Improving your test code with Wrangler

Huiqing Li[1], Adam Lindberg[2], Andreas Schumacher[3] and Simon Thompson[1]

[1] School of Computing, University of Kent, UK
`{H.Li,S.J.Thompson}@kent.ac.uk`
[2] Erlang Training and Consulting Ltd.
`adam.lindberg@erlang-consulting.com`
[3] Ericsson SW Research, Ericsson AB
`andreas.schumacher@ericsson.com`

Abstract. In this paper we show how the 'similar code' detection facilities of Wrangler, combined with its portfolio of refactorings, allow test code to be shrunk dramatically, under the guidance of the test engineer. This is illustrated by a sequence of refactorings of a test suite taken from an Erlang project at Ericsson AB.

Key words: Erlang, similar code, refactoring, testing, clone detection, generalisation, strategies

1 Introduction

Wrangler [1,2] is a tool that supports interactive refactoring of Erlang programs[4]. It is integrated with Emacs and now also with Eclipse. Wrangler itself is implemented in Erlang. Wrangler supports a variety of refactorings, as well as a set of "code smell" inspection functionalities, and facilities to detect and eliminate code clones. In this paper we explore a case study of test code provided by Ericsson SW Research[5]

Why is test code particularly prone to clone proliferation? One reason is that many people touch the code: a first few tests are written, and then others author more tests. The quickest way to write these is to copy, paste and modify an existing test, even if this is not the best way to structure the code, it can be done with a minimal understanding of the code.This observation applies equally well to long-standing projects, particularly with a large element of legacy code.

What comes out very clearly from the case study is the fact that refactoring or clone detection cannot be completely automated. In a preliminary experiment by one of the Wrangler developers (Thompson) the code was reduced by some 20% using a "slash and burn" approach, simply eliminating clones one by one. The result of this was – unsurprisingly – completely unreadable. It is only with the collaboration of project engineers, Lindberg and Schumacher, that we were

[4] We acknowledge the support of the 7th Framework Programme of the European Commission for the ProTest[3] project to which the work reported here contributes.
[5] We are grateful to Ericsson SW Research for permission to include portions of the code in this paper.

able to identify which clone candidates should be removed, how they could be named and parameterised and so forth. Moreover it requires domain insight to decide on which clones might not be removed. These questions are discussed at length here.

The remainder of the paper is organised as follows. Section 2 describes the Wrangler refactoring tool, and in particular the support that it provides for clone detection and removal. Section 3 describes the case study itself, and Section 4 highlights some lessons coming from the case study (cross-referenced with the stages of the case study). We then describe future work and conclude the paper.

2 Clone detection and removal with Wrangler

Duplicated code, or the existence of code clones, is one of the well-known bad code smells when refactoring and software maintenance is concerned. 'Duplicated code', in general, refers to a program fragment that is identical or similar to another, though the exact meaning of 'similar' might vary slightly between different application contexts.

While some code clones might have a sound reason for their existence [4], most clones are considered harmful to the quality of software, as code duplication increases the probability of bug propagation, the size of both the source code and the executable, compile time, and more importantly the maintenance cost [5,6].

The most obvious reason for code duplication is the reuse of existing code (by *copy*, *paste* and *modify* for example), logic or design. Duplicated code introduced for this reason often indicates program design problems such as the lack of encapsulation or abstraction. This kind of design problem can be corrected by refactoring out the existing clones in a later stage [7,8,9], it could also be avoided by first refactoring the existing code to make it more reusable, then reuse it without duplicating the code [8]. In the last decade, substantial research effort has been put into the detection and removal of clones from software systems; however, few such tools are available for functional programs, and there is a particular lack of tools that are integrated with existing programming environments.

Wrangler Wrangler [1,2] is a tool built in the School of Computing at The University of Kent, with support from the European Commission. Wrangler supports interactive refactoring of Erlang programs. It is integrated with Emacs and now also with Eclipse. Wrangler itself is implemented in Erlang. Wrangler supports a variety of refactorings, as well as a set of "code smell" inspection functionalities, and facilities to detect and eliminate code clones.

Wrangler provides a 'similar' code detection approach based on the notion of *anti-unification* [10,11] to detecting code clones in Erlang programs, as well as a mechanism for automatic clone elimination under the user's control. The *anti-unifier* of two terms denotes their *least-general common abstraction*, therefore captures the common syntactical structure of the two terms.

In general, we say two expression/expression sequences, `A` and `B`, are *similar* if there exists a 'non-trivial' least-general common abstraction, `C`, and two sets of substitutions σ_1 and σ_2 which take `C` to `A` and `B` respectively. By 'non-trivial' we mean mainly that the size of the least-general common abstraction should satisfy some threshold, but certainly other conditions could be added.

This clone detection approach is able, for example, to spot that the two expressions `((X+3)+4)` and `(4+(5-(3*X)))` are similar as they are both instances of the expression `(Y+Z)`, and so both instances of the function

```
add(Y,Z) -> Y+Z.
```

Our approach uses the Abstract Syntax Tree (AST) annotated with static semantic information as the internal representation of Erlang programs. Scalability, one of the major challenges faced by AST-based clone detection approaches, is achieved by a two-phase clone detection technique. More details of the process can be found in [12].

Wrangler clone detection Wrangler provides facilities for finding both identical and similar code. Two pieces of code are said to be *identical* if they are the same when the values of literals and the names of variables are ignored, while the binding structures are the same. The Wrangler definition of *similarity* is given above. For both identical and similar code, two operations are possible:

Detection This operation will identify all code clones (up to identity or similarity) in a module or across a project. For each clone the common generalisation for the clone is generated in the report, and can be cut and pasted into the module prior to clone elimination.

Search This operation allows the identification of all the code that is identical or similar to a *particular selection*, so is directed rather than speculative.

Examples of both will be seen in the case study.

3 The Case Study

The case study examined part of an Erlang implementation of the SIP (Session Initiation Protocol) [13]. As a part of SIP message processing it is possible to transform messages by applying rewriting rules to messages. This SIP message manipulation (SMM) is tested by a test suite contained in the file `smm_SUITE.erl`, which is our subject here.

The size of the sequence of versions of the files – in lines of code – is indicated in Figure 1, which shows that the code has been reduced by about 25% through these transformations. As we discuss at the end of this section there is still considerable scope for clone detection and elimination, and this might well reduce the code by a further few hundred lines of code.

Version	LOC	Version	LOC	Version	LOC
1	2658	6	2218	10	2149
2	2342	7	2203	11	2131
3	2231	8	2201	12	2097
4	2217	9	2183	13	2042
5	2216				

Fig. 1: The size of the refactored files

The sequence of transformations

In this section we give an overview of the particular steps taken in refactoring the SMM test code.

Step 1 We begin by generating a report on similar code in the module. 31 clones are detected, with the most common on being cloned 15 times. The generalisation suggested in the report is

```
new_fun() ->
    SetResult = ?SMM_IMPORT_FILE_BASIC(?SMM_RULESET_FILE_1, no),
    ?TRIAL(ok, SetResult),
    %% AmountOfRuleSets should correspond to the amount of rule sets in File.
    AmountOfRuleSets = ?SMM_RULESET_FILE_1_COUNT,
    ?OM_CHECK(AmountOfRuleSets, ?MP_BS, ets, info, [sbgRuleSetTable, size]),
    ?OM_CHECK(AmountOfRuleSets, ?SGC_BS, ets, info, [smmRuleSet, size]),
    AmountOfRuleSets.
```

which shows that the code is literally *repeated* sixteen times. This function defintion can be cut and pasted into the test file, and all the clones folded against it. Of course, it needs to be renamed: we choose to call it `import_rule_set_file_1` since the role of this function is to import rule sets which determine the actions taken by the SMM processor, and this import is a part of the common setup in a number of different test cases. The function can be renamed when it is first introduced, or after folding against it, using the *Rename Function* refactoring.

Step 2 Looking again at similar code detection we find a twelve line code block that is repeated six times. This code creates two SMM filters, and returns a tuple containing names and keys for the two filters. This is a common pattern, under which the extracted clone returns a tuple of values, which are assigned to a tuple of variables on function invocation, thus:

```
{FilterKey1, FilterName1, FilterState, FilterKey2, FilterName2}
       = create_filter_12()
```

We have named the function `create_filter_12`; this reflects a general policy of not trying to anticipate general names for functions when they are introduced. Rather, we choose the most specific name, generalising it – or indeed the functionality itself — at a later stage if necessary, using Wrangler.

Step 3 At this step a 21 line clone is detected:

```
new_fun() ->
    {FilterKey1, FilterName1, FilterState, FilterKey2, FilterName2}
        = create_filter_12(),
    ?OM_CHECK([#smmFilter{key=FilterKey1,
              filterName=FilterName1,
              filterState=FilterState,
              module=undefined}],
          ?SGC_BS, ets, lookup, [smmFilter, FilterKey1]),
    ?OM_CHECK([#smmFilter{key=FilterKey2,
              filterName=FilterName2,
              filterState=FilterState,
              module=undefined}],
          ?SGC_BS, ets, lookup, [smmFilter, FilterKey2]),
    ?OM_CHECK([#sbgFilterTable{key=FilterKey1,
                   sbgFilterName=FilterName1,
                   sbgFilterState=FilterState}],
          ?MP_BS, ets, lookup, [sbgFilterTable, FilterKey1]),
    ?OM_CHECK([#sbgFilterTable{key=FilterKey2,
                   sbgFilterName=FilterName2,
                   sbgFilterState=FilterState}],
          ?MP_BS, ets, lookup, [sbgFilterTable, FilterKey2]),
    {FilterName2, FilterKey2, FilterKey1, FilterName1, FilterState}.
```

Inspecting this shows up a smaller clone, encapsulated in the suggested function

```
new_fun(FilterState, FilterKey2, FilterName2) ->
    ?OM_CHECK([#sbgFilterTable{key=FilterKey2,
                   sbgFilterName=FilterName2,
                   sbgFilterState=FilterState}],
          ?MP_BS, ets, lookup, [sbgFilterTable, FilterKey2]).
```

which we choose to replace first: as a general principle we found it more useful to replace clones *bottom up*. The clone was replaced by the function

```
check_filter_exists_in_sbgFilterTable(FilterKey, FilterName,FilterState) ->
    ?OM_CHECK([#sbgFilterTable{key=FilterKey,
                   sbgFilterName=FilterName,
                   sbgFilterState=FilterState}],
          ?MP_BS, ets, lookup, [sbgFilterTable, FilterKey]).
```

where as well as renaming the function and variable names, the order of the variables is changed. This can be done simply by editing the list of arguments, because before folding against the function there are no calls to it, since it is newly introduced.

Steps 4–5 Introduce two variants of `check_filter_exists_in_sbgFilterTable`:

– In the first the check is for the filter occurring only once in the table, so that a call to `ets:tab2list` replaces the earlier call to `ets:lookup`.
– In the second the call is to a different table, `sbgFilterTable` being replaced by `smmFilter`.

Arguably these three alternatives could have been abstracted into a common generalisation, but it was felt by the engineers that each of the three functions each encapsulated a meaningful activity, whereas a generalisation would have had an unwieldy number of parameters as well as being harder to name appropriately.

Step 6 Erlang provides two mechanisms for finding out whether the code for a module `M` is loaded:

```
    erlang:module_loaded(M) -> true | false
    code:is_loaded(M) -> {file, Loaded} | false
```

Use of the former is deprecated outside the code server, but both are used in this file. We want to *remove the deprecated calls*, all of which are symbolic calls in contexts like:

```
    ?OM_CHECK(false, ?SGC_BS, erlang, module_loaded, [FilterAtom1])
```

So what we do is to define a new function, in which we abstract over module name, the type of blade and the expected result of the call to `erlang:module_loaded`.

```
code_is_loaded(ModuleName, BS, Result) ->
    ?OM_CHECK(Result, BS, erlang, module_loaded, [ModuleName]).
```

We then fold against this definition to remove all calls to `erlang:module_loaded`, expect for that in the definition of `code_is_loaded` itself. We can then write a *different* definition of this function, which implements the same functionality using the other primitive:

```
code_is_loaded(BS, ModuleName, false) ->
    ?OM_CHECK(false, BS, code, is_loaded, [ModuleName]).
code_is_loaded(BS, ModuleName, true) ->
    ?OM_CHECK({file,atom_to_list(ModuleName)}, BS,
            code, is_loaded, [ModuleName]).
```

At this point it is possible to stop, having introduced the `code_is_loaded` function. Alternatively, in order to keep the code as close as possible to its previous version, we can *inline* this function definition. In the next step we will see another reason for doing this inlining.

Step 7 We note that as well as finding symbolic calls to `code:is_loaded` within the `OM_CHECK` macro call, it is also called within `CH_CHECK`. We are unable to replace a macro call by a variable, and so we write – by hand – a generalisation in which the macro call is determined by an atom parameter

```
code_is_loaded(BS, om, ModuleName, false) ->
      ?OM_CHECK(false, BS, code, is_loaded, [ModuleName]);
code_is_loaded(BS, om, ModuleName, true) ->
      ?OM_CHECK({file, atom_to_list(ModuleName)},
             BS, code, is_loaded, [ModuleName]);
code_is_loaded(BS, ch, ModuleName, false) ->
      ?CH_CHECK(false, BS, code, is_loaded, [ModuleName]);
code_is_loaded(BS, ch, ModuleName, true) ->
      ?CH_CHECK({file, atom_to_list(ModuleName)},
             BS, code, is_loaded, [ModuleName]).
```

It is here that inlining of the `code_is_loaded` function in step 6 is valuable: it allows us to deal with *premature generalisation*, under which we find that we want further to generalise a function without layering a number of intermediate calls: we inline the earlier generalisations and then build the more general function in a single step.

Steps 8,9 In this step a ten line clone was identified found, but rather than replacing that – which combines a number of operations – it was decided to look at sub-clones, and this indicated code used 22 times in the module, extracted as

```
check_add_rule_set_to_filter(FilterKey, FilterName, RuleSetName,
                 FilterRuleSetPosition, Result) ->
    AddResult =
        ?SMM_ADD_RULE_SET_TO_FILTER(FilterKey, FilterName,
                   RuleSetName, FilterRuleSetPosition),
    ?TRIAL(Result, AddResult).
```

This gives the ninth version of the code, and two similar sub-clones are extracted thus:

```
check_ruleset_name_in_filter(FilterName, RuleSetName) ->
    {ok, RuleSetKey} = ?SMM_NAME_TO_KEY(sbgRuleSetTable, RuleSetName),
    check_ruleset_key_in_filter(RuleSetKey, [[FilterName]]),
    RuleSetKey.

check_ruleset_key_in_filter(RuleSetKey, Result) ->
    ?OM_CHECK(Result,
          ?MP_BS, ets, match, [sbgIsmFilterRuleSetUsageTable,
                 {'_', {RuleSetKey, '_'}, '_', '$1'}]).
```

which gives the tenth version of the code.

Step 10 Clone detection now gives this clone candidate:

```
new_fun(FilterName, NewVar_1) ->
    FilterKey = ?SMM_CREATE_FILTER_CHECK(FilterName),
    %%Add rulests to filter
    RuleSetNameA = "a",
    RuleSetNameB = "b",
    RuleSetNameC = "c",
    RuleSetNameD = "d",
    ... 16 lines which handle the rules sets are elided ...
    %%Remove rulesets
    NewVar_1,
    {RuleSetNameA, RuleSetNameB, RuleSetNameC, RuleSetNameD, FilterKey}.
```

The main body of the function sets up four rule sets and adds them to a filter, but the function identified contains extraneous functionality at the start and end:

- the filter key is created as the first action: `FilterKey = ...`, and
- (in at least one of the clones) the rulesets are removed thus: `NewVar_1` .

Instead, the body is extracted thus:

```
add_four_rulesets_to_filter(FilterName, FilterKey) ->
    RuleSetNameA = "a",
    RuleSetNameB = "b",
    RuleSetNameC = "c",
    RuleSetNameD = "d",
    ... 16 lines which handle the rules sets are elided ...
    {RuleSetNameA, RuleSetNameB, RuleSetNameC, RuleSetNameD}.
```

in which the final action doesn't need to be passed in as a parameter, and the `FilterKey` becomes a parameter rather than a component of the result.

Step 11 In a similar way to the previous step, a lengthy clone is identified, but in the abstraction the first line is omitted, making it an abstraction without parameters: `setup_rulesets_and_filters`.

Step 12 This final step consists of a sequence of stages concerned with refactoring the form of data representation in dealing with

- sets of attributes, which are transformed into named lists, such as

  ```
  Attributes_1=[LineW1, ColumnW1, TypeAtom, FailingRuleSetW1, ResasonW1],
  ```

- other sets of attributes are represented by 0-ary functions:

  ```
  ruleset_error_attributes() -> [?sbgRuleSetErrorLineNumber, ..., ...
  ```

Finally, we can replace an explicit zipping together of lists by the uses of function `lists:zip/2`, and this gives us a much better structured function:

```
import_warning_rule_set(main) ->
    %% Since the rule set file contains errors, no rule sets will be imported.
    ...
    AttributeNames = [line_number, column, type, failing_rule_set, error_reason],
    Key1 = 1,
    ?ACTION("Do SNMP Get operations on the"
        "sbgRuleSetErrorTable with key ~p.~n", [Key1]),
    GetResult1 = ?SMM_SMF_GET(rule_set_error, [{key, Key1}], AttributeNames),
    LineW1 = 4,
    ColumnW1 = ?SMM_RULE_SET_WARN_1_COL,
    TypeAtom = warning,
    TypeMibVal = ?SMM_REPORT_TYPE(TypeAtom),
    FailingRuleSetW1 = " ",
    ResasonW1 = ?SMM_RULE_SET_WARN_1_REPORT(LineW1),
    Attributes_1 = [LineW1, ColumnW1, TypeAtom, FailingRuleSetW1, ResasonW1],
    ?TRIAL(lists:zip(AttributeNames, Attributes_1), GetResult1),
    ... 40 lines elided ... .
```

an incidental benefit of the inspection was to reveal that the lines

```
TypeAtom = warning,
TypeMibVal = ?SMM_REPORT_TYPE(TypeAtom),
```

were repeated, doubtless a cut-and-paste error, which went undetected because the pattern matches succeeded at the second occurrence.

Continuing the case study: further clone detection

The work reported here produced a sequence of twelve revisions of the code, but it is possible to make further revisions. In this section we look at a selection of reports from similar and identical code search, and comment on some of the potential clones identified.

Similar code The similar code detection facility with the default parameter values reports 16 further clones, each duplicating code once. The total number of duplicated lines here is 193, and so a reduction of some 145 lines could be made by replacing each clone into a function definition plus two function calls.

Looking for similar code with the similarity parameter reduced to 0.5 rather than the default of 0.8 reports 47 clones, almost three times as many. Of these, eight duplicate the code twice (that is, there are *three* instances of the code clone) and some of these provide potential clients for replacement. However, not all of them appear to be good candidates for replacement. Take the example of

```
/Users/simonthompson/Downloads/smm_SUITE13.erl:1755.4-1761.50:
This code has been cloned twice:
/Users/simonthompson/Downloads/smm_SUITE13.erl:1772.4-1778.50:
/Users/simonthompson/Downloads/smm_SUITE13.erl:1789.4-1795.50:

The cloned expression/function after generalisation:

new_fun(FilterAtom1, FilterAtom2, NewVar_1, NewVar_2, NewVar_3) ->
    NewVar_1,
    code_is_loaded(?SGC_BS, om, FilterAtom1, NewVar_2),
    code_is_loaded(?MP_BS, om, FilterAtom1, false),
    code_is_loaded(?SGC_BS, om, FilterAtom2, NewVar_3),
    code_is_loaded(?MP_BS, om, FilterAtom2, false).
```

This code has three parameters, and the first is an arbitrary expression, `NewVar_1`, evaluated first in the function body. A more appropriate candidate is given by omitting this expression, and giving the generalisation

```
new_fun(FilterAtom1, FilterAtom2, NewVar_1, NewVar_2) ->
    code_is_loaded(?SGC_BS, om, FilterAtom1, NewVar_1),
    code_is_loaded(?MP_BS, om, FilterAtom1, false),
    code_is_loaded(?SGC_BS, om, FilterAtom2, NewVar_2),
    code_is_loaded(?MP_BS, om, FilterAtom2, false).
```

This particular generalisation has a similarity score of more than 0.8, but does not appear in the default report because it involves only 4 expressions, and the default cut-off is a sequence of at least 5.

Identical code The standard report to detect identical code reports 87; the number is larger here because the default threshold for reporting here is to consist of at least 20 tokens rather than 5 expressions or more. However, a number of *over-generalisations* result from this report; for example, the function

```
new_fun(ModuleName1, NewVar_1, NewVar_2, NewVar_3, NewVar_4) ->
    code_is_loaded(?SGC_BS, NewVar_1, ModuleName1, NewVar_2),
    code_is_loaded(?SGC_BS, NewVar_3, ModuleName1, NewVar_4).
```

is reported as occurring 23 times. Arguably, generalising to replace these two expressions will not result in code that is more readable than it is already: it clearly states that it checks for two pieces of code being loaded.

Carrying on There are clearly some more clones that might be detected, but as the work progresses the effort involved in identifying and replacing code clones becomes more than the value of transforming the code in this way, and so the engineers performing the refactoring will need to decide when it is time to put the work aside.

4 Lessons learned

This section highlights the lessons learned during the activity reported in the previous section, cross-referring to the steps of that process when appropriate.

Inlining is a useful refactoring There is a clear use case for function *inlining* or *unfolding* when performing a series of refactorings based on clone elimination [Step 7]. The scenario is one of *premature generalisation* thus:

– identify common code, and generalise this, introducing a function for this generalisation;
– subsequently identify that there is a further generalisation of the original code, which could benefit from being generalised;
– the problem is that some of the original code disappeared in the first stage of generalisation, and so it needs to be inlined in order to generalise it further.

Of course, it would be possible to keep the intermediate generalisation as well as the final one, but in general that makes for less readable code, requiring the reader to understand two function definitions and interfaces rather than one.

Inlining is also useful to support a limited form of API migration [Step 6].

Bottom-up is better than top-down We looked at ways in which clones might be removed, and two approaches seem appropriate: bottom-up and top-down. In the latter case we would remove the largest clones first, while in the other approach we would look for small clones first, particularly those which are the most common. We decided to use the *bottom-up* approach for two reasons [Steps 3, 8, 9].

– Using this it is much easier to identify pieces of functionality which can easily be *named* because they have an identifiable purpose.
– it is also likely that these will not have a huge number of parameters, and in general we look for code which is not over-general.

Finally, there is the argument that – to a large extent, at least – it should not matter about the order in which clone removal takes place, since a large clone will remain after small clones are removed, and *vice versa*.

Clone removal cannot be fully automated What we have achieved in this example is clearly *semi-automated*: we have the Wrangler support for identifying candidates for clones but they may well need further analysis and insight from uses to identify what should be done. For example,

– Is there a spurious last action which belongs to the next part of the code, but which just happens to follow the clone when it is used? If so, it should not be included in the clone [Steps 10,11]. This can also apply to actions at the start of the identified code segment.

- Another related reason for this might be that this last operation adds another component to the return tuple of an extracted function; we should aim to keep these small.
- We might find that we identify behaviour of interest which occurs close to an identified clone; then use similar expression search to explore further.
- An identified clone may contain two pieces of separate functionality which are used together in many cases, but not in all cases of interest. Because of the thresholding of clone parameters it might be that we only see the larger clone because the smaller one is below the threshold chosen for the similar code report [Steps 3, 8, 9].

Self-documenting code Is it always useful to name values, as in

```
BlahMeaning = ... blah_exp ... ,
FooMeaning  = ... foo_exp ... ,
Result = f(BlahMeaning, FooMeaning),
```

rather than

```
Result = f(... blah_exp ... , ... foo_exp ...)
```

as the former case is *self documenting* in a way that the latter is not. On the other hand, is it the responsibility of the client of an API, here for the function `f`, to document this API at its calling points? If it is to be documented at a calling point it can be done by choice of variable names, or by suitably placed comments.

Naming values How should constants best be named in Erlang code. Three options are possible: namely definition by

- a macro definition
- a function of arity 0
- a local variable

The code at hand does the first and last: the second was added during the refactoring process.

Another option presents itself: instead of worrying about what is contained in a set of variables, should a `record` be used instead? This has some advantages, but records have drawbacks in Erlang. Names are not first class, so cannot be passed as parameters or values, which is something that can be used in practice, such as passing a list of field names to the `zip/2` function [Step 13].

Improvements to Wrangler The case study also brought to light a number of improvements that might be made to Wrangler. Inlining was the most important, and is now included in the latest release of the system.

It was also suggested that a number of options could be added to the *Code Inspector*: this highlights "bad smells" and other notable code features, such as:

variables used only once, variables not used, and rebinding of variables (that it, bound variables occurring in a pattern match).

Finally, the question was raised about how much refactoring sequences used on one module could be *reused* in refactoring another. This question of memoisation or scripting merits further work.

5 Conclusions and Future Work

As we have reported, the exercise here was only possible as a collaboration between the developers of Wrangler and engineers engaged in developing and testing the target system. Together it was possible substantially to re-engineer the test code to make it more compact and more structured. As well as illustrating the way in which Wrangler can be used, we were able to provide guidelines on refactoring test code in Erlang which can also be applied to systems written in other languages and paradigms.

Wrangler is under active development as a part of the ProTest project, and the insights gained here will feed into its further development.

References

1. Li, H., Thompson, S., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T.: Refactoring Erlang Programs. In: EUC'06, Stockholm, Sweden (November 2006)
2. Li, H., Thompson, S., Orosz, G., T"oth, M.: Refactoring with Wrangler, updated. In: ACM SIGPLAN Erlang Workshop 2008, Victoria, British Columbia, Canada
3. ProTest: Property based testing. `http://www.protest-project.eu/`
4. Kapser, C., Godfrey, M.W.: "Clones Considered Harmful" Considered Harmful. In: Proc. Working Conf. Reverse Engineering (WCRE). (2006)
5. Roy, C.H., Cordy, R.: A Survey on Software Clone Detection Research. Technical report, School of Computing, Queen's University at Kingston, Ontario, Candada
6. Monden, A., Nakae, D., Kamiya, T., Sato, S., Matsumoto, K.: Software Quality Analysis by Code Clones in Industrial Legacy Software. In: METRICS '02, Washington, DC, USA (2002)
7. Balazinska, M., Merlo, E., Dagenais, M., Lague, B., Kontogiannis, K.: Partial Redesign of Java Software Systems Based on Clone Analysis. In: Working Conference on Reverse Engineering. (1999) 326–336
8. M. Fowler: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
9. Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: ARIES: Refactoring Support Environment Based on Code Clone Analysis. In: IASTED Conf. on Software Engineering and Applications. (2004) 222–229
10. Plotkin, G.D.: A note on inductive generalization. Machine Intelligence **5** (1970) 153–163
11. Reynolds, J.C.: Transformational systems and the algebraic structure of atomic formulas. Machine Intelligence **5** (1970) 135–151
12. Li, H., Thompson, S.: Similar Code Detection and Elimination for Erlang Programs. In: PADL10. (2010) to appear.
13. SIP: Session Initiation Protocol. `http://tools.ietf.org/html/rfc3261`