Annales Univ. Sci. Budapest., Sect. Comp. 37 (2012) 355–368

IMPROVING QUALITY OF SOFTWARE ANALYSER AND TRANSFORMER TOOLS USING SPECIFICATION BASED TESTING

Máté Tejfel, Melinda Tóth, István Bozó,
Dániel Horpácsi and Zoltán Horváth
(Budapest, Hungary)

Communicated by László Kozma

(Received January 15, 2012; accepted February 10, 2012)

Abstract. Reengineering tools, such as code analysers and transformers can give useful assistance during the software development process. On the other hand, the quality of these tools is crucial: improper analysis might deceive the programmer, and imprecise transformations might mess up the source code.

Testing is the most commonly applied method for improving software quality. Nevertheless, in the case of analyser and transformer tools every sort of testing methods gets laborious and problematic due to the fact that both the inputs and the results of such tools are complete programs or parts of programs.

This paper introduces a specification based testing method for RefactorErl, an analyser and transformer tool for Erlang. RefactorErl, like other reengineering tools, has an abstract internal representation of the source code that models the syntactic and semantic structure of the analysed program. In the presented testing method the formal specification of the model is transformed into properties describing the consistency of the built internal representation. This enables a less difficult way of testing in a high abstraction level. In order to improve the efficiency of the method, the applied test database contains automatically generated programs as well as manually written cases.

The presented method can be easily adopted to other reengineering tools using high level internal representation.

Key words and phrases: Specification based testing, analyser and transformer tools.
2010 Mathematics Subject Classification: 68M15.
1998 CR Categories and Descriptors: D.2.5.
Supported by KMOP-1.1.2-08/1-2008-0002, European Regional Development Fund, ELTE-Soft Ltd. and Ericsson Hungary.

1. Introduction

During the development of either small-scale or industrial-scale software, reengineering tools enabling analysis and refactoring of the source code ease debugging, comprehension, and code reuse, which may reduce costs. However, the quality and the reliability of these tools are crucial: imprecise analysis may lead to incorrect program transformations altering the semantics along with the behaviour of the software, which is undesired and inadmissible.

Testing is the most commonly applied method for improving software quality. Nevertheless, in the case of analyser and transformer tools every sort of testing methods gets laborious and problematic due to the fact that both the inputs and the results of such tools are complete programs or parts of programs. The complexity of the test data apparently makes it difficult both to specify and to verify the properties of the program.

At the same time, reengineering tools usually have some kind of internal representation of the source code in order to model the syntactic and semantic structure of the analysed program. A well-designed representation is fully equivalent to the original code, precisely captures the derivable program properties, and assists code comprehension as well as code transformation.

This paper introduces a specification based testing method for RefactorErl [2], an analyser and transformer tool for Erlang. This testing method verifies static code analysis by monitoring the consistency of the built program model. The formal specification of the analysis is transformed into testing properties describing the consistency of the abstract program graph being used for internal representation in RefactorErl. This enables a less difficult way of testing in a high abstraction level, and can improve the reliability by revealing misconceptions between specification and implementation. In order to improve the efficiency of the method, the applied test database contains automatically generated programs as well as manually written cases.

The presented method can be easily adopted to other reengineering tools using high level internal representation.

The rest of the paper is structured as follows. First, we introduce RefactorErl, then Section 3 describes the internal graph representation of programs as used in RefactorErl. In Section 4 the applied testing method is presented, and after that, Section 5 describes our automatic program generation method. Finally, Section 6 concludes.

2. RefactorErl tool

This section introduces RefactorErl [8, 14], a refactoring and code comprehension tool written for Erlang [6], in Erlang. It supports code analysis as well as semi-automatic code refactoring, i.e. semantics-preserving syntactic transformations, and offers a semantic query language providing semantic information about the code for developers. The latter gives a useful and comfortable way of program comprehension. The tool also provides an interface for module and function clustering, and defines a set of software complexity metrics and a query language to query the result of them.

RefactorErl introduces a Semantic Program Graph representation for Erlang. It performs lexical, syntactic and semantic analysis on the source code, and stores the result of the analysis in the Semantic Program Graph. Based on an extensible Erlang language description we generate a scanner and a parser, and the tool is fully layout and comment preserving. RefactorErl provides an asynchronous parallel semantic analyser framework to define different kinds of static analysis in an incremental, modular way. The tool provides a framework for syntax-based transformations and after a syntax driven change on the graph the semantic analyser framework automatically restores the semantic layer.

The tool has several editor plugins (Emacs, Eclipse, Vi), different console interfaces and a web-based interface to support multi-user and remote usage.

3. Intermediate program representation in RefactorErl

RefactorErl stores and manipulates the source code in a three layered graph, called the Semantic Program Graph (SPG). An SPG is a rooted, directed, and labelled graph containing the lexical, syntactic and semantic units of an Erlang program. Also, SPG captures the possibly cross-layer relations among those program units.

The lexical layer contains both the original and the preprocessed tokens of the programs, while the syntactic layer stores the abstract syntax tree of the preprocessed source code. The topmost layer is the semantic layer, which contains the results of the different kinds of semantic analyses, such as function call analysis, module reference analysis, and variable binding analysis. Based on the former steps, further static source code analysis can be performed, extending and making the semantic level of the graph more precise (i.e. side-effect analysis, data-flow analysis [18], dynamic function reference analysis [7], control-flow analysis [1]).

To give the idea of this Semantic Program Graph, let us show a simple example illustrating its basic structure.

```
-module(example).

prod([]) ->
    1;
prod([H|T]) ->
    Prod = prod(T),
    H * Prod.
```

This short module (`example`) defines only one function, `prod/1`, which calculates the product of the elements of a list. The semantic and syntactic layer of the Semantic Program Graph of this module is presented in Figure 1. The semantic level contains the data-flow edges too.

3.1. Data-Flow Graph

RefactorErl introduces a Data-Flow Graph (DFG) in order to capture the direct data-flow among Erlang expressions. The DFG is currently part of the SPG, but it might be considered as a separate graph. The $DFG = (N, E)$ represents the expressions of the Erlang programs as their corresponding syntax nodes (N) and the data-flow relations as labelled edges (E) among them. We have introduced the following edge types:

- $e_1 \overset{flow}{\rightarrow} e_2$ denotes that the value of e_2 can be a copy of the value of e_1;
- $e_1 \overset{dep}{\rightarrow} e_2$ represents that the value of e_2 depends on the value of e_1;
- $e_1 \overset{cons_i}{\rightarrow} e_2$ and $e_3 \overset{sel_i}{\rightarrow} e_4$ denotes that e_2 is compound data and it contains e_1 as its i-th element; similarly, we can select the i-th element of compound data e_3, which is e_4;
- $e_1 \overset{call_{g,i}}{\rightarrow} e_2$ and $e_3 \overset{ret_{g,i}}{\rightarrow} e_4$ represent inter-functional data-flow through function calls, we denote the copying of the actual parameters of a function call of g to the formal parameters with the edge $\overset{call_{g_i}}{\rightarrow}$, and we denote the copying of the return value with $\overset{ret_{g_i}}{\rightarrow}$.

Besides the direct flow edges we have introduced a data-flow relation to represent the indirect data-flow among expressions, called first order data-flow *reaching* [18].

We build the DFG based on formal rules describing the direct data-flow relations among expressions [13, 18]. The left hand side of each rule describes

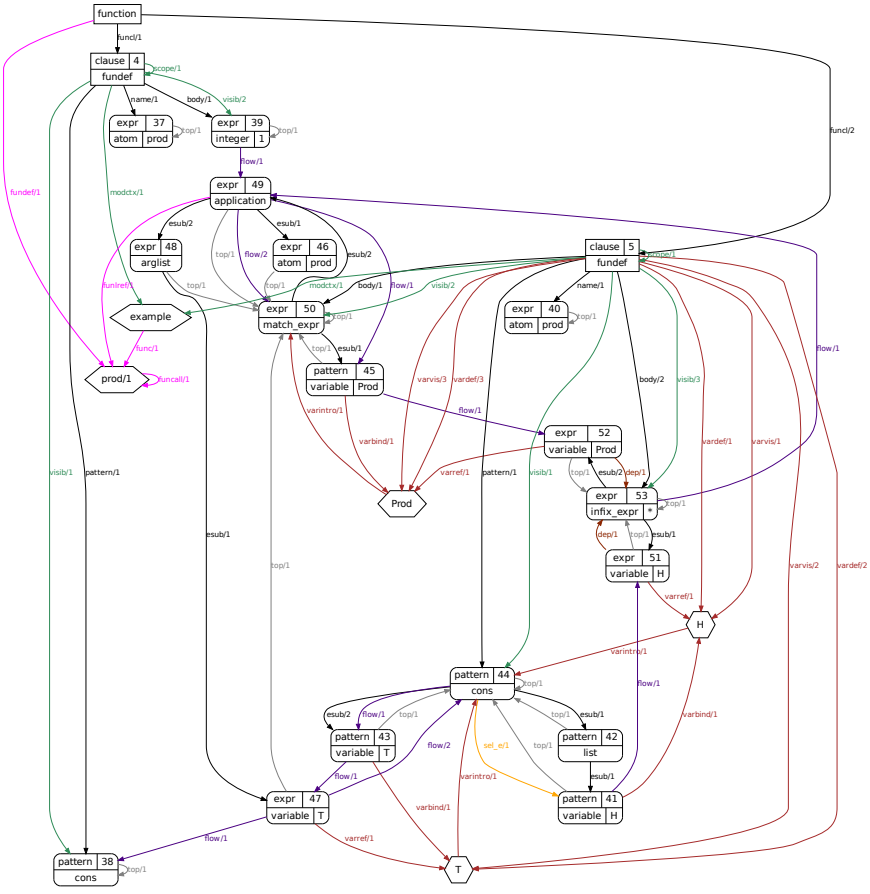

Figure 1. Semantic Program Graph of prod/1 — semantic and syntactic level

the syntax of a language element, and the right hand side of the rule describes the data-flow among the introduced elements. In this paper we use the following notation: e denotes an Erlang expression, p denotes a pattern and g denotes a guard expression of the language.

The rule presented in Figure 2 illustrates the data-flow in the case of a match expression. According to the semantics of Erlang, when we match a value to a variable, the value of the variable (and also the value of the match expression) is the same as the right hand side of the match expression. Therefore, the $\overset{flow}{\rightarrow}$ edges from the right hand side expression e represent that the value of e flows to p and e_0.

Expression	Graph edges
e_0: $p = e$	$e \overset{flow}{\rightarrow} e_0$ $e \overset{flow}{\rightarrow} p$

Figure 2. The Data-Flow Rule of the Match Expression

The Infix expression Rule (Figure 3) describes that the value of an infix expression depends on the values of its operands.

Expression	Graph edges
e_0: $e_1 \circ e_2$	$e_1 \overset{dep}{\rightarrow} e_0$ $e_2 \overset{dep}{\rightarrow} e_0$

Figure 3. The Data-Flow Rule of the Infix Expression

When building the DFG, we apply the corresponding rule on each expression from the SPG.

For instance, when the program contains the `A = 1 + 2` expression, we apply the match expression rule first ($e_+ \overset{flow}{\rightarrow} p_A$, $e_+ \overset{flow}{\rightarrow} e_=$) and then the infix rule ($e_1 \overset{dep}{\rightarrow} e_+$, $e_2 \overset{dep}{\rightarrow} e_+$) – where e_+ denotes the infix expression `1+2`, e_1 denotes the integer `1`, p_A denotes the pattern `A`, etc.

3.2. Control-Flow Graph

The Control-Flow Graph (CFG) contains every execution path of the program for every possible input. The control-flow analysis is language dependent as it is based on the semantics of the language. The Erlang programming language has strict evaluation, thus before evaluating a compound expression, the subexpressions must be evaluated first.

For building the CFG for a function we use knowledge from the SPG. Unlike the DFG, the CFG is built as a completely separate graph; however, we use the same vertex identifiers as SPG (extended with dummy vertices for return nodes, joining branches and error nodes) in order to ease mapping between the SPG and the CFG.

Like the DFG, the CFG is a set of nodes and the control-flow is represented by a set of edges among the expressions. There are different control-flow edge types:

- $\rightarrow$: represents sequencing;
- $\overset{yes}{\rightarrow}, \overset{no}{\rightarrow}$: branching expressions, like case expression;

- $\overset{ret}{\rightarrow}$: returning form a function, case expression, etc.;
- $\overset{send}{\rightarrow}$, $\overset{rec}{\rightarrow}$: label for message sending and receiving expressions;
- $\overset{call}{\rightarrow}$: function call expression.

We build the CFG of Erlang functions based on formal rules. The rules are defined corresponding to the semantics of the language [1]. Figure 4 shows the control-flow building rules of `case expressions`. The formal description of the case expression is in the `Expression` column and the corresponding CFG edges are in the `Graph edges` column.

In the case of the `case expression`, first the head expression (`e`) is evaluated, then the return value of the evaluated expression is matched against the patterns. If the pattern matches, next the guard expression is evaluated ($p_i \overset{yes}{\rightarrow} g_i$). If the guard evaluates to `true`, this clause will be chosen for evaluation ($g_i \overset{yes}{\rightarrow} e_i^1$). If either the pattern matching fails or the guard expression evaluates to `false`, the control flows to the next pattern ($p_i \overset{no}{\rightarrow} p_{i+1}, g_i \overset{no}{\rightarrow} p_{i+1}$). The return value of the case expression is the value of the last expression of the evaluated branch ($e_i^j \rightarrow ret\ case$). If neither of the patterns match, an exception is thrown, thus the control flows to an error node ($p_n \rightarrow error$).

4. Specification-based testing

High quality analyser and refactoring tools can be very useful during the development process of any program. The most commonly applied method for improving software quality is testing. We can use testing methods also in the case of reengineering tools; however, the application can be much more complex because of the need of complete programs or parts of programs as test data.

In the case of formal specification-based testing instead of the creation of many different test cases we have to formalise the required properties of the program and describe the generation method capable to create input data automatically [9]. Since we can apply the same property for many test cases, this can significantly reduce the cost of testing. Additional advantage is that this method covers unusual cases more likely than the hand-written tests which facilitate to detect hidden errors.

This approach is especially beneficial when – as in the case of RefactorErl – there exists some abstract specification as a starting point for the formalization of required properties. However, for automatic testing we have to describe the formalisation in a machine-processable way. The fact that there is a little gap

Expression	Graph edges
e_0: case e of p_1 when $g_1 \to e_1^1, \ldots, e_{l_1}^1$; $\vdots$ p_n when $g_n \to e_1^n, \ldots, e_{l_n}^n$ end	$e_0' \xrightarrow{e}, \ e \to p_1,$ $p_1 \xrightarrow{yes} g_1, \ p_1 \xrightarrow{no} p_2,$ $\vdots$ $p_{n-1} \xrightarrow{yes} g_{n-1}, \ p_{n-1} \xrightarrow{no} p_n,$ $p_n \xrightarrow{yes} g_n, \ p_n \xrightarrow{no} error,$ $g_1 \xrightarrow{yes} e_1^1, \ g_1 \xrightarrow{no} p_2,$ $\vdots$ $g_{n-1} \xrightarrow{yes} e_1^{n-1}, \ g_{n-1} \xrightarrow{no} p_n,$ $g_n \xrightarrow{yes} e_1^n, \ g_n \xrightarrow{no} error,$ $e_1^1 \to e_2^1, \ \ldots, \ e_{l_1-1}^1 \to e_{l_1}^1,$ $\vdots$ $e_1^n \to e_2^n, \ \ldots, \ e_{l_n-1}^n \to e_{l_n}^n,$ $e_{l_1}^1 \to ret\ case,$ $\vdots$ $e_{l_n}^n \to ret\ case,$ $ret\ case \to e_0$

Figure 4. The Control-Flow Rule of the Case Expression

between the initial abstract specification and the machine-processable properties usually means that the abstract specification is extremely large since we have to describe the requirements for every different method. Otherwise, if there is a big gap between them, it is a laborious process to transform the initial properties to machine-processable ones.

In the case of RefactorErl – as mentioned before –, high level abstract specifications exist for describing the required properties of different analyser modules of the system. Fortunately, at the same time, the system – as in the usual case of reengineering tools – has a high level internal representation of the source code in order to model the syntactic and semantic structure of the analysed program. The analysers operate on this representation, which makes it possible to verify them in this relatively high level.

We have transformed the abstract required properties of RefactorErl to properties describing different consistency requirements for the representation graph. In this way, instead of verifying high level abstract properties, we can check the consistency of the internal graph resulted by the analysers.

As an example, let us consider the data-flow rule of match expression described in Section 3.1 and introduced by Figure 2.

We can describe this rule in the level of the internal graph representation in the following way. Considering the graph resulted by the data-flow analyser for every node which represents match expression, there exists a *flow* edge from the second child of the node (representing the second operand of the expression) to the investigated node and an other *flow* edge from the second child to the first child of the node (representing the first operand of the expression).

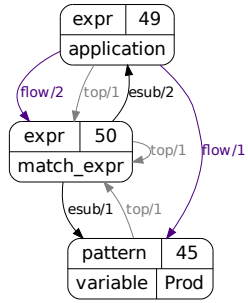

Figure 5. Part of Semantic Program Graph of prod/1

For illustration Figure 5 introduces the corresponding part of the representation graph in the case of `prod/1` function (described in Section 3), where `Node 50` represents a match expression. Since the operands of an expression are accessible along *esub* edges, we can interpret the transformed rule here as a *flow* edge that has to lead from `Node 49` (the second child of `Node 50`) to `Node 50` and an other to `Node 45` (the first child of `Node 50`).

Using the internal representation of RefactorErl, we can describe the investigated rule in Erlang in the following way. We have to collect from the graph every node illustrates match expression and check the described property for these nodes:

```
match_expr_test(Module) ->
   MatchExprNodes = getMatchExprNodes(Module),
   lists:map(fun props/1, MatchExprNodes).
```

For verifying the property for a given node we have to determine the two children of the node and check if the corresponding two *flow* edges exist:

```
props(MatchExpr) ->
    case checkTwoChild(MatchExpr) of
       true -> Ch1 = getFirstCh(MatchExpr),
               Ch2 = getSecondCh(MatchExpr),
               check_matchExpr(MatchExpr, Ch1, Ch2);
       false -> {error_in_matchexpr, MatchExpr}
    end.

check_matchExpr(MatchExpr, Ch1, Ch2) ->
    Ch2Flows = getFlows(Ch2),
    case lists:member(Ch1, Ch2Flows) and
           lists:member(MatchExpr, Ch2Flows) of
        true  -> ok;
        false -> {error_in_matchexpr, MatchExpr}
    end.
```

Here the `getFlows(Ch2)` function determines every node reachable from `Ch2` node along *flow* edge.

Since we can formalise the requirements in Erlang in a machine-processable way, we can use QuickCheck testing tool (see [9]) for verifying the properties of RefactorErl. However, for this automatic testing we need Erlang programs as test data. The next section gives an overview how we can randomly generate Erlang programs.

5. Program generation

As stated already, validation can be made more efficient by applying random inputs, which cover unusual cases more likely than the hand-written values. The aforementioned random Erlang programs, which are used as test data in the graph consistency check process, are produced by using QuickCheck generators. QuickCheck lets the tester define the way the universally quantified variables inside the verified properties get their value. Data generators can create random values of data types supported in the language. Simple types have their first-order generators, which can be composed into complex generators by higher-order generator combinators.

Programs may have different representations, so the first step is to decide which one to create a generator for. Apparently, generating source code as the concatenation of random program parts as strings is not feasible, at least if we would like the process to yield syntactically valid code. Since programs can be seen as their abstract syntax trees (AST) as well, we can generate syntax trees, and then we can print them into text. Therefore, the most essential ingredients we need for creating a generator for random Erlang programs are the type

definition of Erlang ASTs and a pretty printer for those ASTs. Obviously, this is quite a difficult task, and also, a type description is still not enough for producing random values. We have to bridge the gap between type specifications and data generators in advance.

Altough there are some efforts to create generators from type specifications [19], our solution to the above issue is the following: we define the set of possible Erlang ASTs not by a type definition but by a formal production grammar, which we transform into data generators. That is, we defined an attributed grammar which generates the formal language of Erlang ASTs, and the production rules of this formal grammar can be automatically turned into QuickCheck generators. The transformation is carried out by our special compiler. Apparently, an attributed grammar can be much more expressive than a simple type structure definition in Erlang. With the formal attributes, one can describe not only the structure of the data, but its semantics as well, and this fact implies that we can generate any desired subset of Erlang programs.

In order to efficiently test the code analysis and graph consistency of RefactorErl, we have given a grammar that defines programs having complex dataflow, a dozen of program entities and cross-references. This fact along with the randomness and proper distribution provided by QuickCheck guarantees that sooner or later the produced programs will include the most interesting constructs with a complex structure, which likely leads to finding misconceptions or bugs in the analyser system.

6. Related work and conclusion

Considering related work there are plenty of different application areas where specification-based testing has been used effectively (for example see [3, 10, 11, 16]). However, in the case of reengineering tools the situation is significantly different because of the difficulties described in Section 4. The most closely related work is the approach of Huiqing Li and Simon Thompson [12], nevertheless they define low level properties for specific refactoring transformations instead of high level general properties and fixed test database instead of randomly generated Erlang programs.

This paper introduced a specification-based testing method for RefactorErl, a refactoring and code comprehension tool written for Erlang. High level abstract properties describing the required behaviour of different analyser modules of RefactorErl were transformed to the level of internal graph representation. Thus, it became possible to check the consistency of the representation instead of verifying high level properties. The paper also described a generation

method used for the creation of random Erlang programs as input data for the tests to make it more effective.

Applying the illustrated method, more than 20 different properties in the level of internal graph representation of RefactorErl were described, and based on these properties thousands of test cases were generated with the usage of QuickCheck testing tool and using the program generation method illustrated in Section 5.

Acknowledgement. The authors thank László Lövei and Róbert Kitlei for their ideas and work, who were members in the RefactorErl project.

References

[1] **Bozó, I. and M. Tóth,** Building dependency graph for slicing Erlang programs, *Periodica Politechnica*, (2010), accepted.

[2] **Bozó, I., D. Horpácsi, Z. Horváth, R. Kitlei, J. Kőszegi, M. Tejfel and M. Tóth,** Refactorerl – source code analysis and refactoring in erlang, in: *Proceeding of the 12th Symposium on Programming Languages and Software Tools*, Tallin, Estonia, 2011.

[3] **Claessen, K. and J. Hughes,** QuickCheck: a lightweight tool for random testing of Haskell programs, in: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ACM, New York, NY, USA, 2000, pp. 268–279.

[4] **Daniel, B., et al.,** Automated testing of refactoring engines, in: *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, New York, NY, USA, 2007, pp. 185–194.

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

[6] **Erlang Homepage (2011),** `https://www.erlang.org`

[7] **Horpácsi, D. and J. Kőszegi,** Static analysis of function calls in erlang – refining the static function call graph with dynamic call information by using data-flow analysis, in: *Proceedings of the Central and Eastern European Conference on Software Engineering Techniques*, Debrecen, Hungary, August 2011.

[8] **Horváth, Z., L. Lövei, T. Kozsik, R. Kitlei, A. Víg, T. Nagy, M. Tóth and R. 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, volume 54(2009) Sp. Issue, Studia Universitatis Babes-Bolyai, Series Informatica*, Cluj-Napoca, Romania, July, 2009, pp. 7–16.

[9] **Hughes, J.,** Software testing with QuickCheck, in: *Proceedings of the Third summer school conference on Central European functional programming school, CEFP' 09*, Springer-Verlag, 2010, pp. 183–223.

[10] **Jürjens, J. and G. Wimmel,** Specification-Based Testing of Firewalls, in: *Revised Papers from the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics, Akademgorodok, Novosibirsk, Russia*, Springer-Verlag, 2001, pp. 308–316.

[11] **Khurshid, S. and D. Marinov,** TestEra: Specification-Based Testing of Java Programs Using SAT, *Automated Software Engineering*, Kluwer Academic Publishers, **11(4)** (2004), 403–434.

[12] **Li, H. and S. Thompson,** Testing Erlang Refactorings with QuickCheck, in: *Implementation and Application of Functional Languages*, (eds.: O. Chitil, Z. Horváth and V. Zsók), Springer-Verlag, 2008, pp. 19– 36.

[13] **Lövei, L.,** Automated module interface upgrade, in: *Erlang '09: Proceedings of the 8th ACM SIGPLAN workshop on Erlang*, Edinburgh, Scotland, September, 2009, pp. 11–22.

[14] **RefactorErl Home Page (2011),** `http://plc.inf.elte.hu/erlang/`

[15] **Soares, G.,** Making Program Refactoring Safer, in: *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, New York, NY, USA, 2010, pp. 521–522.

[16] **Tsai, W. and Y. Chen and R. Paul,** Specification-Based Verification and Validation of Web Services and Service-Oriented Operating Systems, in: *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, IEEE Computer Society, Washington, DC, USA, 2005, pp. 139–147.

[17] **Tóth, M., 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, Revised Selected Lectures*, Komárno, Slovakia, June, 2009, pp. 372–390.

[18] **Tóth, M., I. Bozó, Z. Horváth and M. Tejfel,** 1^{st} order flow analysis for Erlang, in: *Proceedings of the 8th Joint Conference on Mathematics and Computer Science*, Komárno, Slovakia, 2010, pp. 403–416.

[19] **Papadakis, M. and K. Sagonas,** A PropEr Integration of Types and Function Specifications with Property-Based Testing, in: *Proceedings of the 2011 ACM SIGPLAN Erlang Workshop*, Tokyo, Japan, September 2011, pp. 39–50.

M. Tejfel, M. Tóth, I. Bozó, D. Horpácsi and Z. Horváth
Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
H-1117 Budapest, Pázmány P. sétány 1/C
Hungary
matej@caesar.elte.hu
tothmelinda@caesar.elte.hu bozoistvan@caesar.elte.hu
daniel-h@caesar.elte.hu
hz@caesar.elte.hu