QuickChecking Refactoring Tools

Dániel Drienyovszky Dániel Horpácsi
University of Kent and Eötvös Loránd University
{dd210, dh254}@kent.ac.uk

Simon Thompson
University of Kent
S.J.Thompson@kent.ac.uk

Abstract

Refactoring is the transformation of program source code in a way that preserves the behaviour of the program. Many tools exist for automating a number of refactoring steps, but these tools are often poorly tested. We present an automated testing framework based on QuickCheck for testing refactoring tools written for the Erlang programming language.

Categories and Subject Descriptors D. Software [*D.2 SOFTWARE ENGINEERING*]: D.2.5 Testing and Debugging: Testing tools

General Terms Verification

Keywords refactoring, Wrangler, RefactorErl, Erlang, random program generation, QuickCheck, attribute grammar, yecc, property

1. Introduction

Refactoring [10, 15] is a process of rewriting program source code without changing its meaning whilst improving properties such as maintainability, clarity or performance. Refactorings range from simple ones, like renaming a variable, to more complex ones, like generalising a function. Refactoring transformations may affect large parts of the code base for a project and in particular may require modifications of more several different modules from a project. Applying such code-to-code transformations are common practice among software developers, consequently tool support for refactoring is widespread in mainstream programming languages. Refactoring tools/engines help to automate the routine aspects of certain code transformations. For Erlang there are three refactoring tools available: Wrangler [3] from the University of Kent, RefactorErl [2] from Eötvös Loránd University and last but not least, Tidier [6, 18], a completely automatic code cleaning tool from the National Technical University of Athens.

These tools are hard to test, as they require manually written test cases aiming to cover every corner case of the language being refactored. Evidently, by using only such case-based testing we never can provide a comprehensive check of the refactoring engines. To increase confidence in these tools, a more efficient testing approach has to be applied. We investigate automated testing of refactoring tools by generating random programs and verifying whether the refactorings preserve the meaning of these randomly-generated programs. We have chosen QuickCheck as our testing tool and two Erlang refactoring tools, Wrangler and RefactorErl as the tools to be tested.

In this paper we describe the difficulties of the testing of refactoring tools (Section 2) and we also present our contribution, which is to make the entire testing process fully automated. The method is composed of two separate parts.

- First, we have created a QuickCheck-based random generator for producing Erlang programs which are used as the input of the refactoring transformations. More precisely, we have formalised the programming language to be refactored (namely, Erlang) with a proper grammar class and then based on this description we have derived a corresponding QuickCheck generator for the syntactically and semantically correct programs of the language (see Section 3).
- Second, we have created a definition of equivalence between Erlang modules, which leans on the dynamic behaviour of the programs. Furthermore, we have formalised this equivalence relation by means of QuickCheck properties (see Section 4).

Using these two ideas together, we have built a fully automated testing framework for Erlang code-to-code transformation tools. The final sections of the paper comment on the results we have derived, on related work, and our conclusions, where we note that the tool we have built here is applicable to all the refactoring tools for Erlang, and that the approach we outline could equally well be applied to testing refactoring tools for other languages.

2. Validating refactoring tools

There are many ways of establishing program correctness, including formal proof mechanisms as well as several testing strategies. While developing software, evidently, we would like to be sure of our program's correctness. Since formal methods are mostly too difficult to apply, despite some preliminary work reported in [20], we focus on testing as a mechanism for validating and improving the quality of our programs.

2.1 Case-based testing

There are many testing approaches, which aim to check as many program parts as possible, as assessed in different ways. With testing, basically, we are not able to prove the program's correctness, but we can establish that in numerous cases the program does what we expect from it, and this can give us a degree of confidence that it indeed satisfies its requirements.

Commonly, programmers apply simple use-case based testing to check fundamental requirements. However, in the case of complex software it is impossible to cover all the most common and most interesting cases just by test cases written by hand. For instance, in our specific case, there always may be found unusual instances that are valid source code but that would seldom be writ-

Erlang'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0253-1/10/09…$10.00

ten by human programmers. The latter sort of code should also be handled correctly by a practical refactoring tool.

Since refactoring tools can mess up or even corrupt our code base by accident, they have to be well-tested. They can only be useful accessories of the development process if they can be expected to perform the transformation steps properly, without making any mistakes. We apply such tools instead of performing transformations by hand because the refactoring software should be much more precise than the human programmer can ever be. An efficient and comprehensive testing method has to be used in order to achieve a reasonable confidence in the reliability of the tool.

Testing of refactoring tools is difficult due to the complexity of their input and output: both the input and the output of such programs are program source code. Such code is a complex data type, since it embodies not only syntactic well-formedness but also semantic correctness too. Furthermore, defining the semantics of the refactorings, namely, how a transformation has to be performed on different input data, is not straightforward task either. By using case-based testing on the transformation steps we can cover the main features of the functionality with a reasonable labour, however, more sophisticated approaches are also applicable, such as property-based testing methods.

2.2 Property-based testing in QuickCheck

Property-based testing makes a generalisation of usual test cases by eliminating the concrete input from the test case and replacing it with randomly generated test data. So then, the test cases describe only the properties (the main points) of the specified case rather than describing a concrete input-output pair. The properties can be efficiently checked on large number of randomly generated test inputs.

Such testing methods may be regarded as a fusion of the formal proof methods and the traditional test case based testing. When using property based testing, we do not define specific input-output pairs that describe the requirements, but we specify in a logical property the expected behaviour on inputs satisfying the specified conditions. The expected behaviour is drawn in terms of specification properties. During the test, these properties are checked in a large number of test cases. Usually, the test input is randomly generated by the framework, based on special data generators. The data generators describe the structure and the essential properties of the input data to be used for testing. Also, the distribution of the random data can also be controlled through the generators.

QuickCheck is a well-designed implementation of the property-based testing method for functional programs, including the Erlang programming language. QuviQ QuickCheck [1, 4, 5], the commercial QuickCheck implementation for Erlang, is a tool for automatic testing of Erlang programs against a user-written specification. The testing method known as 'QuickChecking' means the checking of specification properties (that is the expected functionality) in a large number of randomly generated cases. QuickCheck properties are expressed in standard Erlang code, using the macros and functions defined in the QuickCheck library.

As we mentioned already, property-based testing involves two kinds of activity.

- The first is the description of the testing data used as input for the program being tested. *Data generators* describe the way that the test data is generated, as well as the expected probability distribution of the randomly generated data.
- The other is the *specification of the properties* expected of the program. These are typically universally-quantified properties, and the data produced by the generators are used as the actual values of the universal variables.

Property-based testing can provide comprehensive testing of several kind of software. In this paper we present property-based testing of Erlang refactoring tools, which involves a definition of a data generator for Erlang module source code as well as a property for determining whether two modules, namely a module before and after refactoring, are equivalent.

3. Random program generation

While creating data generators we can use built-in data generators and in addition, QuickCheck allows the programmers to define their own data generators to create more complex random values. Generators for the built-in types are defined by the framework, so we have to create generators only for our own types by combining the built-in generators by using generator combinators.

In this paper, the term 'first-order generator' means simple data generators that are not parameterised with any other generators. On the other hand, the term 'higher-order generator' strands for the so-called generator combinators, which are generators that may take one or more generators as their arguments. First-order generators only take simple Erlang terms as parameters. They are the core of the generation (since they do not combine other generators but indeed create data values). On the other hand, higher-order generators combine other data generators and may result in arbitrarily complex data generators.

Despite the fact that QuickCheck generators provide a powerful toolkit for defining test data, in the case of larger programs operating on complex input, writing generators by hand is tedious and results in complicated source code, containing substantial 'boiler plate', that is hard to maintain. With a more general notation, that is, with a metalanguage more powerful than the QuickCheck generators, we can reduce the complexity of the description and the amount of the wasted coding time. In this approach QuickCheck generators are a low-level formalism and our meta-notation is a high-level formalism that eases the description of the test data.

Attribute grammars

A formal grammar is a set of rules, which describes a formal language [8], for example, the syntax of a programming language. Usually programming language syntax is formalised with EBNF (Extended Backus-Naur Form) [7], which is a meta-syntax notation used to express context-free grammars (CFG). However, also to describe the static semantics of a language – such as the binding structure of variables and other identifiers – a more expressive formalism is needed.

Attribute grammars (AG) [12, 16] are generalisations of context-free grammars, where the grammar rules are extended with semantic computation rules to calculate associated values or *attributes*. With attribute grammars both the syntax and the semantics are representable together. The attributes are divided into two groups: synthesised attributes and inherited attributes. The former are computed from constants and attributes attached to the children, the latter depend only on constants and the parents or siblings.

Synthesised attributes are used to calculate and store results like the value of an expression, whereas inherited attributes are used to carry the context of a node, such as an environment of variable bindings in scope at that point. In some approaches, synthesised attributes are used to pass semantic information up the parse tree, while inherited attributes help pass semantic information down and across it.

There are important subclasses of attribute grammars, which have some restrictions on the form of the attribute computation rules [13]. S-Attributed grammars involve synthesised attributes only, L-Attributed grammars allow attribute inheritance with the restriction that dependencies from a child to the child itself or to the child's right are not allowed. Formally, given a rule $A \rightarrow X_1 X_2 \ldots X_n$ in the L-Attributed grammar, each inherited attribute of X_j $(1 \leq j \leq n)$ depends only on attributes of $X_1, X_2, \ldots, X_{j-1}$ and on inherited attributes of A. Furthermore, synthesised attributes of X_j may depend on its own inherited attributes. Synthesised attributes of A depend on inherited attributes of A and on any attributes of the right hand side symbols. This definition of L-attribution effectively means that there are no cycles in the attribute calculation, and that calculation can conclude in a single pass.

Grammars and testing

Test data may be defined by means of formal grammars. This kind of testing is usually called *grammar-based testing*[19]. In this concept, a test datum is a word of the language defining the domain of the tested function and this language can be given by means of formal grammars. We have created a grammar-based meta-notation for QuickCheck data generators and in our experience, data described in our notation usually is about 5 times more compact than writing the same data with standard QuickCheck generators. For example, in the case of a simple language ($a^n b^n c^n$), compiling some 10 lines of description results in about 55 lines of Erlang code containing the QuickCheck generators.

In QuickCheck for Erlang there is already a module with similar capabilities: `eqc_grammar` [1] is a library module of the QuviQ QuickCheck distribution. This tool is able to create QuickCheck generators from a yecc-like[1] grammar description, but in contrast to our work, it does not support attributes, EBNF notations and many other features that are covered in detail in Dániel Horpácsi's master's thesis [11].

The most significant difference from already available generator generators is that in our work the data generators are generated not from context-free grammars, but from L-attributed grammars. The latter formal grammar class is much more expressive than the former one. The notation of the `eqc_grammar` is based on context-free grammars and cannot be used to describe context-dependent data. Since the Erlang language is in the latter group, a more expressive metalanguage is needed. The Erlang syntax and static semantics can be conveniently described by using L-Attributed grammars, so we decided to design a QuickCheck generator generator for such grammars.

3.1 Generator metalanguage

A grammar-based generator generator takes a proper grammar description and produces data generators according to the grammar rules, or in other words, to the grammar nonterminals. Then, the generator belonging to the root symbol generates strings of the language described by the grammar. We have to create a metalanguage that can denote L-attributed grammars and can be efficiently compiled into data generators.

As already noted, the notation aims to express L-Attributed grammars, which are able to describe inheritance in the grammar. For those who are familiar with the usual attribute grammar notation, we present a very simple attribute grammar in both the usual and the new notation to illustrate the difference. The example describes the $a^n b^n c^n$ language, which is one of the simplest non-context-free languages. Figure 1 shows it in usual notation and Figure 2 shows it in our EBNF-like notation.

The main structure of the both descriptions are similar. The grammar is written as a group of rules and inside the rules there may be alternatives. The main difference lies in the place where the attribute computations are written. In the usual notation the semantic rules are separated from the common grammar rules. In contrast, in our notation the attribute computation rules are written on the spot, just after the entity to which the attributes belong.

$$
\begin{array}{lll}
\langle\text{abcSeq}\rangle ::= & \multicolumn{2}{l}{\langle\text{aSeq}\rangle\ \langle\text{bSeq}\rangle\ \langle\text{cSeq}\rangle} \\
 & \multicolumn{2}{l}{\quad \textit{InSize}(\langle\text{bSeq}\rangle) \leftarrow \textit{Size}(\langle\text{aSeq}\rangle)} \\
 & \multicolumn{2}{l}{\quad \textit{InSize}(\langle\text{cSeq}\rangle) \leftarrow \textit{Size}(\langle\text{aSeq}\rangle)} \\
\langle\text{aSeq}\rangle ::= & \mathbf{a} & \\
 & & \textit{Size}(\langle\text{aSeq}\rangle) \leftarrow 1 \\
 & \mid & \langle\text{aSeq}\rangle_2\ \mathbf{a} \\
 & & \textit{Size}(\langle\text{aSeq}\rangle) \leftarrow \textit{Size}(\langle\text{aSeq}\rangle_2) + 1 \\
\langle\text{bSeq}\rangle ::= & \mathbf{b} & \\
 & & \textit{Condition: InSize}(\text{bSeq}) = 1 \\
 & \mid & \langle\text{bSeq}\rangle_2\ \mathbf{b} \\
 & & \textit{InSize}(\langle\text{bSeq}\rangle_2) \leftarrow \textit{InSize}(\langle\text{bSeq}\rangle) - 1 \\
\langle\text{cSeq}\rangle ::= & \mathbf{c} & \\
 & & \textit{Condition: InSize}(\text{cSeq}) = 1 \\
 & \mid & \langle\text{cSeq}\rangle_2\ \mathbf{c} \\
 & & \textit{InSize}(\langle\text{cSeq}\rangle_2) \leftarrow \textit{InSize}(\langle\text{cSeq}\rangle) - 1
\end{array}
$$

Figure 1. $a^n b^n c^n$ grammar in usual AG notation

```
1  abc_seq -> a_seq
2           ~> b_seq [@size = '$1'.size]
3              c_seq [@size = '$1'.size].
4
5  %% a_seq -> a        :: [@size = 1]
6  %%         | a_seq a :: [@size = '$1'.size + 1].
7  a_seq -> {a} :: [@size = length('$1')].
8
9  %% b_seq -> (when size_is_1) b
10 %%         | b_seq [@size = '$0'.size - 1] b.
11 b_seq -> {'$0'.size, b}.
12
13 c_seq -> {'$0'.size, c}.
```

Figure 2. $a^n b^n c^n$ grammar in our notation

This kind of formalism fits well with the constraints of L-Attributed grammars, where, due to the restrictions of the inheritance, attribute computations may refer only to their left. In our notation, the attribute computation section can refer only to symbols being on its left. This approach is similar to the sequential programming style, in which a statement may only refer already declared variables.

Due to the fact that attribute computations are written just after the symbol that they affect, the nonterminals do not have to be indexed on the right hand side, since the position of the attribute computation rule determines on which symbol it is defined. Synthesised attributes are given separately at the end of the rule.

In *line 7* (Figure 2) in comparison with the usual formalism we can see a useful simplification, that is, one can use repetition (lists) instead of primitive recursion, which will be shorter and easier to understand. In yecc (and therefore in eqc_grammar) rule alternatives and repetition are not supported, so with our formalism it is easier to express grammars, because it is closer to EBNF rather than to BNF.

Moreover, as can be seen in *line 9*, in order to make the notation express conditions based on attribute values, we added support for guards in rule alternatives. In *line 2* and *line 3* the setting of the inherited attributes is shown, and then in *line 11* and *line 13* it can be seen how the attributes can be accessed. To ease the attribute computation, one can use any kind of Erlang expression to compute the attribute value. As should be evident, the notation is concise and is similar to the usual AG formalisms.

[1] Yecc is an LALR-1 parser generator for Erlang, similar to yacc.

Special grammar rules: embedded rules

Generating the right hand side belonging to a grammar rule theoretically is a single atomic step. That is, every value belonging to the symbols on the rule's right-hand side are generated simultaneously. While generating, for example, an Erlang function clause, apparently the generation of the clause patterns and of the clause body may not be simultaneous, since the body may well depend on the patterns, or formal parameters, in the function head. In such cases the right hand side values of the grammar rule may not be generated together in a single round. To denote this, we use a special arrow symbol in the grammar description, which separates the different parts of the rule. In other words, productions may be regarded as sequences of separately generated right hand side element groups, where the groups are separated by ~> symbols. After every such (possibly empty) group one can write any Erlang code and can manipulate the current attributes.

Consider a simple definition of Erlang clauses, in which a clause consists of a formal parameter list (patterns) and a body (expressions). Obviously, the formal parameters and the body of a clause are semantically interdependent, since the variables bound in the patterns might be used inside the clause body. Therefore, the generation of the expressions is embedded into the generation of the patterns. In other words, the two generation steps are performed in sequence. After generating the proper pattern and expression lists, a subtree is synthesised that accords to the function clause, like this:.

```
function_clause -> {~ N, pattern}
                ~> {~ M, bodyexpr}
                :: create_clause('$1', '$2').
```

The embedded rules are compiled into applications of the *bind* built-in QuickCheck generator combinator. By using this combinator, we get a monadic-style execution of the value generators (in Haskell QuickCheck this feature is implemented with monads).

Special grammar rules: recursive rules

As explained, earlier, formal grammars mainly consist of grammar rules. Basically, a grammar rule has a nonterminal on its left hand side and a list of either terminals or nonterminals on its right hand side. The rule defines the meaning, the way of production of the nonterminal being on the left. If that symbol also appears on the right, the rule is said to be recursive. Recursion might be indirect as well, that is, the rule's right contains a nonterminal whose definition refers to the current rule. Some of the recursive rules can be eliminated by using repetition, the others have to be handled or modified properly in order to avoid infinite recursions.

Repetition By applying EBNF-style repetition, many primitive recursive rules can be eliminated from grammar descriptions. Usually, when generating lists of entities, in BNF one has to create a primitive recursive rule, which has both a 'productive' and a 'base' alternative. Consider the following example which may generate lists of expressions. You can see that the recursion can be eliminated by using repetition.

The recursive description:

```
exprs -> expr exprs
       | expr
```

And the same rule by using repetition:

```
exprs -> {expr}
```

According to the actual context, one can use repetition in two different ways depending on the way of handling the attributes. Also, lists of entities may be generated with a given (fixed) size or else a randomly generated size. The generation of repeated parts is basically implemented by using QuickCheck's *list* and *vector* generator combinators, the former for variable sized lists and the latter for lists with a given size (the size parameter can be either a variable name, a constant or a macro/function call). However, if the generation of the list elements may be interdependent, that is, certain list elements may depend semantically upon each other, then the generation gets more difficult. In the latter case, special auxiliary generator combinators are included into the resulting source code, which help the generation of dependent lists.

While independent list elements are generated simultaneously and all elements inherit the same attribute list from their parent (in other words, every list element is generated over the same attribute list and cannot affect each other), in dependent list generation, elements are generated one after the other and each one inherits the attributes from the previous one. That is, the generated attributes flow through the list and the currently generated elements can affect the following siblings. The method and the notation is quite similar to rule embedding. In this case, we would say that all sublists are embedded. Dependent repetition symbols and embedded rules can be seen in the following example.

```
module   -> {attribute} {~ ?M, function}.
function -> {?N, clause}.
clause   -> {~ pattern} {~ expr}.
```

Controlling recursion While using a grammar description for parser generation, the alternatives are equivalent in the respect of applicability, since the input string determines which alternatives have to be used for reduction. During a random generation, in the case of rules built up from many alternatives the framework should somehow choose one of them. In our solution, the generator randomly makes a choice among the alternatives and the selected one is going to be used for generating the current subtree. Obviously, if the alternatives are equivalent, the mentioned choice is totally random, all the alternatives have the same chance to be chosen. However, in some cases it is expected to make a kind of priority order among the rule alternatives in order to control the structure and the properties of the randomly generated data.

In a rule, all rule alternatives may be associated with a frequency (or weight), based on which they will be chosen. Obviously, by adjusting the probability of the different alternatives the generated data structure accordingly changes. An alternative's weight can mean its relevance as well as the complexity of the subtree that may be generated by its application. In the case of primitive recursive rules (for example, generating list data structures) the weight of the rule alternatives may affect the size of the generated data.

Theoretically, recursive generation should terminate by a proper setting of probabilities. However, in practice, structurally recursive generation processes may not terminate, instead, infinitely enlarge the generated structure. To avoid infinitely recursive application chains, a recursion depth limit was injected into the generation process. The current depth of the recursion is registered during the rule applications, more precisely, a counter registers the number of the available recursive calls before hitting the limit. The counter is decreased every time when a recursive call is performed. If the limit is hit (in other words, the counter becomes zero), then only simple (usually non-recursive) alternatives can be applied. Thus, the generation terminates on the current subtree. This integer expression generator shows this in action:

```
intexpr(0) -> int :: erl_syntax:integer('$1').

intexpr(N) ->
    intexpr(0)
  | intexpr(decr(N)) infixop intexpr(decr(N))
 :: erl_syntax:infix_expr('$1',
        erl_syntax:operator('$2'), '$3').
```

The 'simple' rule alternatives could be found by analysing the recursiveness of the right hand side, but in our metalanguage the programmer has to mark the non-recursive, simple cases. In our formalism the rules are written in a function-like format and in particular they can have arguments. The mentioned counter registering the depth of the recursion is manually decreased and passed to the (directly or indirectly) recursive symbols. This solution gives the programmer full control over the recursive generation.

3.2 An Erlang grammar definition

The generator generator framework is applicable to produce any kind of data that can be described using formal grammars. In our case, we have used the framework to generate Erlang programs. Consequently, we have created a grammar definition of the Erlang language, more precisely, a definition of the sequential programming language elements.

First of all, we had to decide, at what abstraction level to generate programs, since source code can have many kinds of representation, such as the well-known textual representation, token stream, or abstract syntax tree (AST). This decision will in turn determine the further difficulties of the description, because different representations may introduce different problems during the generation process.

We decided to use the latter representation, namely Erlang Syntax Trees. Such trees can easily be represented and handled using the Erlang Syntax Tools library, which is included in the standard Erlang distribution and includes modules declaring useful types and functions helping in construction and pretty-printing such syntax trees. With the functions of the `erl_syntax` module it is simple to create ASTs in a bottom-up strategy. By using the Syntax Tools application we only have to focus on the generation of abstract syntax trees instead of the textual program code. Compiling the grammar description we can get a QuickCheck generator that can produce random, compilable Erlang source code. In the background, the generation method creates an Erlang syntax tree which is then pretty-printed.

Using the current language definition we can generate any number of modules containing random function definitions that may refer functions from another generated modules. Functions may have randomly one or more function clauses, which do not shadow each other and have randomly generated patterns. The function bodies may contain many kinds of Erlang expressions, including IO statements, case expressions and match expressions as well. Moreover, generated code may invoke library functions. Match expressions can bind variables, and other expressions may refer those variables (but cannot re-bind them) afterwards.

Every generated language element is well-typed, since types are managed by storing related informations in attributes. The types used during the generation are randomly generated as well.

Finally, many properties of the generated code can be parameterised, such as the number of the generated functions, the difficulty of the generated expressions and the maximum level of nested case expressions. By adjusting the grammar and the parameters we can make the generated code quite similar to real-world programs.

3.3 Transformation

We have implemented a compiler (a generator generator) for our grammar definition, which produces a single Erlang module containing functions returning QuickCheck generators for each production rule preserving its meaning. The compiler uses the standard Erlang scanner (with some extensions) and a yecc-generated parser. After scanning and parsing the grammar definition, firstly it checks some constraints on the grammar (for example, every declared nonterminal is defined as a rule, the are no symbol duplications, every right hand side symbol is defined in the file), then generates the

```
-module(prop).

-export([prop/0]).

-include_lib("eqc/include/eqc.hrl").

prop() ->
    test:prop_beh_eqv(rename_mod,
                      fun gen_rename_mod_args/1).

gen_rename_mod_args(Filename) ->
    ?LET(Atom, test:gen_atom(),
         [Filename, Atom, [], 8]).
```

Figure 3. Example of rename module property

output Erlang module: if the input file is `abc.eyrl`, then the output file will be `abc.erl`, which is constructed of the generator functions belonging to the grammar rules, the attached Erlang code cut from the grammar definition file (without any changes) and further essential function and macro definitions being for the notation features.

The generated output file is checked whether it compiles or not (using the Erlang compiler strong validation). If the module is compilable, with the `erl_tidy` module (included in Erlang Syntax Tools) it is tidied (unused functions are removed from the code, some syntax constructs are rewritten in a more readable form) and then compiled again, for reasons of optimisation.

Despite the fact that in the generated Erlang module every function can be called from the outside (that is, they all are exported), only the function belonging to the root symbol can be used without any parameters (the others require parameters carrying information about the attributes). The return value of this function is a valid QuickCheck generator and passing this generator to QuickCheck results in the expected random data.

4. Properties

If a refactoring was performed correctly, the behaviour of the program should not have changed: it should return the same output for the same input, throw the same exceptions, send the same messages in the same order. We say that the original and refactored versions are behaviourally equivalent.

To test behavioural equivalency we follow these steps: generate random programs, perform the refactoring, pick a function, generate random arguments guided by type information, evaluate the function and then compare the result of the two versions.

Since Erlang programs contain many functions, and a refactoring may only modify a single function we could pick an irrelevant function to test and miss an error. This is natural, and the solution is to run many tests to minimize the chance that we miss the erroneous function. A similar thing happens with arguments when the function has multiple clauses, which is fairly typical.

To minimize the chance of missing an error, we have to run a large number of test, so test execution speed matters. The two slowest phases are program generation and the refactoring itself. By running the later phases more than once after every refactoring we can reduce the chance of missing errors without having to execute the slower steps repeatedly as many times.

Erlang is a dynamically typed language, which means that we can supply any term as argument to a function, and at the worst case we get a runtime exception. Arguably this doesn't help with catching real bugs. The dialyzer tool [17] can infer type information for functions from a codebase. This makes it possible to generate proper arguments for the function under test, so that we can avoid programs under test failing for reasons of data being ill-typed.

In Erlang I/O uses message passing under the hood, hidden from the user. The messages are in a certain format and are sent to an I/O device, which is a separate process. The format of the messages is well documented, and any process that can handle them can be used as an I/O device. The testing framework uses an I/O device which behaves like a ram file that is initially empty. This I/O device additionally keeps track of the messages received.

To test behavioural equivalency we evaluate both the old and the new version of a function, and compare the outputs and I/O traces. This can be done concurrently, saving us time. This is particularly true when the functions are non terminating due to the random nature of them. In this case we halt the evaluation after a given time. Concurrent execution means we only have to wait for this timeout only once, halving the time needed to test.

4.1 Example property

In order to define a behavioural equivalence property for a refactoring, the user has to call `test:prop_beh_eqv/2` with the appropriate arguments. The first argument is an atom, which is the name of the function in the `wrangler` module that implements the refactoring. The second parameter is a callback function, that given a filepath should return suitable arguments for the refactoring function. In the simplest case the callback returns a list containing the arguments. If the testing should be restricted to a specific function, the return value should be a three-tuple with the function name and arity for the said function and the argument list for the refactoring function.

There is another version of `test:prop_beh_eqv`, which takes an additional third argument. This is a callback function that receives the result of the previous callback function and returns a boolean indicating whether to proceed with the test or not.

The simplest property is for the rename module refactoring, the whole code is given in Figure 3.

5. Results

We have designed and implemented a notation for L-Attributed grammars and created a compiler for it. So far we have formalised a subset of the Erlang language with it and got promising results.

Moreover, by using random generation we have implemented the testing of four refactoring steps provided by Wrangler: rename variable, rename function, generalise function and tuple function arguments. We have found two bugs, one in rename variable regarding incorrectly handling patterns in function parameters, and one in generalise function regarding incorrectly transforming the function leading to compiler errors. Both of these errors are fixed in the latest release of Wrangler.

6. Related work

[9] is a similar study done for refactoring engines integrated in IDEs for mainstream languages. The program generator described is specific to the language they use and it can be parametrized by code fragments, so it would be difficult to adapt to other domains. They test the results of refactorings in a different way too: they test hand written structural properties as opposed to behavioural equivalence, and do this by testing the results of two different refactoring engines against each other, rather than testing the old and new code directly.

[14] describes previous work using QuickCheck for testing Wrangler. They did not use random program generation, refactoring a static codebase instead, and the only property formulated is successful compilation of the refactored program.

7. Conclusions and future work

We have demonstrated that automated, property-based random testing of refactoring tools is able to discover new bugs, and therefore it is a useful addition to the testing processes of tool developers.

How to check behavioural equivalence of arbitrary message passing programs, or refactorings which have wider ranging effects is still an open question.

Our main contribution is random program generation and behavioural equivalence testing, which together give much wider coverage, scalability and maintainability to the testing of refactoring engines.

References

[1] Quviq QuickCheck, June 2010. http://www.quviq.com/.

[2] RefactorErl, June 2010. http://plc.inf.elte.hu/erlang/.

[3] Wrangler, June 2010. http://www.cs.kent.ac.uk/projects/forse/.

[4] T. Arts and J. Hughes. Erlang/QuickCheck. In *In Ninth International Erlang/OTP User Conference*, November 2003.

[5] T. Arts et al. Testing Telecoms Software with Quviq QuickCheck. In *ACM SIGPLAN workshop on Erlang*. ACM Press, 2006.

[6] T. Avgerinos and K. F. Sagonas. Cleaning up erlang code is a dirty job but somebody's gotta do it. In *Erlang Workshop*, pages 1–10, 2009.

[7] J. W. Backus et al. Revised report on the algorithmic language ALGOL 60. 1997.

[8] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 1956. doi: 10.1109/TIT.1956.1056813.

[9] B. Daniel et al. Automated testing of refactoring engines. In *ESEC/FSE*, pages 185–194, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-811-4.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, July 1999. ISBN 0-201-48567-2.

[11] D. Horpácsi. Testing refactoring tools by generating random Erlang modules. Master's thesis, ELTE, Budapest, Hungary, 2010.

[12] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 1968. doi: 10.1007/BF01692511.

[13] P. M. Lewis et al. Attributed translations (Extended Abstract). In *STOC '73*. ACM, 1973.

[14] H. Li and S. Thompson. Testing Erlang Refactorings with QuickCheck. In *IFL*, pages 19–36, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-85372-5.

[15] W. F. Opdyke. Refactoring object-oriented frameworks. Technical report, 1997.

[16] J. Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.*, 1995. ISSN 0360-0300.

[17] K. F. Sagonas. Experience from Developing the Dialyzer: A Static Analysis Tool Detecting Defects in Erlang Applications. In *Workshop on the Evaluation of Software Defect Detection Tools (Bugs'05)*, 2005.

[18] K. F. Sagonas and T. Avgerinos. Automatic refactoring of erlang programs. In *PPDP*, pages 13–24, 2009.

[19] L. P. Sobotkiewicz. A New Tool for Grammar-based Test Case Generation. Technical report, University of Victoria, 2008. MSc thesis.

[20] N. Sultana and S. Thompson. Mechanical Verification of Refactorings. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM Press, 2008.