Improved Testing through Refactoring: Experience from the ProTest Project

Huiqing Li and Simon Thompson

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

Abstract. We report on how the Wrangler refactoring tool has been used to improve and transform test code for Erlang systems. This has been achieved through the removal of code clones, the identification of properties for property-based testing and the definition of testing-aware refactorings and test-framework-specific refactorings. While some of the observations are Erlang-specific, others apply to test code in general.

We argue that refactoring is particularly valuable within testing. The Wrangler tool for Erlang provides support for clone detection and removal, and this has been used effectively both for clarifying test code and in extracting higher-level properties from test suites. We also report on refactorings within particular test frameworks, and on the constraints on refactorings that test frameworks impose.

1 Background

ProTest [1], EU FP7 grant 215868, concerns property-based testing. The interactions of refactoring and testing are the responsibility of the Kent team.

Erlang [2] is a strict, impure, dynamically typed functional programming language with support for higher-order functions, pattern matching, concurrency, communication, distribution, fault-tolerance, and dynamic code loading. There are a number of test frameworks for Erlang, including EUnit [3] and Common Test [4] as well as the property-based, random testing tool, Quviq QuickCheck [5]. Each of these tools imposes a number of constraints on the way that test code is written including naming, macro definitions and callback functions.

Wrangler [6] is an interactive tool for refactoring Erlang programs. Wrangler is integrated with Emacs and also with Eclipse through ErlIDE. Wrangler supports a collection of refactorings, including renaming, function extraction and generalisation, folding and inlining definitions and moving definitions between modules. All refactorings work across multiple-module projects. Wrangler also has a set of "code smell" inspection functionalities as well as facilities to detect and eliminate code clones and to improve the module structure of projects.

Wrangler 'similar' code detection is based on the notion of *anti-unification* [7] to detect code clones in Erlang programs; it also has a mechanism for automatic clone elimination under the user's control. The *anti-unifier* of two terms

L. Bottaci and G. Fraser (Eds.): TAIC PART 2010, LNCS 6303, pp. 198–202, 2010.

denotes their *least-general common abstraction*, and we say that two expression/expression sequences, `A` and `B`, are *similar* if there exists a 'non-trivial' least-general common abstraction, `C`, and two substitutions which take `C` to `A` and `B` respectively. By 'non-trivial' we mean that the size of the least-general common abstraction should satisfy some threshold relative to the clone instances. To eliminate a clone, we define a function whose body is the anti-unifier: each instance is given by transforming the substitution into the actual parameter list. More details of the implementation are in [8].

2 Improving Test Code by Clone Elimination

We have been able to use *clone detection and elimination* to make test files simultaneously more concise and more comprehensible. A case study with Ericsson SW Research is reported in detail in [9]: we review the conclusions here.

The case study examined the test code for part of an Erlang implementation of the SIP (Session Initiation Protocol). Working with engineers Adam Lindberg and Andreas Schumacher from Ericsson and Erlang Solutions Ltd. we reduced the code from 2658 lines of code to 2042 in a series of twelve clone eliminations. At the conclusion further clone eliminations and other refactorings were possible. The basic pattern of refactoring was to repeatedly perform these steps.

- Run the clone detection report.
- Choose a clone for removal.
- Introduce the common generalisation as a new function definition.
- Rename the function and the parameters.
- Eliminate the instances by 'folding' them against the function definition.

At first sight this appears to be an entirely mechanical process, but the case study showed that it was quite the reverse: we look at the reasons now.

Which of the reported clones? The first clone detection reports 31 clones, from one of 86 lines, repeated once, to one of 6 lines, appearing 15 times. Which should be chosen? Our experience was that it was best to choose a smaller clone (repeated many times) rather than a larger one. The reason for this *bottom up* approach was that it was substantially easier to understand and therefore to name the former rather than the latter.

Include all the reported code? The clone report has certain threshold parameters: e.g. the default search is for clones of at least five statements. We found cases where a clone contains *sub-clones* which had not been reported because of thresholding, but where we chose to eliminate the sub-clone (first).

What are their names? We were working with engineers who were familiar with the SIP environment, even if they had not written the tests themselves. Only with their insights were we able to give appropriate names to the new functions and their formal parameters.

When and how to generalise? How general should code be? We saw situations where we chose not to further generalise code even when that was

possible, since we would prefer to keep separate versions of similar code with different names. On the other hand, we encountered situations where we had generalised prematurely, and so we had to inline our generalisations in order to make a more general clone elimination. In both cases it was a matter of "programmers' taste" as to when to (stop) generalising.

'Widows' and 'orphans'. We saw cases where the detected clone contained a clear piece of functionality, but by accident, as it were, an extra line appeared at the start or end, because this functionality was usually preceded or succeeded by a particular action. Despite this, we chose to eliminate the meaningful clone, rather than including the maximum number of lines, since naming the former would be possible while naming the latter would not.

Taken together, these reasons show that this cannot be a "push button" automatic operation, but rather it can only be accomplished in collaboration with domain experts. On the other hand, we found that without the automated facilities of Wrangler for finding candidates to eliminate, and generalisations to eliminate them, it would have been impossible to perform an exercise like this.

3 Extracting Properties from Tests

QuickCheck testing involves the statement of properties, which are typically universally quantified, and checking whether they hold of randomly generated values. One of the aims of the ProTest project is to extract properties from tests, and the latest release of Wrangler provides one mechanism for this, thus:

- Identify a clone which consists of a complete test case, with the function `body` giving the common generalisation.
- Replace clone instances by function calls: `body(args1)`, `body(args2)` etc.
- Define a QuickCheck generator `args()` for `[args1, args2, ...]` and a universal property stating "`body(X)` holds for any value in `args()`".
- Generalise the generator to `arg_types()`, which generates arbitrary tuples of the argument types.
- Generalise the property to "`body(X)` holds for any value in `arg_types()`".

Future work will identify its applicability and limitations, as well as extensions.

4 Making Refactorings Testing Framework-Aware

The refactorings in Wrangler have been designed to respect the conventions of the testing frameworks for Erlang. This affects not only the transformations themselves but also the pre-conditions for their applications. This is reported in detail in [10]: we summarise the pertinent points here.

Naming Conventions. When a naming convention is enforced by a testing framework, the refactorer must ensure that this naming convention is observed. For example, the suffix `_test_()` has a particular significance in EUnit. Changing names in this way must therefore be checked.

Callback Functions. QuickCheck testing with state machines requires the tester to implement certain callback functions. A callback function has a specified function interface, and refactorings need to respect this.

Meta-programming. Each of the testing frameworks uses meta-programming. For example, symbolic function calls of the form {`call, Mod, Fun, Args`} are used by QuickCheck abstract state machines. Frameworks for other languages may well use reflection for similar purposes.

Macros. QuickCheck testing code uses macros very heavily: this can cause problems for semantic analysis, which is at the foundation of pre-condition checking for refactorings. Pre-processors (e.g. CPP) are used in other frameworks.

Evidently most of the points raised here are equally applicable to testing frameworks for other programming languages.

5 Refactoring within Test Frameworks

Recent releases of Wrangler support a number of refactorings for test code *within* the QuickCheck framework. These were developed together with the QuickCheck team to support particular use cases of the tool, and include:

- Refactorings to introduce local definitions (`?LET`) and to merge local definitions and quantifiers (`?FORALL`).
- In state-machine based testing it is usual to begin with the state machine data being a simple value, but during test development for it to become a record. We support this transformation.

The latter *data type* refactoring is possible because of the stylised way that the state data is handled in QuickCheck. It would be much more difficult to perform in general because of the dynamic type system of Erlang.

This work in progress is to be supplemented in due course with refactorings within EUnit and Common Test.

6 Conclusions

We have shown the various ways that refactoring tools can improve testing for Erlang programs, and reflected on aspects of each of these. While some points relate specifically to Erlang, the main arguments of the paper apply equally well to testing in other languages and frameworks.

References

1. ProTest project, `http://www.protest-project.eu/`
2. Cesarini, F., Thompson, S.: Erlang Programming. O'Reilly Media, Inc., Sebastopol (2009)
3. Carlsson, R., Rémond, M.: EUnit, `http://svn.process-one.net/contribs/trunk/eunit/doc/overview-summary.html`

4. Common Test, `http://www.erlang.org/doc/apps/common_test/index.html`
5. Arts, T., et al.: Testing Telecoms Software with Quviq QuickCheck. In: Proceedings of the Fifth ACM SIGPLAN Erlang Workshop. ACM Press, New York (2006)
6. Li, H., Thompson, S., Orosz, G., Töth, M.: Refactoring with Wrangler, updated. In: ACM SIGPLAN Erlang Workshop 2008, Victoria, British Columbia, Canada (2008)
7. Plotkin, G.D.: A note on inductive generalisation. Machine Intelligence 5 (1970)
8. Li, H., Thompson, S.: Similar Code Detection and Elimination for Erlang Programs. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 104–118. Springer, Heidelberg (2010)
9. Li, H., et al.: Improving your test code with Wrangler. Technical Report 4-09, School of Computing, University of Kent (2009)
10. Li, H., Thompson, S.: Testing-framework-aware Refactoring. In: Third ACM Workshop on Refactoring Tools, ACM Digital Library (2009) (to appear)