheck properties to test the function: the property `prop_add/0` tests whether the first element of the return value of `add_mul/2` is the sum of the two parameters, and the property `prop_mul/0` tests whether the second element of the return value is the product of the two parameters. The module `test` also introduces the function `pow` to raise `X` to the power `Y` and a property to check the power function: $I^J = I^{J-1} * I$

```
-module(test).
-export([add_mul/2, pow/2]).
-export([prop_add/0, prop_mul/0, prop_pow/]).

add_mul(X, Y) ->
    Add = X + Y,
    Mul = X * Y,
    {Add, Mul}.

prop_add() ->
    ?FORALL({I, J}, {int(), int()},
            element(1, sth(I, J)) == I + J).

prop_mul() ->
    ?FORALL({I, J}, {int(), int()},
            element(2, sth(I, J)) == I * J).

pow(X, Y) ->
    math:power(X, Y).

prop_pow() ->
    ?FORALL({I, J}, {int(), int()},
            pow(I, J) == pow(I, J-1) * J).
```

We can transform this module by the *Introduce function* refactoring [4]. This refactoring takes an expression or a sequence of expressions as an argument and creates a new function definition from it, then replaces the selected expressions with a function application that calls the newly created function. We can perform this transformation by selecting the `X + Y` expression:

```
add_mul(X, Y) ->
    Add = add(X, Y),
    Mul = X * Y,
    {Add, Mul}.

add(X, Y) ->
    X + Y.
```

Our goal is to select those test cases that are affected by the change made by the *Introduce function* refactoring. It is obvious that the property `prop_pow` is not affected, and neither is the property `prop_mul`. The refactoring changed only the value of the variable `X` that is the first element of the resulted tuple. Since `prop_mul` uses only the second element of the result of the function, we can deduce that this property is not affected by the change, so we should recheck only the property `prop_add`.

It is hard to calculate this manually for a complex source code modification on a large legacy code consisting of millions of lines. We build a Dependence Graph containing the data and control dependencies among expressions. Then we perform static program slicing [14] on the Dependence Graph to determine the affected code parts after a change on the source code, and finally based on the program slice, we calculate the properties to recheck.
Note. One can say that calculating the affected test cases for refactorings is not relevant. Let we give a counterexample. The goal of the ParaPhrase-Enlarged project [1] is to detect those parts of the source code where parallelism can be introduced by refactoring. The prior aim of the project is to work with meaning preserving transformations. It is necessary to carefully define the side-conditions of these transformations. However these side-conditions have to be very strict to ensure the meaning preserving transformations, therefore a huge number of transformations is denied. Using semi-automatic interactive transformations and weakening these side-conditions, we can extend the applicability of parallelisation. For example, the user can decide whether he wants to keep the order of “good side-effects” or not (a good side-effect can be an `ets` table reading). Although the user applied the transformation, it is recommended to retest the application after the parallelisation.

3. Intermediate Program Representation

Static program slicing is a technique to calculate the impact of a change on the source code. In order to calculate the program slices, different levels of knowledge should be available about the source code: we have to calculate the data and control dependence/relations among the expressions and we need static syntactic and semantic information for that. We build different abstract program representations for efficient calculation of the dependencies. In this section we briefly introduce the used intermediate representations, such as the Semantic Program Graph, Data-Flow and Control-Flow Graph [19].

3.1. **Semantic Program Graph.** The RefactorErl system introduces a Semantic Program Graph (later SPG) [13] to represent syntactic and static semantic information about the source code. The SPG is a rooted, directed, labelled graph that consists of three layers. The first layer includes the lexical layer, the middle layer is the Abstract Syntax Tree (later AST) of the program, and the third layer extends the AST to a SPG by adding different semantic information, like variable binding structure, function call information etc. Because of the graph representation and the semantic layer, it is more efficient to gather information about the source code than traversing the AST.

3.2. **Data-Flow Graph.** Based on the information available in the SPG, we can build a Data-Flow Graph (DFG). The $DFG = (N, E)$ is a directed, labelled graph containing the expressions of the Erlang programs as nodes (N) and the direct data-flow relations between them as edges (E). We have introduced six types of data-flow edges ($n_i \in N$):

- $n_1 \xrightarrow{flow} n_2$ – represents that the value of the node n_2 can be a copy of the value of n_1.
- $n_1 \xrightarrow{call} n_2$, $n_3 \xrightarrow{ret} n_4$ – the former one represents data-flow between the formal parameters of the functions and the actual parameters of the function calls. The latter one represents the data-flow between the return value of the function and the function applications. These edges represent that the values of the nodes are the same as in the $\xrightarrow{flow}$ edge.
- $n_1 \xrightarrow{sel} n_2$, $n_3 \xrightarrow{cons} n_4$ – these edges represent the data-flow among a compound data type and its elements. The former one represents that we select an element of an expression, and the latter one that we create a compound expression from elements.
- $n_1 \xrightarrow{dep} n_2$ – represents direct dependencies among expressions: the value of n_2 depends on the value of n_1.

We build an interfunctional DFG based on syntax driven formal rules and we have defined a relation on the DFG to express the indirect data-flow among the expressions of the Erlang programs called *First order data-flow reaching* [23]: $n_1 \overset{1f}{\rightsquigarrow} n_2$ means that the value of n_1 can flow into n_2, so the two values are the same.

3.3. **Control-Flow Graph.** We have defined compositional rules [21] for building the Control-Flow Graph (CFG) of Erlang functions according to the semantics of the language. The CFG is built by traversing the AST, following the semantic rules of the language.

The $CFG = (N, E)$ is a directed, labelled graph containing the expressions of the Erlang programs as nodes (N) and the direct data-flow relations between them as edges (E). We have introduced six types of control-flow edges ($n_i \in N$):

- $n_1 \longrightarrow n_2$ – represents that before evaluating n_2 we have to evaluate n_1
- $n_1 \xrightarrow{yes} n_2, n_3 \xrightarrow{no} n_4$ – represent conditional evaluation in the event of conditional branching and pattern matching

- $n_1 \xrightarrow{funcall} n_2$ – denotes that we have a function call. We build intrafunctional CFG-s for each function, and we resolve the function calls when creating a compound control dependence graph (See in Section 4.1).
- $n_1 \xrightarrow{ret} n_2$ – represents a return to a previously partially evaluated expression
- $n_1 \xrightarrow{send} n_2$ – represents that before evaluating n_2 we send the message that is the value of n_1
- $n_1 \xrightarrow{rec} n_2$ – represents that before evaluating n_2 we have to receive an expression

4. CALCULATING DEPENDENCIES

We need both the data-flow and the control-flow graph to calculate the real dependencies among expressions. However, it is not so efficient to use them for program slicing because every dependence edge calculation could require several graph traversals. Therefore we build a Control Dependence Graph from the CFG and then we add the data dependencies calculated from the DFG to that graph. The resulted graph is called Dependence Graph and contains the direct data and control dependencies among expressions. We can determine indirect dependencies by traversing this graph.

4.1. **Control Dependence Graph.**

The Control-Flow Graph contains every execution path of a certain function, and it also contains the sequencing among the evaluated expressions. However, when we want to calculate the impact of some change on the source code, it is not necessarily true that the evaluation of an element in a sequence has effect on the next elements of the sequence. Therefore we have to eliminate the unnecessary sequencing from the CFG and only the real control dependencies are taken into account.

To build the CDG, we have to build the Post-Dominator Tree [15] of the function (PDT). We say that a node n_2 from the CFG post-dominates the node n_1 if every execution path from n_1 to the *exit point* of the function contains n_2. Using the PDT and the CFG, we can calculate the CDG for a function. Since the CFGs are intrafunctional, the built CDGs do not contain the dependencies triggered by the function calls, message passing and message receiving. Such dependencies will be resolved while composing the intrafunctional CDGs into a composed CDG [17].

While building the CFGs, we examine the functions, whether a function may fail or not, and mark the expressions where the CDGs will be connected. This information is used while composing the CDGs to determine interfunctional dependencies.

The function may potentially fail at run-time if it has no exhaustive patterns, it contains an expression that may fail or throws an exception. The function application may affect the evaluation of the expressions following in the sequence, thus this dependency must be taken into account. The expressions following the function application node in the execution order will be directly dependent on the application node. These dependencies apply only for functions that may fail.

4.2. **Dependence Graph.** In the composed CDG, the edges of the graph denote control dependencies among the statements and expressions of the involved functions. This information in itself is insufficient for performing impact analysis. To reveal real dependencies among the statements of the program, data-flow and data dependency information is also required. The data dependency is calculated from the data-flow graphs of Erlang programs. We define data dependence between two nodes $n_1 \overset{ddep}{\rightsquigarrow} n_2$ if:

- there is a direct dependency edge between them – $n_1 \xrightarrow{dep} n_2$
- n_2 is reachable from n_1, so the value of n_1 can flow to n_2 – $n_1 \overset{1f}{\rightsquigarrow} n_2$

The data dependence relation ($\overset{ddep}{\rightsquigarrow}$) is transitive:

$$\frac{n_1 \overset{ddep}{\rightsquigarrow} n_2,\ n_2 \overset{ddep}{\rightsquigarrow} n_3}{n_1 \overset{ddep}{\rightsquigarrow} n_3}$$

The composed CDG is extended with the additional data dependencies, thus we obtain the Dependence Graph (DG) and we can perform program slicing on the DG.

This graph can be extended with some useful information like behaviour dependencies [20], which provide information how the behaviour of the function or the entire program is affected if the data at some statement is changed. With these additional edges we make the DG more accurate.

4.3. **Example Graphs.** The following function implements the factorial function in Erlang. When the factorial function takes `0` as an argument, it returns `1`, otherwise if the value of the parameter is greater than zero, it returns with the product of `N` and the factorial of `N-1`.

```
fact(0) ->
    1;
fact(N) when N > 0 ->
    N * fact(N-1).
```

Figure 1 shows the Control-Flow Graph of the factorial function. The evaluation of the function branches on pattern matching (`0` and `N`) and also

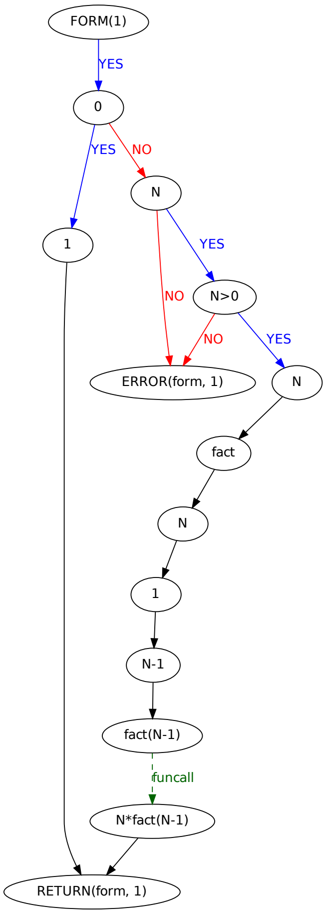

Figure 1. Control-Flow Graph of the factorial function

on the guard evaluation (`N>0`). The CFG contains a $\overset{funcall}{\longrightarrow}$ edge according to the function application `fact(N-1)`.

Figure 2 presents the Control Dependence Graph of the factorial function. The $\overset{dd}{\longrightarrow}$ edges represent direct control dependencies among expressions, the $\overset{inhdep}{\longrightarrow}$ edges represent the inherited control dependencies based on the function calls and the $\overset{resdep}{\longrightarrow}$ edges denote the resumption dependencies when the called function could fail.

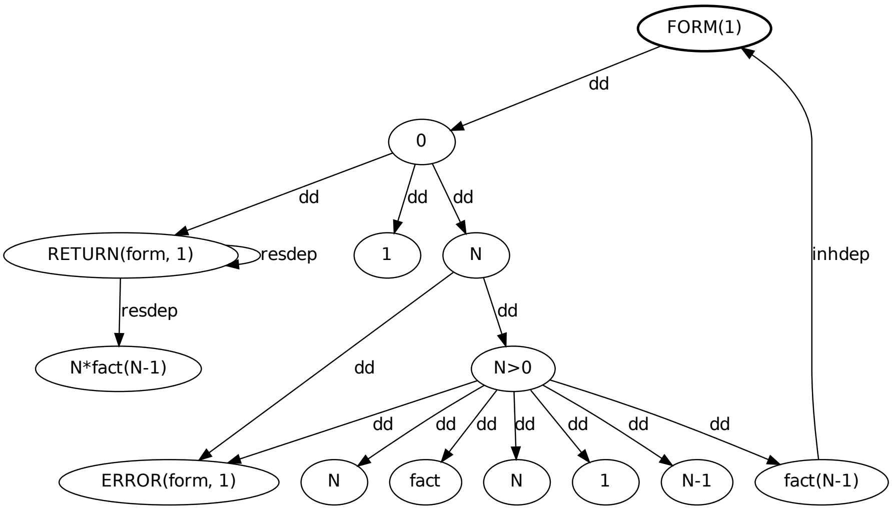

Figure 2. Control Dependence Graph of the factorial function

Figure 3 introduces the Dependence Graph containing both the control (black coloured edges) and the data dependencies (red coloured and dashed edges: $\xrightarrow{ddep}$). Calculating the affect of a change on the source code means to traverse this graph following the directed dependence edges without regarding its label. For instance, the expression `1` control depends on the expression `0` and the expression `fact(N-1)` data depends on the expression `1`, and therefore starting the slicing from the expression `0` results in a slice that contains expression `1`, expression `fact(N-1)`, etc.

5. Program Slicing for Test Case Selection

While some parts of the program are affected by a transformation of the source code, others are not. Let us consider the following simple example with three statements: `X = 2, Y = 3, Z = X + Y`. Replacing the integer `2` in the first match expression with another value does not affect the second match expression, but affects the third one, because of the data dependency among them (represented by the variable `X`). Therefore our task is to select a subset of expressions that depends on the value calculated at some point of interest, what is called static forward slice of the program.

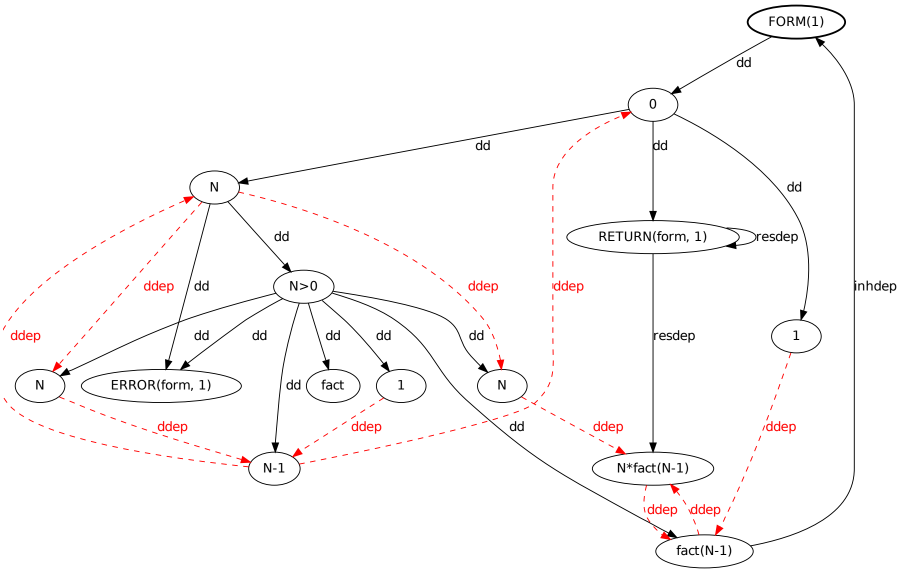

FIGURE 3. Dependence Graph of the factorial function

A forward program slice contains those expressions of the program that depend on the slicing criteria. The slicing criteria is an expression of the program. To calculate the program slice we have to build the Dependence Graph of the program and gather the expressions dependent on the slicing criteria.

The dependencies (control, data, behaviour, etc) among the expressions of the observed application are stored in the calculated Dependency Graph (Section 4). If expression `B` depends on expression `A` then there is a directed edge in the DG started from node `A` to node `B`. Thus, to calculate the expressions that depend on the value of another expression means to traverse the DG in forward direction.

We note here that traversing the DG in backward direction results in the backward program slice of the program containing those expressions that potentially affect the slicing criteria.

From our point of view, the slicing criteria is the set of expressions changed by the performed refactorings. The slicing algorithm extended with some more steps (we assume that the Semantic Program Graph of the program is available, because RefactorErl performs the refactorings on the SPG of the programs):

```
skel() ->
    Ch_Exprs = get_changed_exprs(),
    Ch_Funs  = get_changed_funs(Ch_Exprs),
    Af_Funs  = tr_closure(Ch_Funs,
                          [{funcall, forward},
                           {funcall, backward}])
    DFG      = build_dfg(Af_Funs),
    CFG      = build_cfg(Af_Funs),
    CDG      = build_cdg(CFG),
    CompDG   = resolve_dep(build_CompDG(CDG)),
    DG       = add_data_dep(DFG, CompDG),
    Slice    = traverse(DG, forward),
    examine(Slice, [{test, qc}]).
```

FIGURE 4. Erlang skeleton for the slicing algorithm

- calculate the affected expressions
- determine the functions that contain the changed expressions
- calculate the functions that are potentially affected by the change on the source : perform a transitive closure calculation on the call graph in both directions (forward and backward) starting from the changed functions
- build the Data-Flow and Control-Flow Graphs for the potentially affected functions
- build the Control Dependence Graph
- create the compound DG and resolve the dependencies
- calculate data dependencies between the expressions of the compound DG based on the DFG
- traverse the DG in forward direction starting from the set of changed expressions to collect all of the nodes that are affected by them. The resulted slice is a non-executable slice of the program.
- analyse the resulted slice to select the test cases to be rechecked (see in Section 5.1)

5.1. **Selecting QuickCheck Properties.** Since the test cases of Erlang applications are mainly implemented in Erlang modules (for example in EUnit [7], CommonTest [7], TestServer [7], QuickCheck [2]) we have to add those test cases to the Semantic Program Graph of RefactorErl. The analysis calculates the Dependency Graph based on the contents of the SPG, and the resulted slice will contain the test cases affected by the change of the source code.

Further analysis could evaluate the resulted test case set. For instance, an empty set of the cases means that the application was not fully tested, and we can make suggestions for the type of further test cases.

```
examine(Slice, Props) ->
    TestType = get_prop(Props),
    Funs     = get_funs(Slice),
    case TestType of
        qc ->
            filter(fun is_qc_fun/1, Funs)
        ctest -> ...
        eunit -> ...
    end.
```

FIGURE 5. Erlang skeleton for the test case selection algorithm

Based on the resulted slice, we use the following method to select the affected properties to be rechecked after the transformation: every property that contains at least one expression from the resulted program slice must be retested. Therefore, we have to determine the functions containing the expressions from the program slice and then we have to check the body of the function whether it defines an Erlang QuickCheck property (*eqc property*). Since the programmers define the QuickCheck properties using the well-defined set of *eqc* macros that are substituted to *eqc** function calls, we can calculate the affected properties based on the call graph of the preprocessed programs.

Identifying non-QuickCheck test cases is also possible, only some background knowledge is required about the test suit. That can be a naming convention (*prop_*, test_*, *_test*) or the exact set of test cases (name of the test suits or modules containing the tests).

6. MODIFYING THE SOURCE CODE MANUALLY

The introduced method is described in terms of refactoring transformations as a case of applicability. The analysis can be easily adapted to consider other modifications as well. The transformations can be performed either manually or with external tools.

The main challenge is to locate the changes in the source code. It affects the `get_changes_exprs` part of the algorithm (shown on Figure 4). Currently the external changes are detected on the function level in RefactorErl. This results in loss of accuracy of the analysis.

To get around this the change detection of the tool could be improved in the future. Comparing the syntax trees or finding the differences in the source code and locating the changes in the graph could solve the problem.

6.1. Combined Modifications.

There are situations when calculating the set of test cases after each modification is not sensible. It is more realistic to first perform a sequence of transformations (either refactoring or manual modifications) and then run the testing.

Logging the modifications results in a set of changes. This set can be used as input for the analysis, thus extending the applicability.

The analysis is prepared for handling sets of starting points, but in some cases it needs to be improved.

This approach raises some new questions that will be answered in the future.

For example:

- How to detect the undone change?
- How to solve overlapping modifications?
- etc.

7. Using Dynamic Analysis

For well-defined input data the dynamic analysis can provide more accurate results. In Erlang the `cover` [8] tool comes with other dynamic analyser tools. It provides a coverage analysis for Erlang programs in different levels of granularity. With this tool the test cases can be verified whether the relevant code parts are tested.

Compared `cover` to our analysis, starting the coverage analysis results in a dynamic slice of the program. The slice contains lines of the source code that the effect of the change may reach (forward direction in call chain). Usually the test cases are at the caller side that is in backward direction of the call chain.

Our analysis determines the effect of changes in both directions. This means the test cases will be involved in the statically computed slices (as precisely as it is possible from statically available information).

Our analysis used with cover is a good complement in a testing process. First we determine the set of test cases with our analysis then check the coverage of test on changed code parts.

8. Related Work

Program slicing (introduced by Mark Weiser [24]) is a well-known technique in object-oriented area, and program slices are commonly used to measure the impact of a change on the source code. There are different kinds of slicing techniques [18]. The most popular among them is the dependency graph based program slicing [14]. These kinds of analysis are not really widespread in functional languages, but control-flow analysis techniques have been presented [16] for some functional languages.

In order to perform static analysis on the given set of source code an intermediate representation for the source code is needed. This representation should include the expressions, language constructs and the relations/dependencies among them. Such representations are widely used in compiler techniques and source code analysis, but mainly for imperative and object-oriented programming languages. This representation is the Program Dependence Graph (later PDG), which includes control dependence and data dependence information. As a first step in building the PDG a Control Flow Graph (later CFG) is necessary. By means of the CFG a Post-dominator tree and the Control Dependence Graph (later CDG) is built based on the well known techniques used at compilers [15]. Combining the CDG with data dependence information we obtain the PDG. Our main goal was to develop similar methods for the functional programming language, Erlang. It was not straightforward because of the special language elements and semantics of the Erlang language. The known techniques for imperative languages assume a distinguished main procedure that is in relation with the other procedures or functions of the program. In Erlang, there can be several functions that frame the interface of the module. Thus we select a function or a set of functions that are affected by the change of the performed refactoring, and start to build the dependence graph from these functions. In addition, the language was designed for developing parallel and distributed applications, thus a detailed analysis is required to build appropriate CFGs.

Reducing the number of test cases is also an interesting topic [11]. For instance, there is a paper ([5]) that describes a methodology for regression test case selection for object-oriented design using the Unified Modelling Language. This paper gives a mapping among design changes and gives a classification of test cases: reusable, retestable and obsolete. In another paper [12] the authors presented a method for data-flow based selection using intraprocedural slicing algorithms.

Our mechanism is built for the functional programming language, Erlang, but it could be applicable for other strict functional languages. The main task is to build a control and a data-flow graph. Both require deep knowledge about the syntax and semantics of the selected language.

9. Conclusions and Future Work

After some program transformations are made on the source code, regression testing should be performed. In this paper we have presented an impact analysis mechanism to select a subset of test cases that are affected by a change on the source code. Rerunning an accurately selected test subset could result in the same testing coverage as a full regression test, but it takes less time than the complete test.

In this paper we have briefly described the used mechanism for impact analysis: dependency graph based program slicing. We have described how to build Dependence Graph from Erlang programs, and the necessary intermediate source code representations (Control- and Data-Flow Graph) to calculate it.

In the future we are going to refine the analysis: adding more Erlang specific dependency edges to the Dependency Graph, reduce the size of the resulted slices with more static and maybe also with dynamic information. We are also planning to analyse methods that can approximate the resulted slice without building the Dependence Graph, and in this way make the test case selection faster.

References

[1] *ParaPhrase–Enlarged Project homepage*, 2013.

[2] *Quviq QuickCheck*, 2013.

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

[4] István Bozó and Melinda Tóth. 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.

[5] L. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on uml designs. In *18th IEEE International Conference on Software Maintenance (ICSM'02)*, 2002.

[6] ELTE-IK, PNYF. *Refactorerl Homepage and Tool Manual*, 2013.

[7] Ericsson AB. *Erlang Documentation*, 2013.

[8] Ericsson AB. *Erlang Documentation – Cover*, 2013.

[9] Ericsson AB. *Erlang Reference Manual*, 2013.

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

[11] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. In *ACM Transaction on Software Engineering and Methodology*, volume 10, pages 188–197, 1998.

[12] Rajiv Gupta, Mary Jean, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance*, pages 299–308. IEEE Computer Society Press, 1992.

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

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

[15] S.S. Munchnick. *Advanced Compiler Design and Implementation* . Morgan Kauffmann Publishers, 1997.

[16] O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, 1991.

[17] J.A. Stafford. A formal, language-independent, and compositional approach to control dependence analysis. In *PhD thesis, University of Colorado, Boulder, Colorado, USA*, 2000.

[18] Frank Tip. A survey of program slicing techniques. In *Journal of Programming Languages*, volume 3, pages 121–189, 1995.

[19] M. Tóth and I. Bozó. Static Analysis of Complex Software Systems Implemented in Erlang. In *Central European Functional Programming Summer School Fourth Summer School, CEFP 2011, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743*, 2012.

[20] M. Tóth, I. Bozó, Z. Horváth, L. Lövei, M. Tejfel, and T. Kozsik. Impact analysis of Erlang programs using behaviour dependency graphs. In *Central European Functional Programming School. Third Summer School, CEFP 2009. Revised Selected Lectures.*, 2010.

[21] Melinda Tóth and István Bozó. *Building dependency graph for slicing Erlang programs.* Paper accepted to Periodica Politechnica, 2010.

[22] Melinda Tóth, István Bozó, and Zoltn Horváth. Applying the query language to support program comprehension. In *Proceeding of International Scientific Conference on Computer Science and Engineering*, pages 52–59, Stara Lubovna, Slovakia, Sep 2010.

[23] Melinda Tóth, István Bozó, Zoltn Horváth, and Máte Tejfel. 1^{st} order flow analysis for erlang. In *Proceedings of 8th Joint Conference on Mathematics and Computer Science*, 2010.

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

EÖTVÖS LORÁND UNIVERSITY AND ELTE-SOFT LTD.
E-mail address: `{bozoistvan,tothmelinda,hz}@elte.hu`