Testing-framework-aware Refactoring

Huiqing Li
School of Computing, University of Kent, UK
H.Li@kent.ac.uk

Simon Thompson
School of Computing, University of Kent, UK
S.J.Thompson@kent.ac.uk

Abstract

Testing is the predominant way of establishing evidence that a program meets it requirements. When both test code and the application under test are written in the same programming language, a refactoring tool for this language should be able to refactor both application code and testing code together. However, testing frameworks normally come with particular programming idioms, such as their use of naming conventions, coding patterns, meta-programming techniques and the like. A refactoring tool needs to be aware of those programming idioms in order to refactor test code properly. Meanwhile the particularities of test code also suggest refactorings that are particularly applicable to test code.

In this paper we present our experience of extending Wrangler, a refactoring tool for the Erlang programming language, so as to handle the three common testing frameworks for Erlang, as well as discussing the refactoring of test code in its own right.

Categories and Subject Descriptors D.2.3 [*SOFTWARE ENGINEERING*]: Coding Tools and Techniques; D.2.6 []: Programming Environments; D.2.7 []: Distribution, Maintenance, and Enhancement; D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Applicative (functional) languages; Concurrent, distributed, and parallel languages; D.3.4 []: Processors

General Terms Languages, Design

Keywords Erlang, refactoring, testing, Wrangler, program analysis, program transformation, testing framework.

1. Introduction

While it is not possible to prove that a program is correct by testing, this is still the predominant way of establishing evidence that a program meets its requirements. Various testing methodologies are intended to make the search for faults as thorough as possible, and in addition there are a number of general frameworks designed to make the testing process both efficient and effective. For Erlang the available systems include those which are based on writing test suites and others which are random and property-based.

The most commonly used testing tools for Erlang include the OTP Test Server and Common Test (1; 2), EUnit (4), and QuickCheck (11; 14). Test Server, Common Test and EUnit are based on writing test suites, though EUnit is designed to support unit tests, in contrast to the system-level testing provided by Test Server and Common Test. QuickCheck allows users to express properties required of a system; these properties are checked by running the system with randomly generated test data.

From the refactoring point of view, the common aspect of each of these three frameworks is that the testing code is itself Erlang program text. So, an Erlang refactoring tool should be able to refactor code written under these testing frameworks. However, this cannot be achieved without addressing the particular idioms of these systems, such as their use of naming conventions, callback functions, meta-programming and the like. This mainly affects those refactorings that change the interfaces of functions and modules.

Test code is code - albeit of a particular kind. Test code can be refactored in its own right. Apart from those general refactorings, most testing frameworks also suggest a set of testing-framework-specific refactorings.

Wrangler (5; 6; 7; 8) is a refactoring tool which supports interactive refactoring for Erlang programs. It is integrated with both Emacs and Eclipse (through the ErlIDE plugin (3)). Wrangler supports a variety of refactorings, as well as a set of "code smell" inspection functions, and facilities to detect and eliminate code clones (9).

In the first releases of Wrangler testing frameworks were not taken into account. Therefore when a program containing test code was under refactoring, things could easily go wrong without even a warning. For example, in EUnit functions with arity zero and names ending in `_test_` represent test generator functions, which represent a collection of EUnit tests; carelessly renaming a function whose name ends in `_test_()` to one with another suffix would break the test code. In order to support consistent refactoring of appli-

3rd Workshop on Refactoring Tools '09 Oct. 26, 2009, Orlando, FL

cation code and test code, Wrangler now takes into account the conventions of the testing frameworks discussed above. We are also in the process of extending Wrangler to support refactorings that are specific to these testing frameworks.

In the remainder of this paper we first survey the three testing frameworks for Erlang in Section 2, and then give a short overview of Wrangler in Section 3. Section 4 explains the extension of Wrangler to support consistent refactoring of test code when application code is refactored, and testing-framework-specific refactorings are discussed in Section 5. Finally, conclusions are drawn in Section 6.

2. Testing Tools for Erlang

In this section, we give a short overview of the three systems supporting testing for Erlang, namely the Erlang/OTP Test Server and Common Test, EUnit and QuickCheck.

2.1 Erlang/OTP Test Server and Common Test

In Erlang/OTP Test Server and Common Test (1; 2), a test suite is an Erlang module that contains test cases. A test suite module normally has a name of the form `*_SUITE.erl` (where `*` is used to denote the remainder of the name). A collection of callback functions must be implemented in each test suite module. A test suite consists of a number of test cases, and test case is written as an Erlang function using a special coding pattern: each test case has generally three parts, describing the documentation, specification and execution of the test.

These parts are implemented as three clauses of the same function. The *documentation* clause matches the argument atom `doc` and returns a list of strings describing the test case. The *specification* clause matches the argument `suite` and returns the test specification for the test case. The *execution* clause implements the actual test case. It takes one argument, `Config`, which contains configuration information. These coding patterns should be observed by Erlang refactorers.

The result of the test is a set of HTML pages which present the total result, the suite result, log information and where in the test case source file the test failed (if it did). This framework also has a test specification mechanism to set out which test suites and test cases are (not) to be run.

2.2 EUnit

EUnit (4) is a lightweight unit testing framework for Erlang. A unit is a "well defined" component, such as a function, a module or an application. Within EUnit the tester adds test functions or test generating functions to a module, including the `eunit.hrl` header file.

A test function name should be of the form `*_test`, and a test generating function name on the form of `*_test_`. Symbolic representation of test data is be used by test generating functions to generate test objects. For example, the tuple `{generator, ModuleName, FunctionName}` is used to represent the test objects generated by calling the function `ModuleName:FunctionName()`, where `ModuleName` and `FunctionName` represent module and function names.

A collection of predefined macros are provided to abbreviate the test code. Test code can coexist with the application code in the same module, but it is also possible to put test code into a separate module. EUnit assumes that a module named `m_tests` represents the test module for module `m`.

2.3 QuickCheck

QuickCheck (11; 14) is a property-based testing tool for Erlang. Programs are tested by writing properties (using Erlang syntax) and test case generators in the source code. QuickCheck tests these properties with automatically generated test cases and examines whether the system under test satisfies them. By default, any function with arity zero whose name begins `prop_` is assumed to be a property.

When using QuickCheck to test, it is important to separate the testing of *pure* and *impure* functions. An impure function can modify the global state of the system, while a pure function will not. Impure operations are tested using an Abstract State Machine (ASM). ASM test cases are lists of symbolic commands, each of which binds a symbolic variable to the result of a symbolic function call. For example, `{set, {var, 1}, {call, erlang, whereis, [a]}}` is a command to set variable `1` to the result of calling function application `erlang:whereis(a)`. The use of an ASM also requires the tester to implement a set of callback functions of the ASM.

3. An Overview of Wrangler

Wrangler (5; 6; 7; 8) is a refactoring tool which supports interactive refactoring for Erlang programs. It is integrated with both Emacs and Eclipse. Wrangler supports a variety of refactorings: *Rename variable/module/function/process*, *Generalise function definition*, *Move function from a module to another*, *Function/Macro extraction*, *Fold expressions against function/macro*, *Tuple function parameters*, etc.

Apart from refactorings, Wrangler also provides functionalities for "bad smell" detection and semantics-aware expression/variable search. Among others, Wrangler's identical/similar code detection is able to detect identical/similar code fragments across multiple modules. The concept of "similarity" here is based on the idea of *least-general common abstraction*, which is also known as *anti-unification* (12; 15). Two expression/expression sequences are similar if their *least-general common abstraction* satisfy some similarity score, which is customizable by the user. More details of the clone detection algorithm and the refactoring used to eliminate clones are given in (9).

4. Extension of Existing Refactorings

Since each of the three testing frameworks uses Erlang as the programming language for writing test code, much of the extension is achieved by existing functionality; the rest

needs to address the particular idioms of the frameworks. In general this extension affects all the refactorings that change function and module interfaces.

With the current version of Wrangler (0.8), refactorings affected by this extension include *Rename function/module*, *Generalise function definition*, *Function extraction*, *Tuple function arguments*, *Move function definition to another module*. Instead of discussing the extension of each refactoring, we give a summary of the different aspects that needed to be addressed during the extension.

4.1 The Testing Framework(s) Used

Checking which testing frameworks are used by the program under refactoring is trivial, since each testing framework requires a different header file to be included in the program.

4.2 Naming Conventions

When a naming convention is enforced by a testing framework, the refactorer must ensure that this naming convention is observed. For example, when EUnit is used, renaming of a function ending in `test()` or `_test_()` to a name with a different ending (or *vice versa*) should generate a warning message; renaming the module `m` to some other name should also check whether there is a test module named `m_tests`, and if so, the test module should also be renamed.

4.3 Callback Functions

Both Erlang/OTP Test Server and QuickCheck abstract state machines require the tester to implement certain callback functions. A callback function has a specified function interface that governs both the function name and the parameters accepted by the function. A refactorer should be aware of those callback functions, and always warn the user when the refactoring to be applied would turn a callback function into non-callback function (or *vice versa*).

4.4 Meta-programming

Each of the testing frameworks uses *meta-programming* to some extent. For example, symbolic function calls of the form `{call, ModuleName, FunctionName, Args}` are used by QuickCheck abstract state machines, and EUnit makes use of symbolic test data representation as mentioned earlier. Note that meta-programming is not restricted to testing frameworks, and the same format of symbolic function call as used by QuickCheck could also be used by normal application code in Erlang; however inferring whether meta-programming is used by normal Erlang applications need deep data-flow analysis, and is not fully supported yet.

Take `{call, ModuleName, FunctionName, Args}` as an example: ideally a refactorer should be able to modify the module name, function name or the argument list whenever the module name, or the function interface referred to is modified by a refactoring. However, given the fact that in Erlang atoms have multiple roles – syntactically module name, function name, process name are all atoms – and an atom could also act as a literal, it is also possible that the same tuple format is used to mean different things in different contexts.

Given this uncertainty, Wrangler takes a rather conservative approach. For refactorings like renaming, when Wrangler cannot infer whether an atom represents the module/function name to be named from syntactic information, it will check the context in which the atom is used. If the context indicates a high probability that the atom represents the module/function name to be renamed, it will rename it; otherwise leave it unchanged. In both cases a warning message asking for the manual inspection from the user is given.

For refactorings that change the parameter of a function, Wrangler will try to keep the original function interface in the program, although its function body will be replaced with an application of the new function introduced. This is possible due to the fact the Erlang allows the same function name to be re-defined with a different arity.

4.5 Macros

Refactoring programs containing macros is generally supported by Wrangler, but early releases of Wrangler did not look into the actual definition of macros, and this turned to be a problem when refactoring QuickCheck code where macros are used very heavily.

In fact most of the QuickCheck library functions for writing properties are provided via macros; consider the example of the `FORALL` macro in

```
prop_gcd()->
  ?FORALL(X, nat(),
     ?FORALL(Y, nat(),
        ex:gcd(#rec{num1=X,num2=X*Y) == X)).
```

where it is used to represent universal quantification in way that allows tests to be generated for the property.

The heavy use of macros and the complexity of macro definitions make it sometimes impossible to resolve the binding structure of variables without looking into the actual macro definitions as the example function above shows. The study of QuickCheck has led us to improve the way in which macros are handled in Wrangler. Two kinds of ASTs are kept during the refactoring process, one with all macro applications expanded and one with macro expansion bypassed. The former is used to infer the accurate binding structure of variables, which is then passed on to the latter.

4.6 Coding Patterns

The Erlang/OTP Test Server and Common Test framework ask testers to write test cases following a special coding pattern. For example, a test case generally takes one parameter, has three function clauses representing the documentation part, the specification part and the execution part, and each function clause takes a specific pattern to match. In this context the refactorer should make sure this coding pattern is not violated during the refactoring process.

Take the *Generalisation* refactoring as an example. This refactoring generalises a function definition over an expression in the function body by adding a new parameter to the function, replacing the expression selected with the new variable, and adding the expression as a new parameter to all the call-sites of this function. This refactoring certainly changes the function interface, and generalisation of a test case function will make it no longer a test case. With Wrangler, this kind of violation again is avoided by keeping the original function in the program, but its function body will become an application instance of the new function.

5. Testing-framework-specific Refactorings

To make Wrangler testing-framework-aware, we aim to make Wrangler not only be able to refactor application code and test code consistently, but also be able to support testing-framework-specific refactorings. Our study of the three testing frameworks shows that different refactorings will supplement the different testing frameworks.

5.1 Erlang/OTP Test Server and Common Test

Test code written under the Test Server and Common Test framework has a rather constrained top-level structure because of the coding pattern followed; however, our case studies show that most test cases have very similar structure, and the *copy*, *paste*, then *modify* style of editing is very heavily used, which results in substantial amount of duplicated code.

Tool support for duplicated code detection and elimination would help to provide better abstraction of some repeatedly used functionalities, improve the code structure, reduce the size of the code, and make it much easier to understood. Together with our project partner from Ericsson, Sweden, we have used Wrangler's support for duplicated code detection and elimination (9); the results are discussed in (10).

5.2 EUnit

EUnit code could also be helped by refactoring. Some of the refactorings to be added to Wrangler are:

- Convert tests written in plain Erlang into EUnit tests.
- Group isolated EUnit tests into a single test generator.
- Move EUnit tests in an application module to a separate test module.
- Normalise EUnit tests to a standard pattern.
- Extract common setup and tear-down code into *fixtures*.

5.3 QuickCheck

With QuickCheck, our research has been focused on refactorings that create properties, and refactorings that change the structure of existing properties. For example using the techniques of similar code detection and elimination, it is possible to turn a set of common test cases into a number of calls to a single function. From these calls it is possible to extract a QuickCheck property by the following steps:

- First identify all the calls to a particular function, and extract their arguments into a list of tuples, each tuple representing one call to the function.
- The tests can then be turned into a property by choosing `oneof` the list of test data, with the tuple chosen becoming the arguments to the call of the test function. Here `oneof` is a simple example of a QuickCheck *generator*.

6. Conclusions

We have presented our experience of extending Wrangler to accommodate testing framework programming idioms, and discussed our ongoing work to support testing-framework-specific refactorings. While this research focus on Erlang and Wrangler, the same problem should apply to other programming language domain and refactoring tools as well.

This research is supported by EU FP7 Collaborative project ProTest (13), grant number 215868; we thank our funders and colleagues for their support and collaboration.

References

[1] Erlang OTP/TestServer documentation page. `http://www.erlang.org/project/test_server/html/index.html`

[2] Erlang Common Test documentation page. `http://www.erlang.org/doc/apps/common_test/index.html`

[3] ErlIDE home page `http://erlide.sourceforge.net/`

[4] EUnit documentation. `http://svn.process-one.net/contribs/trunk/eunit/doc/overview-summary.html`

[5] H. Li, S. Thompson: *Tool Support For Refactoring Functional Programs.* In Partial Evaluation and Program Manipulation (PEPM'08). San Francisco, California, USA (2008).

[6] H. Li, S. Thompson: *A Comparative Study of Refactoring Haskell and Erlang Programs.* In Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006

[7] H. Li, S. Thompson: *Testing Erlang Refactorings with QuickCheck.* In IFL2007, Freiburg, Germany, 2007.

[8] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, T. Nagy: *Refactoring Erlang programs.* In: The Proceedings of 12th International Erlang/OTP User Conference, 2006.

[9] H. Li, S. Thompson: *Similar Code Detection and Elimination for Erlang Programs.* Twelfth International Symposium on Practical Aspects of Declarative Languages, Jan. 2010. (to appear)

[10] Li, H., Lindberg, A., Schumacher, A., Thompson, S.: *Improving Your Test Code with Wrangler.* Technical Report 4-09, School of Computing, Univ. of Kent, UK

[11] Open source Erlang QuickCheck home page. `http://www.cs.chalmers.se/~rjmh/ErlangQC/`

[12] G. D. Plotkin: *A Notes on Inductive Generalization.* Machine Intelligence. 5:153-163,1970.

[13] ProTest. `http://www.protest-project.eu/`

[14] QuviQ QuickCheck homepage. `http://www.quviq.com/`

[15] J. C. Reynolds: *Transformational Systems and the Algebraic Structure of Atomic Formulas.* Machine Intelligence. 5,1970.