Testing Erlang Refactorings with QuickCheck

Huiqing Li and Simon Thompson

Computing Laboratory, University of Kent, UK
{H.Li, S.J.Thompson}@kent.ac.uk

Abstract. Refactoring is a technique for improving the design of existing programs without changing their behaviour. Wrangler is a tool built at the University of Kent to support Erlang program refactoring; the Wrangler tool is written in Erlang.
In this paper we present the use of a novel testing tool, Quviq QuickCheck, for testing the implementation of Wrangler. QuickCheck is a specification-based testing tool for Erlang. With QuickCheck, programs are tested by writing properties in a restricted logic, and using the tool these properties are tested in randomly generated test cases.
This paper first gives overviews of Wrangler and Quviq QuickCheck, then discusses the various ways in which refactorings can be validated, and finally shows how QuickCheck can be used to test the correctness of refactorings in an efficient way.

1 Introduction

Refactoring [7] is a technique for transforming program source code in such a way that it changes the program's internal structure and organisation, but not external behaviour. The key characteristic that distinguishes refactoring from general code manipulation is its focus on structural change, strictly separated from changes in functionality. Functionality-preservation requires that refactorings do not introduce (or remove) any bugs. Refactorings typically have two aspects: *program analysis* is required to check that certain side-conditions are met by the program in question in order for the refactoring to preserve behaviour, and *program transformation* which carries out the actual program restructuring. In a slogan: *"Refactoring = Condition + Transformation"*.

Refactorings can be done manually, but this can be tedious and error-prone for small programs, and impractical for larger systems. Software tools (*"refactoring engines"*) can help programmers perform refactorings automatically, and are available for a variety of languages, including Smalltalk, Java, C#, C++, Haskell, Erlang, etc. With a refactoring tool, the programmer only needs to select which part of the program to be refactored and which refactoring to apply, and the tool will automatically check the side-conditions and apply the transformation throughout the whole program if the side-conditions are satisfied. Wrangler is the tool that we are implementing to support refactoring Erlang [1] programs, and this forms one aspect of 'Formally-Based Tool Support for Erlang Development'[1] [6], a joint research project between Universities of Kent and Sheffield.

[1] FORSE is supported by EPSRC, UK.

Implementing a practical and usable refactoring tool for a real world programming language is by no means trivial. A refactoring tool needs to get access to the program's syntax and static semantics (possibly including type information), to implement different kinds of program analysis and transformation, and to preserve the comments, and potentially, layout, of the transformed program. Among other criteria, such as efficiency, usability and completeness, the reliability of a refactoring tool is vital for it to be accepted in practice. A bug within a refactoring tool can introduce bugs in the refactored programs silently, and such bugs may be impossible to detect statically, if they result in a valid program which behaves differently from the original.

The correctness of refactorings implemented can be ensured from several aspects including, but not limited to, a clear specification clarifying the pre-conditions, transformation rules, and/or post-conditions of each refactoring; a verification that argues the correctness of the specification, and most importantly a thorough testing of the refactoring tool. A traditional way of testing refactoring tools is to create test cases manually. Each test case contains an input program, a refactoring command, and the expected result, which could be either the refactored version of the input program or the original input program (along with a failure message) depending on whether the side-conditions are satisfied. Then these tested cases are usually run with a unit testing tool, such as EUnit [3] for Erlang. Writing test cases manually is tedious and hard to cover all possible refactoring scenarios. Incomplete test suite potentially leaves bugs in refactoring tools.

We present the technique of using Quviq QuickCheck [8], a tool developed by Quviq AB, to automate the testing of Wrangler. Instead of writing small test programs, we use real-world available Erlang programs as our refactoring input programs. For example, one of the Erlang programs we have used is Wrangler itself, which currently contains 25 modules, 20K lines of code in total. Quviq QuickCheck tests running code against formal specification, using controllable random test case generation combined with automated test case simplification to assist error diagnosis. With Quviq QuickCheck, we automate the generation of refactoring commands and the checking of refactoring outputs. Refactoring commands are generated randomly using the information stored in the annotated abstract syntax tree (AAST) of the input program. Along with the development of each refactoring, we write a collection of properties that the refactoring should satisfy. Failing to meet one or more of these properties indicates bugs in the implementation or properties. Each time the testing is run, it generates 100 refactoring commands by default, applies each command to the input program, and checks that the properties being tested return true in every case. This way, we are able to integrate the specification and testing of refactorings very naturally.

The rest of the paper is structured as follows. In sections 2 and 3, we give introductions to Wrangler and Quviq QuickCheck. Section 4 gives an overview of the different ways in which refactoring engines can be tested, and in section 5 we explain our approach to testing Wrangler with QuickCheck, including the

generation of refactoring commands, and the kind of general properties that we use to test refactorings. In section 6, as an example, we illustrate the testing of *renaming a function*. In section 7, we give an evaluation of our approach; related work is presented in section 8, and conclusions and future work are given in section 9.

2 Wrangler – An Erlang Refactorer

Wrangler [11,12] is the tool that we are building to provide support for interactive refactoring of Erlang programs. The current version of Wrangler supports a number of structural refactorings, including *rename an identifier*, *generalise a function definition*, *function extraction*, *move a function definition between modules*, *fold expressions against a function definition*, etc, and functionalities for duplicated code detection. More process structure related refactorings are being added.

Wrangler is built on top of the Erlang `syntax-tools` package [14] which provides a representation of the Erlang AST within Erlang. `syntax-tools` allows syntax trees to be augmented with additional information as necessary. The Wrangler AST representation is annotated with a variety of information:

- Comments in the source code are inserted as attachments to the nodes in the AST at the appropriate place.
- Each function or variable name is associated with its actual source location and the location of its defining occurrence, thus reflecting the binding structure of the program.
- The start and end location of each syntactic entity in the source code is also stored in the augmented AST, allowing entities to be located by means of their position, as well as supporting pretty-printing facilities.
- Category information indicating the kind of syntax phrase the AST node represents, such as expression, function, pattern and so on is also included in the tree.
- Finally, free and bound variable information is also attached to the AST representation of each syntax phrase in the source code.

Wrangler is embedded in the Emacs editing environment; to manage communication between the refactoring engine and Emacs we make use of the functionalities provided by Distel [13], an Emacs-based user interface toolkit for Erlang.

To perform a refactoring with Wrangler, the focus of refactoring interest has to be selected in the editor first. For instance, an identifier is selected by placing the cursor at any of its occurrences; an expression is selected by highlighting it with the cursor. Next, the user chooses the refactoring command from the *refactor* menu, and inputs the parameter(s) in the mini-buffer if required. The Wrangler tool checks that the focus item is suitable for the refactoring selected, that the parameters are valid, and that the refactoring's side-conditions are satisfied.

If all these checks are successful, Wrangler will then perform the refactoring, and update the program with the new result, otherwise it will give an error message, and abort the refactoring with the input program unchanged. *Undo* is supported by Wrangler; applying *undo* once reverts the program back to its state immediately before the last refactoring was performed.

Snapshots of Wrangler are given in Figures 1-2 with a particular refactoring scenario showing the generation of function `repeat/1`. In Figure 1, the user has selected the expression `io:format("Hello\n")` in the definition of `repeat/1`, has chosen the *Generalise Function Definition* command from the *Refactor* menu, and is just entering a new parameter name *A* in the mini-buffer. Then the user would press the *Enter* Key to perform the refactoring. After the side-condition checking and program transformation, the result of this refactoring is shown in Figure 2: the new parameter *A* has been added to the enclosing function definition `repeat/1`, which now becomes `repeat/2`; the highlighted expression has been replaced with `A()`; and at the call-site of the generalised function, the selected expression, wrapped in a *fun*-expression, is now supplied to the function call as its first actual parameter. We enclose the selected expression within a function closure because of its side-effect, so as to ensure that the expression is evaluated at the proper points. As a design decision, if the generalised function is exported by the current module, an auxiliary function is created to ensure that the interface of the module is unchanged, as shown in this example.

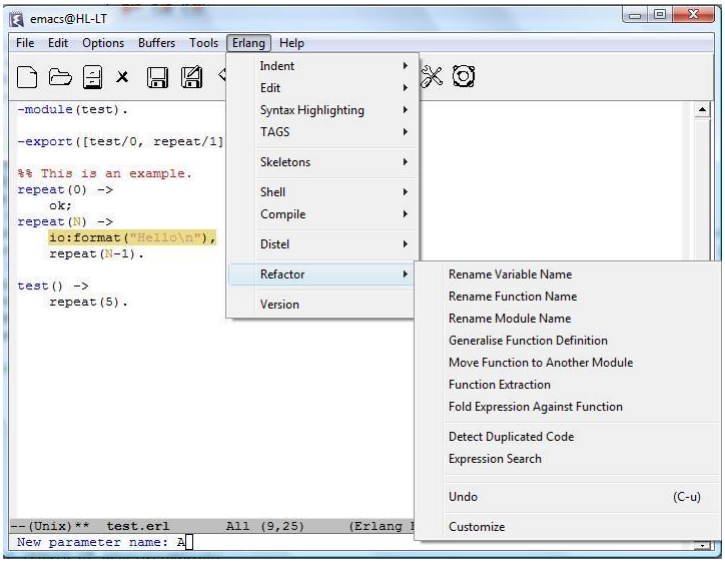

Fig. 1. A snapshot of Wrangler

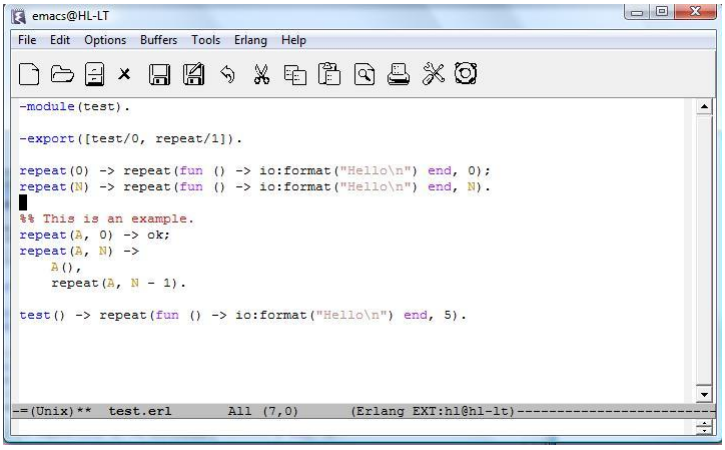

Fig. 2. A snapshot of Wrangler showing the result of generalisation

3 Quviq QuickCheck

Quviq QuickCheck is a property-based testing tool, developed from Claessen and Hughes' earlier QuickCheck tool for Haskell [4], re-designed for Erlang with a number of extensions, of which the most significant is an ability to simplify failing test cases automatically [8].

Quviq QuickCheck provides an API in Erlang that allows users to write *properties* that are expected to hold of programs; these properties are themselves expressed as Erlang source code. QuickCheck also defines a variety of *generators* and combining forms for generators by means of which the user can generate test data of the appropriate type and distribution for their needs.

As an example, consider the standard list reverse function. One property of this function is expressed thus:

```
prop_reverse() -> ?FORALL(Xs, list(int()),
                          list:reverse(list:reverse(Xs))== Xs).
```

As an abstract property, this says that reversing a list of integers twice has the result of returning the original list. In QuickCheck, the functions `int/0` and `list/1` are both data generators: `int/0` generates random integers, and `list/1` generates a list of elements generated by its argument. `?FORALL` is an Erlang macro. `?FORALL(X, Gen, Prop)` binds `X` to a value generated by `Gen` within the property `Prop`. The example property will be said to hold in QuickCheck if `list:reverse(list:reverse(Xs))== Xs)` holds for all values of `Xs` generated by `list(int())`.

The property is checked by running 100 random test cases generated by the generators, and reports success if all tests pass this. If any test case fails, the

(first such) failing case will be printed. 100 is the default value of the number of test cases generated in each run of QuickCheck, and this figure can be customised by the user.

A failing test case indicates bugs in either the implementation under test or the written properties. For example, testing the following property

```
prop_list_delete()->
    ?FORALL(I, int(),
        ?FORALL(List, List(int()),
            not (lists:member(I, lists:delete(I, List)))))
```

against the standard function `lists:delete/2` might report

```
Failed! After 37 tests.
-8
[5, -8, 12, -8, 9]
```

as `lists:delete(I, List)` only removes the first occurrence of I in List. Once a counterexample has been found, the *shrinking* functionality provided by Quviq QuickCheck will allow QuickCheck to minimise the failing case as much as possible. For the above example, the length of the counterexample data will be reduced, and the output above would be augmented by

```
Shrinking......(6 times)
-8
[-8,-8]
```

By writing properties in this style, a QuickCheck user can build up a formal specification, which is then checked against the implementation by QuickCheck. The mutual testing of implementation and specification ensures the correctness of both.

In comparison with traditional automated testing, as provided by systems such as EUnit [3], which runs the same set of tests repeatedly, QuickCheck allows the user to run many different tests with little effort, therefore has the potential to find more bugs. It is, of course, possible to re-run tests simply by re-using a seed value within the random generation, and so to ensure that regression testing takes place if required.

The API provided by QuickCheck contains functions for generating both simple and complex test data, according to distributions described by the user, as well as macros for writing and testing properties. In the following sections, an explanation will be given when an API function or macro is used.

4 Validating Refactoring Engines

Refactorings and refactoring engines can be validated in a number of different ways. In this section we present an overview of the various approaches and their pros and cons, before explaining our approach in more detail in the next section.

In checking whether the result of a refactoring has preserved behaviour, the result naturally needs to compile and run without errors; in the remainder of this section we assume that the results are also checked for being compilable as well as being tested in various ways we discuss.

4.1 Regression testing of refactored programs

The most popular means of validating refactorings in current use is to ensure that refactored code meets all the tests that the original version met. As Object-Oriented refactoring has been identified as one of the central characteristics of an extreme programming approach, it is reasonable to assume that the test data will already be in place, and so the advantage of this approach is that the cost of testing the refactored code is small. This approach means that the refactored code has the same warranty as the original code.

The approach has two limitations. First, the coverage of the code is necessarily partial, and so it is possible that bugs have been introduced in the untested parts of the code. Also, the testing cost can be higher in cases where the test cases have themselves to be refactored: for instance, if a function is generalised, then it is necessary to add an extra datum to the test data for each function call.

4.2 Testing the old and new programs

A variant of the previous approach tests the two versions of the program against each other: on input data taken from an existing test suite, the outputs from two versions of the program can be compared directly. This approach is lower cost in the case where there is no pre-existing test data, since it is not necessary explicitly to state the output values corresponding to the various input data. A disadvantage is that any framework needs to accommodate the co-existence of two versions of the code under test.

Neither this nor the previous approach actually checks the structural changes of the refactored code, and could fail to test that refactorings actually achieve their purpose. For example, program behaviour preservation can be achieved even if a malfunctioning refactoring returns the program structurally unchanged without giving an error message.

4.3 Programs as data

In contrast to the earlier approaches, it is possible to see the refactoring as a program, and so to supply it with a set of input programs and the corresponding output programs that are expect to result. Two variants of the check are possible:

- It is possible to analyse the abstract syntax tree (AST) resulting from the transformation, and to compare this with the expected result. This neglects the layout of the refactored program.

– In contrast, it is possible to specify the source code to be expected, with a given program layout. This is a stronger test than the former; since it not only prescribes the AST but also its particular layout, but this approach is appropriate when refactoring code is expected to be laid out in a way that will make it recognisable to its author.

This was the approach that we used first, using the Haskell package HUnit for testing HaRe (the Haskell refactorer) [10], and EUnit for testing Wrangler.

In our experience, the main disadvantage of writing test cases under this approach is that it is very tedious, and hard to cover all the refactoring scenarios especially when both the implementation and the test cases are written by the same people. Hence we did not gain sufficient assurance about the correctness of the refactorings implemented.

Other variants of this approach involve a degree of random generation; we will explore our particular approach in the next section, and discuss related work in section 8.

4.4 Program verification

Rather than using testing, it is possible to write formal proofs of correctness for refactoring engines. Two approaches suggest themselves:

– It is possible to produce, program by program, separate proofs of equivalence between the original and the refactored programs. Such proofs might be generated by tactic-based proof descriptions, or result from a proof planning process.
– Alternatively, the formal theorem proved can itself contain a quantifier over all programs of a certain form (which are the input to the refactoring in question). Preliminary work under this approach is to be found in Li's thesis [9] and the forthcoming thesis of Sultana [15].

This section has summarised various approaches to validating refactoring engines; we next look at our particular work.

5 Testing Wrangler with QuickCheck

Before adopting QuickCheck as the test engine of refactorings, we used the unit testing approach, as discussed in the previous section. We concluded that this mechanism was not ideal, and so to improve the testing of Wrangler, we have experimented with the idea of using Quviq QuickCheck as the test engine.

Under this approach, a collection of properties are written along with the implementation of each refactoring. These properties specify the conditions that must be met by the program after the refactoring, in order for the transformation to be behaviour-preserving. From the formal specification point of view, these properties can be viewed as the post-conditions of a refactoring. While there are some

general properties which apply to most of the refactorings, for example, all the programs after a refactoring must compile successfully, some properties are particular to individual refactorings, especially those involving structural changes to the program. Writing properties along with the implementation of refactorings, we are able to make testing an integral part of the refactoring development process.

Properties are tested on the refactored version of the input program. While occasionally we have written a few small input programs to test a particular case, mostly we use real-world Erlang programs as the testing code base. Before the testing of a specific refactoring, the code base could be examined to make sure that enough refactoring scenarios are covered in the program. For example, to test a refactoring involving the communication between processes, we should choose programs that contain substantial process communications; and to test a refactoring that transforms a tuple to a record, we need to make sure that tuples and records are reasonably used in the test program. Apart from manual examination, the `collect/1` function provided by Quviq QuickCheck can be used to analyse the distribution of the test data when the testing is complete; and the coverage analysis functionalities provided by the standard Erlang release can be used to analyse how well the code implementing the refactoring is covered by running the test cases.

Once the test program has been chosen, refactoring commands are automatically generated using the information stored in the annotated abstract syntax tree (AAST) of the test program. Both the generation of refactoring commands and the creation of properties make use of the Wrangler infrastructure API. The API provides programmer access to the infrastructure on which Wrangler is built. As the infrastructure has been more thoroughly tested, we trust its robustness in this exercise. Alternatively, we can also test an API function exposed by the infrastructure using the same approach.

More about the generation of refactoring commands and the creation of properties are discussed in the following two sub-sections. Following that, as an example, testing of the *renaming a function* refactoring is examined in more detail.

5.1 Generation of Refactoring Commands

In Wrangler, a refactoring command normally contains the refactoring name, the name of the source file under refactoring, the focus of the refactoring which can be a location/range in the program source, and some user inputs. For example, the refactoring *renaming a function* has the following interface:

```
rename_fun(FileName, SrcLoc={Line, Col}, NewName, SearchPaths)
```

where `FileName` is the name of the file containing the definition of the function to be renamed; `SrcLoc`, which is a tuple containing a line and a column number, represents the location of one of the occurrences of the function name in the source; `NewName` is the new function name, and `SearchPaths` specifies where to

search for those files that could possibly use this function; this is needed when the function to be renamed is exported by the module in which it is defined.

As another example, the refactoring *generalisation of a function definition* has the following interface:

```
generalise_fun(FileName, Range={StartLoc, EndLoc}, ParName)
```

where `FileName` is the name of the source file containing the definition of the function to be generalised; `Range` represents the start and end location of the selected expression in the source, and `ParName` is the new parameter name. As this refactoring only affects the current module, `SearchPaths` is not needed.

Next, we return to the 'renaming' example to explain how refactoring commands can be generated. If a specific file is used as the input program, then the `FileName` is fixed, otherwise a file can be randomly chosen from a directory for each refactoring command. The following function serves to select an Erlang file from a directory.

```
gen_filename(Dir) ->
    {ok,Files} = file:list_dir(Dir),
    ErlFiles = [F|| F <-Files, filename:extension(F)==".erl"],
    oneof(ErlFiles).
```

where the function `oneof/1` is a QuickCheck API function which generates a value using a randomly chosen element of a list of generators; in this example, all the list elements are constant generators.

Instead of generating source locations using the integer generators provided by Quviq QuickCheck, the value of `SrcLoc` is generated based on the location information stored in the AAST representation of the chosen Erlang file. As discussed earlier, in the AAST, each occurrence of a function name is associated with its location in the source, the name of the module in which it is defined, as well as its defining location in that module.

To generate a source location, we first collect all those locations which are associated with the occurrences of function names defined in this file, then choose one from the collection randomly. This way, we can make sure that selected location points to a function name defined in the current module. In order to test the case when the user deliberately points to a location in the source which does not correspond to a function name defined in the module, we can always add fake locations to the collection of real ones, or make use of QuickCheck's fault injection combinators: `fault/1` and `fault_rate/3`.

Some refactorings ask the user to input a new name. For example, to rename a function, the user needs to input the new function name; and to generalise a function definition, the user has to input a new variable name. To improve the possibility that a name conflict/shadow occurs, identifier names are generated from both pre-created fresh names and those used in the refactored program,

since a name conflict/shadow is possible only when the new name is already used by program.

The following function generates refactoring commands for *renaming a function*.

```
rename_fun_commands(Dir) ->
        ?LET(FileName, gen_filename(Dir),
        {FileName,
         oneof(collect_fun_locs(FileName)),
         oneof(collect_names(FileName)),
         Dir}).
```

In the above function, `Dir` specifies where to look for Erlang files to refactor; `?LET` is a macro provided by Quviq QuickCheck (`?LET(Pat, G1, G2)` generates a value from `G1`, binds it to `Pat`, then generates a value from `G2` which may refer to the variables bound in `Pat`); function `collect_fun_locs/1` adds all the locations where a locally defined function name occurs in the selected Erlang file to a list of default locations; `collect_name/1` adds all the function names that occur in the source to a list of pre-created fresh identifiers, and as last, we assume that `Dir` is the only directory to search for those files that would possibly be affected by the refactoring.

Suppose that the testing directory is `"c:/wrangler-0.1/test"`, which has three Erlang files, the following shows part of the refactoring commands generated by the above function in one run of QuickCheck.

```
1% {"test.erl",{3,1},module,"c:/wrangler-0.1/test"}
1% {"refac_rename_fun.erl",{243,64},halt,"c:/wrangler-0.1/test"}
1% {"refac_qc.erl",{184,48},ordsets,"c:/wrangler-0.1/test"}
1% {"test.erl",{5,39},"DDD","c:/wrangler-0.1/test"}
1% {"refac_qc.erl",{366,30},get_pos,"c:/wrangler-0.1/test"}
1% {"refac_rename_fun.erl",{117,33},purge_module,"c:/wrangler-0.1/test"}
```

As an example, the first command means to rename the function whose name occurs at the location: {`line: 3, column: 1`} in file `test.erl` to the new name `module`, and search the directory `"c:/wrangler-0.01/test"` for files in which the function is used, if the function is exported. The percentage at the beginning of each line shows the proportion of the total represented by the command.

5.2 Properties

Formally specified or not, each refactoring comes with a set of pre-conditions, which embody when a refactoring can be applied to a program without changing its meaning; a set of transformation rules which state how the program should be transformed to fulfil the refactoring while keeping the program's semantics unchanged; and a collection of post-conditions which articulate some properties the program should hold after the refactoring has been done. While the pre-condition checking and transformation rules are always explicitly implemented, the checking of post-conditions are normally ignored by the developers of refactoring tools

as we assume that the pre-conditions and transformation rules together should guarantee the post-conditions.

With the QuickCheck testing approach, we can test most of these post-conditions explicitly. Ideally, one post-condition that applies to any refactoring is that the input program and its refactored version should have the same semantics; however whether two programs have the same semantics is in general not decidable. Furthermore, even when the two programs have the same semantics, the refactor still might not have performed the anticipated structural change to the program correctly as mentioned before. Therefore, instead of checking two programs having the same semantics, we test a number of properties that are decidable.

There are a couple of basic properties that should hold by all the refactorings:

- first, the refactoring engine should not crash, i.e. the refactorer should not terminate with an uncaught exception;
- second, if the refactoring has finished without giving an error message, then the refactored version of the program should compile successfully (Wrangler only refactors programs that compile).

Many basic refactorings are bi-directional. Given a refactoring that transforms a program from P to P', we can generally find another refactoring that transforms program P' to P. For example, renaming an entity in a program from A to B, then renaming it back to A, should produce the original program; as another example, first generalising a function definition over an expression, then specialising the function on the newly added parameter with the expression should always produce the original function. This feature of refactoring allows us to write properties that embody mutual testing of refactorings.

During the implementation of Wrangler, we always try to separate the precondition checking part from the transformation part. One of the benefits of doing this is that it allows the mutual testing of condition-checking and transformation. For example, performing the transformation with the knowledge that some of the necessary side-conditions are not satisfied should either make the refactoring engine crash or violate some post-conditions in the case that the transformation (apparently) succeeds.

Apart from those general post-conditions that apply to most of refactorings, each refactoring also has it own particular post-conditions, especially those concerning structural changes of the program, as different refactorings change the program structure in different ways. For some refactorings, there may also be special constraints that should hold during the transformation. For example, some refactorings are suppose to keep the program's module interface unchanged; while others are suppose to keep some particular function interfaces unchanged. All these constraints can be expressed as QuickCheck properties.

There is no limit on the number of properties one can specify to test a refactoring. For complex transformations, instead of writing a small number of very complex properties, we can always write a collection of simpler properties, each of which

specifies only one aspect or a small step of the transformation. Simpler properties are easier to understand, maintain and reuse.

In the following section, we again take the *renaming a function* refactoring as an example to illustrate how properties can be specified and tested.

6 An Example: Testing *Renaming a Function*

Renaming a function is one of the most basic, but very useful, refactorings, supported by almost all the existing refactorers. This refactoring renames a user-selected function name to a new name and updates all the references to it. When the renamed function is exported by the module, this refactoring could potentially affect every module in the program. Suppose the old and new function names (with arity) are `bar/n` and `foo/n` respectively, then the side-conditions on *renaming a function* are as follows.

1. The new name should be a lexically valid function name, otherwise the transformed program will not compile.
2. No binding for `foo/n` may exist in the same scope. This condition avoids *name conflict* in the scope where `bar/n` is defined, and violating this condition will result in the transformed program failing to compile.
3. No binding for `foo/n` may intervene between the binding of `bar/n` and any of its uses, and the binding to be renamed must not intervene between existing bindings and the uses of `foo/n`.
   This condition avoids *name capture*, and violating this condition will lead to the binding structure of the program being changed silently. ('Binding structure' here refers to the association of uses of identifiers with their definitions in a program, and is determined by the scope of the identifiers).
4. Callback functions should not be renamed. Callback functions in Erlang are generally named by Erlang OTP behaviours, and must be implemented by the module calling an OTP behaviour. Renaming, or changing the interface of, callback functions in either single side will break the protocol between the OTP behaviour and the module calling the OTP behaviour, and make the program fail to function properly.

To check the correctness of the implementation, we focus on defining properties depending on whether the refactoring succeeds or not. If the refactoring completes without giving an error message, we then test the following properties.

- Renaming the new function back to its original name should affect the same set of Erlang files in the application, and produce the original program except for variations of layout. This property also implies the condition that the refactored version of the program should compile without errors.
- The function-level binding structure of the refactored version of the program should be the same as, or isomorphic to, that of the original program.
  Unlike some functional languages that allow nested function definitions, Erlang has a very straightforward function defining structure. In Erlang, all

named functions are top-level functions. The function-level binding structure of an Erlang program can be represented as a list of tuples:

$$B = [\{\{M_1, Loc\}, \{M_2, Id, A\}\}]$$

and $\{\{M_1, Loc\}, \{M_2, Id, A\}\} \in B$ if and only if the function name Id, which occurs in module M_1 at location Loc, refers to the function defined in M_2 whose name is Id and arity is A. To take the possible program layout change into account, Loc here is a number reflecting the function name's textual occurrence order in the code, instead of the concrete source location.
Suppose the function `bar/1` defined in module `N` is renamed to `foo/1`, and the binding structures of the program before and after the refactoring are B and B' respectively, then replacing all the occurrences of $\{N, \texttt{foo}, 1\}$ in B' with $\{N, \texttt{bar}, 1\}$ should produce B.
This property is able to find certain bugs that escape detection by the previous property. For example, an implementation that renames every occurrence of the selected function name irrespective of its semantics will be found faulty by this property, but not necessarily by the previous property.

- The programs before and after the refactoring should have the same set of callback functions if which functions are callback functions has been explicitly specified.

If the refactoring fails because one of the side-conditions fails, then the necessity of the side-condition can also be tested. For example

- Transforming the program when side-condition 1 or 2 does not hold should produce a program that does not compile.
- Transforming the program when side-condition 3 does not hold should produce a program that compiles but has a different function-level binding structure.

A simplified version of the top-level function for testing *renaming a function* is given in figure 3. To make it easier to read, we have omitted the part that handles client modules, however this should not affect the idea expressed by this function.

7 Evaluation of Approach

A number of other refactorings have been tested using this approach, including *renaming a variable name*, *generalisation of a function definition*, etc. We actually started to use Quviq QuickCheck after the first preliminary release of Wrangler, which was tested on a number of small test cases using EUnit, and was also manually tested on a large code base.

Even so four bugs were found within the first release of Wrangler in a short time. All these bugs escaped the pre-release testing due to the incomplete coverage of the testing suite. Among these bugs, one silently changed the binding structure

```
qc_rename_fun(Dir) ->
  F = ?FORALL(C, (rename_fun_commands(Dir)),
  begin
    [FileName, SrcLoc, NewName, SearchPaths] = C,
      %% backup the current version of the program.
    file:copy(FileName, "temp.erl"),
      %% get the function name (with arity) to be renamed.
    {Mod, FunName, Arity} = pos_to_fun_name(FileName, SrcLoc),
      %% calculate the binding structure of the current program.
    B1 = fun_binding_structure(FileName),
      %% get the name of the callbacks functions if there is any.
    CallBacks = get_callback_funs(FileName),
      %% apply the refactoring command to the source.
    Res = apply(refac_rename_fun, rename_fun, C),
    case Res of
        %% ChangeFiles contains the names of those files
        %% that have been affected by this refactoring.
      {ok, ChangedFiles} ->  %% refactoring completed successfully.
        B2 = fun_binding_structure(FileName), %% new binding structure.
          %% get the name of the callback functions if there is any.
        CallBacks1 = get_callback_funs(FileName),
        C1 = [FileName,NewName, Arity, FunName, SearchPaths],
          %% rename the function back to its original name.
          %% we cannot use location as it might have been changed.
        {ok, ChangedFiles1} = apply(refac_rename_fun, rename_fun_1, C1),
          %% property1: renaming in both directions affect the same set of files.
        prop1 = ChangedFiles == ChangedFiles1,
          %% property2: rename twice should returns to the original file.
        Prop2 = pretty_print(FileName) == pretty_print("temp.erl"),
          %% property 3: B1 and B2 are isomorphic.
          %% rename/3 replaces Mod, FunName, Arity with Mod, NewName, Arity in B1
        Prop3 = B2== rename(B1, {Mod, FunName, Arity}, {Mod, NewName, Arity}),
          %% property 4: the same set of callback functions.
        Prop4 = CallBacks == CallBacks1,
          %% recover the original program for the next refactoring command.
        file:copy("temp.erl", FileName),
        Prop1 and Prop2 and Prop3 and Prop4;
      {error,ErrorMsg} ->  %% refactoring failed with an error message.
          %% carry out the transformation even though the side-conditions
          %% do not held; do_rename_fun/4 transforms the program.
       _Res = apply(refac_rename_fun, do_rename_fun, C),
       case ErrorMsg of
         {1, _R1} ->     %% failed for side-condition 1;
                       %% the transformed program should not compile.
                    file:copy("temp.erl", FileName),
                    {error, _Reason} = get_AST(FileName), true;
         {2, _R2} ->     %% failed for side-condition 2;
                    file:copy("temp.erl", FileName),
                    {error, _Reason} = get_AST(FileName), true;
         {3, _R3} ->     %% failed for side-condition 3;
                       %% the transformed program should compile, but the new
                       %% binding structure is not isomorphic to the original one.
                    {ok, _AST} = get_AST(FileName),
                    B2 = fun_binding_structure(FileName),
                    file:copy("temp.erl", FileName),
                    B2 /= rename(B1, {Mod,FunName, Arity},{Mod, NewName, Arity})
       end end end),
    qc:quickcheck(F).
```

Fig. 3. The top-level function for testing *renaming a function*

of the program when the *generalisation* refactoring is applied, and was detected by a property we wrote for this refactoring, which states that *generalisation* and *specialisation* are inverse; the other three bugs were all caught by the very basic properties, for example, one bug caused the refactoring engine to crash because of an unmatched case clause; and another caused the refactored code fail to compile because of the improper handling of generalisation on operators.

From our experience so far, the advantages of the QuickCheck approach are as follows:

- We are able to make the development of refactorings and their testing very closely integrated. The meaning of each refactoring was further clarified by the mutual testing of the implementation and the specification.
- Once properties have been written, many different test cases can be run with very little effort, instead of repeating the same set of test cases every time. As any Erlang program can serve as the test program, we can run the testing on as many test programs, especially large programs, as possible.
- Because of the controlled random generation of refactoring commands, and the large amount of tests we can run, more refactoring scenarios will be covered, therefore increasing the possibility of finding more bugs. At this point, one might think of the exhaustive testing of refactorings. While it is possible to enumerate all the possible refactoring commands when the input program is very small, it is not practical with large input programs due to the huge amount of refactoring commands that could be generated.
- This approach scales well to complex refactorings or composite refactorings. Testing of a complex refactoring does not necessitate the specification of very complex properties. Instead, we could write a collection of simple properties, each of which only tests one aspect of the refactoring. A composite refactorings can usually be decomposed into a series of basic refactorings, and each of these basic refactorings can be tested separately using this approach. This also corresponds naturally to the implementation of composite refactorings

While properties can be written separately from the implementation of refactorings, these properties normally make use of the infrastructure on which the refactorings are built, therefore familiarity with the infrastructure is essential for the testing using this approach.

8 Related Work

A number of case studies regarding to the use of Quviq QuickCheck or its predecessor as the test engine have been done and reported, among which one to test an industrial implementation of the Megaco protocol, and faults that have not been detected by other testing techniques were found [2]. This case study also shows the power of shrinking provided by Quviq QuickCheck, and one example is that a test case consisting of a sequence of 160 commands was reduced to just seven. Shrinking of refactoring commands does not make the counterexample any simpler, therefore plays little role in this case study.

The most closely related work on the automated testing of refactorings is the approach of Daniel *et. al.* [5]. The core of this approach is ASTGen, a library for generating abstract syntax trees (ASTs) for Java programs. ASTGen allows the developer to write *imperative generators* whose executions produce abstract syntax trees (ASTs) for refactoring engines. To test a refactoring, a developer writes a generator whose execution produces thousands of programs with structural properties that are relevant for the specific refactoring being tested. Several kinds of properties (oracles) have also been created to automatically check that the refactoring engine transformed the generated program correctly. Compared with this approach, our approach is more lightweight, however a developer does need to make sure that the testing code base covers enough structure features and refactoring scenarios for the refactoring under testing.

9 Conclusion

Refactoring tools ought to allow program developers to quickly and safely refactor their program, especially large programs. However, a robust and safe refactoring tool is hard to develop, and most refactoring tools still contain bugs even after extensive testing. While unit testing does help to find bugs in refactoring tools, it is tedious to manually write test programs, and the coverage of the test cases is hard to guarantee, and it is even harder to test refactoring tools on large systems.

We have explored the idea of using Quviq QuickCheck to automate the testing of refactorings. In this approach, the correctness of refactorings is tested against specifications written in Erlang. Once a test program has been chosen, we automated the generation of refactoring commands and the checking of refactoring outputs. Within a short time, a number of bugs were found in the first release of Wrangler using this approach. The pros and cons of this approach is summarised in section 7.

We envisage exploring a number of further ideas for automated testing of refactorings using QuickCheck.

- It would be also interesting to generate Erlang programs to be refactored to see whether more combinations of Erlang constructions that provoke faults in Wrangler can be found.
- One of the options followed by Daniel *et. al.* in [5] is to compare the effect of two refactoring engines, namely Eclipse and NetBeans for Java. We will explore this option for Wrangler and the refactoring engine built by the group at Eötvös Loránd University, Budapest [12].
- We have not addressed the behaviour checking of programs; it would nevertheless be possible to extend our work to check the results of refactorings against their original version using randomly-generated input values.
- We have assumed the correctness of our infrastructure library; it would be instructive to express and then to test crucial properties of the functions in this library.

We also intend to provide an API to help the specification of properties in the context of refactorings, and we would also like to adopt this approach to test our Haskell refactoring tool, HaRe.

References

1. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
2. Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing Telecoms Software with Quviq QuickCheck. In Phil Trinder, editor, *Proceedings of the Fifth ACM SIGPLAN Erlang Workshop*. ACM Press, 2006.
3. Richard Carlsson and Mickaël Rémond. Eunit: a lightweight unit testing framework for erlang. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 1–1, New York, NY, USA, 2006. ACM Press.
4. Koen Claessen and John Hughes. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.
5. Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *ESEC/FSE 2007: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, New York, NY, USA, September 2007. ACM Press.
6. FORSE. Formally-Based Tool Support for Erlang Development. `http://www.cs.kent.ac.uk/projects/forse/`.
7. Martin Fowler. Refactoring Home Page. `http://www.refactoring.com`.
8. John Hughes. QuickCheck Testing for Fun and Profit. In *Ninth International Symposium on Practical Aspects of Declarative Languages(PADL 07)*, 2007.
9. Huiqing Li. *Refactoring Haskell Programs*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK, September 2006.
10. Huiqing Li, Claus Reinke, and Simon Thompson. Tool Support for Refactoring Functional Programs. In Johan Jeuring, editor, *ACM SIGPLAN Haskell Workshop, Uppsala, Sweden*, August 2003.
11. Huiqing Li and Simon Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs. In Massimiliano Di Penta and Leon Moonen, editors, *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 197–206. IEEE, September 2006.
12. Huiqing Li, Simon Thompson, László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Víg, and Tamás Nagy. Refactoring Erlang Programs. In *The Proceedings of 12th International Erlang/OTP User Conference*, Stockholm, Sweden, November 2006.
13. Luke Gorrie. Distel: Distributed Emacs Lisp (for Erlang). In Eighth International Erlang/OTP User Conference.
14. Richard Carlsson. Erlang Syntax Tools. `http://www.erlang.org/doc/doc-5.4.12/lib/syntax_tools-1.4.3`.
15. Nik Sultana. Verification of Refactorings in Isabelle/HOL. Master's thesis, Computing Laboratory, University of Kent, UK, September 2007.