Let's Make Refactoring Tools User-extensible!

Huiqing Li
School of Computing
H.Li@kent.ac.uk

Simon Thompson
School of Computing
S.J.Thompson@kent.ac.uk

Abstract

We present a framework for making a refactoring tool extensible, allowing users to define refactorings from scratch using the concrete syntax of the language, as well as to describe complex refactorings in a domain-specific language for scripting. We demonstrate the approach in practice through a series of examples.

The extension framework is built into Wrangler, a tool for refactoring Erlang programs, but we argue that the approach is equally applicable to tools for other languages.

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

General Terms Languages, Design

Keywords Erlang, refactoring, Wrangler, API, DSL, analysis, program transformation, extensible, concrete syntax.

1. Introduction

What do we mean when we say 'refactoring'? Interpreted narrowly, it is the process of invoking a refactoring tool to perform one of a fixed repertoire of transformations, whereas the term 'refactoring' is often also used for any process of transforming our programs so that they work in a different way, though preserving their original behaviour. A recent study by Murphy-Hill *et. al.* [14] reports that up to 90% of refactoring is done 'by hand' in practice.

Why such a low proportion of tool use? Tools can only support a limited collection of transformations, and these will be the common, 'vanilla flavoured' refactorings such as renaming, function or method extraction, data-type and class-based refactorings and so on. Users who want to do something more complex, which will typically also be more specific to their application, will have to implement their refactorings for themselves. This could simply mean doing the whole refactoring 'by hand' in an editor or IDE – which may well provide support for some of the steps – or it could involve the user extending an existing refactoring tool for their own purposes.

The latter option – to extend an existing tool such as the Java refactorer in Eclipse – is attractive in theory but seldom used in practice. The cost of understanding the internal representation of programs, the internal APIs and the overall architecture of the tool means that it's just not cost effective for the working programmer to take this option.[1]

In this paper we advocate designing refactoring tools for ease of extension, and report on the extensibility framework built into the Wrangler [11] refactoring tool for Erlang [2]. While this is a tool for a specific language, rather than a generic tool (a point we discuss in more detail in Section 7), the approach and the lessons learned are applicable to refactoring tools for any language. The guiding principle of the design is to improve the cost-benefit ratio, making it simple for a user to define powerful new refactorings with as little effort as possible; in particular:

- Users are able to specify transformations using the *concrete syntax* of Erlang [9], rather than some internal representation of a syntax tree (or an XML version of it). This means that users – who will understand how to write Erlang programs – can express transformations through *templates* and *rules* written in a familiar language rather than having to learn something new.
- Information about *static semantics* that forms the basis for most refactoring pre-conditions is accessible through a simple API.
- As the study [14] points out, some 40% of refactorings performed using a tool occur in batches. We provide a high-level *domain-specific language* for building complex refactorings from simpler components [10].
- We specify a *callback interface* so that user-defined refactorings can be invoked from within Wrangler, em-

WRT, June 01 2012, Rapperswil, Switzerland.

[1] The study [14] suggests that even when tools are configurable, this facility tends not to be used in practice, perhaps because a manual tweak to the default behaviour is easier to do than changing the configuration.

bedded in either Emacs or Eclipse. Doing this allows user-defined refactorings to be applied interactively, previewed and undone, just as for their built-in counterparts. This allows users to develop refactorings in an iterative, test-driven, style.

The main goal of this paper is to prove the usefulness of the framework through a series of examples in Section 6 that demonstrate the practicality of making refactoring tools user-extensible. Before that we briefly introduce Erlang and Wrangler in Section 2, and give a high-level overview of the extension framework: Section 3 covers the template- and rule-based framework for defining elementary Erlang refactorings, Section 4 the domain-specific language for describing complex refactorings, and Section 5 the callback interface (or Erlang `behaviour`) for refactorings. After the examples, Section 7 discusses related work and Section 8 draws some conclusions and addresses future work.

2. Erlang and Wrangler

Erlang [2] is a strict, impure, dynamically typed functional programming language with support for macros, higher-order functions, pattern matching, concurrency, distribution, fault-tolerance, and dynamic code loading.

An Erlang program consists of a number of modules, each of which defines a collection of functions. Only functions exported explicitly through the `export` directive may be called from other modules. In Erlang, a function name can be defined with different arities, and the same function name with different arities represent entirely different functions. Calls to functions defined in other modules generally qualify the function name with the module name: the function `F` from the module `M` is called thus: `M:F(...)`.

Wrangler [11], downloadable from `http://www.github.com/RefactoringTools`, is an open source tool that supports interactive refactoring, API migration and "code smell" detection for Erlang programs. Wrangler is implemented in Erlang, and is integrated with (X)Emacs as well as with Eclipse through the ErlIDE plugin. Wrangler supports a variety of built-in structural refactorings, as well as facilities to detect and eliminate duplicate code [8].

Wrangler uses an Abstract Syntax Tree (AST) to represent Erlang programs. The AST representation generated is designed so that all the nodes in the tree have a uniform structure; building on this we extend nodes with various annotations such as location, static semantic information, etc. We call the extended ASTs *annotated ASTs* (AAST). Wrangler preserves the original layout of the program as much as possible, and supports undo of refactorings and preview of refactoring results.

Wrangler, as an interactive refactoring tool, allows the user to perform what we term *selective* refactorings. By this we mean refactorings that involve a single local transformation, but may be applicable to multiple places across the project. An example of this kind of refactoring is to replace the use of `lists:map/2` with a list comprehension throughout a project. Selective refactoring allows the user to choose which candidates to refactor, and which not.

3. Composing Elementary Refactorings

In this section, we give an overview of how to write new refactorings from scratch in Wrangler.

3.1 Templates and Rules

The template- and rule-based API allows Erlang programmers to express program analysis and transformation in Erlang concrete syntax. In Wrangler, a code template is denoted by an Erlang macro `?T` whose only argument is the string representation of an Erlang code fragment that may contain meta-variables. A meta-variable is a placeholder for a syntax element in the program, or a sequence of syntax elements of the same kind. We also support meta-atoms, that only match atoms, which in Erlang represent function names and so forth.

Syntactically a meta-variable is an Erlang variable, ending with the character `'@'`. A meta-variable ending with a single `'@'` represents a single language element, and matches a single subtree in the AST; a meta-variable ending with `'@@'` is a list meta-variable that matches a sequence of elements of the same sort. For instance, the template

```
?T("erlang:spawn(Args@@, Arg1@)")
```

matches the applications of `spawn` function to one or more arguments, where `Arg1@` matches the last argument, and `Args@@` will match the remaining arguments, if any.

Templates are matched at AST level, that is, the template's AST is pattern matched to the program's AST using structural pattern matching techniques. If the pattern matching succeeds, the meta-variables/atoms in the template are bound to AST subtrees, and the context and static semantic information attached to the subtrees matched can be retrieved as described below in Section 3.2.

The template-based API is not only used to retrieve information about a program, but also to define transformation rules used in the transformation of a program. A rule defines a basic step in the transformation of a program; it involves recognising a program fragment to transform and constructing a new program fragment to replace the old one, and is denoted by a macro `?RULE` with the format of

```
?RULE(Template, NewCode, Cond),
```

where `Template` is a template representing the kind of code fragment to search for; `Cond` is an Erlang expression that evaluates to `true` or `false`; and `NewCode` is an Erlang expression that returns the new code fragment in the format of a string or an AST. All the meta-variables/atoms declared in `Template` are made visible to `NewCode` and `Cond` by means of parse transformation, i.e., applying transformations to the parse tree generated by the compiler before it is further processed and checked for errors, and can be referenced in

defining them; furthermore, it is also possible for `NewCode` to define its own meta-variables to represent code fragments.

Erlang allows a programmer to make many arbitrary syntactic decisions while constructing an expression or function. For example, a `case` expression in Erlang can have multiple expression clauses, and there is often flexibility in the order of the clauses. This flexibility will significantly increase the number or complexity of the rules that have to be written to describe a transformation. To address this problem, we have introduced the concept of a *meta-rule* that provides a concise way to express a collection of rules that a user has to write in order to cover the various syntactic choices that could be made when writing an expression. Like rules, a meta-rule is denoted by a macro `?META_RULE` with the format of

```
?META_RULE(Template, NewCode, Cond),
```

however, a meta rule uses more flexible algorithms for pattern matching, and for the generation of new object code. More details about meta-rules can be found in the Wrangler documentation.

3.2 Support for Program Analysis

Program analysis plays a vital role in refactoring. Very often the program analysis process needs to collect some syntactic or semantic information from an AST. This task is supported by Wrangler in two ways. First, information derived from the program by Wrangler is attached to the AST as *annotations*: we provide functions to extract these annotations from the AST. Second, information available at different nodes can be *collected* together: our API provides a macro to support this collection.

The Erlang macro `?COLLECT` is defined to allow information collection from nodes that match the template specified and satisfies certain conditions. Calls to the macro `?COLLECT` have the format:

```
?COLLECT(Template, Collector, Cond),
```

in which `Template` is a template representation of the kind of code fragments of interest; `Cond` is an Erlang expression that evaluates to either `true` or `false`; and `Collector` is an Erlang expression which retrieves information from the current AST node. We call an application of the `?COLLECT` macro a *collector*.

3.3 AST Traversal Strategies

Each transformation rule or collection macro is written to be applied to a particular node of an AST; in order for these operations to be applied across a complete AST it is necessary to use an AST *traversal strategy*, as introduced in [1]. An AST traversal strategy walks through the AST in certain order, and applies transformation rules to nodes that meet certain conditions or collects some information from each node visited when program analysis is concerned.

A number of pre-defined AST traversal strategies are provided through the Wrangler API. Traversal strategies can be distinguished according to three criteria:

- The *purpose* of the traversal, i.e., collection information (TU) or AST transformation (TP).
- The *termination* condition for the traversal (FULL/STOP/ ONCE), i.e. whether to continue the AST traversal after a successful match and how.
- The *order* in which the AST nodes are visited, i.e top-down traversals (TD) or bottom-up traversals (BU).

In Wrangler, a traversal strategy macro is named to reflect the three aspects above. For example, the traversal strategy `FULL_TD_TU` means that the AST is to be traversed top-down, all the nodes will be visited, and the traversal will return the information collected.

A traversal strategy macro takes two arguments. The first is a collection of transformation rules or a collection of information collectors, and the second specifies the scope to which the transformation or analysis is to be applied.

4. Scripting Composite Refactorings

The idea of composite refactorings was first proposed by Opdyke [16], and investigated by Roberts [17] and others. Existing approaches to composite refactoring tend to focus on the derivation of a combined precondition for a composite refactoring, so that the entire precondition of the composite refactoring can be checked on the initial program before performing any transformation [3, 7]. The ostensible rationale for this is to give improved performance of the refactoring engine. However, given the usual way in which refactoring tools are used in practice – where the time to decide on the appropriate refactoring to apply will outweigh the execution time – we do not see that the efficiency gains that this approach might give are of primary importance to the user.

In contrast, our aim is to increase the *usability* and *applicability* of the refactoring tool, by expanding the way in which refactorings can be put together. Our work does not try to carry out precondition derivation, instead each primitive refactoring is executed in the same way as it is invoked individually, i.e., precondition checking followed by program transformation. While it may be less efficient when an *atomic composite refactoring*, whose meaning we will explain later, fails during the execution, it does have its advantages in expressivity. In an overview, Wrangler's framework tries to address the following limitations when composite refactorings are described manually:

- When the number of primitive refactoring steps involved is large, enumerating all the primitive refactoring commands could be tedious and error prone.
- The static composition of refactorings does not support generation of refactoring commands that are program-dependent or refactoring scenario dependent, or where a

subsequent refactoring command is somehow dependent on the results of an earlier application.

- Some refactorings refer to program entities by source location instead of name, as this information may be extracted from cursor position in an editor or IDE, say. Tracking of locations is again tedious and error prone; furthermore, the location of a program entity might be changed after a number of refactoring steps, and in that case locations become untrackable.
- Even though some refactorings refer to program entities by name (rather than location), the name of a program entity could also be changed after a number of refactoring steps, which makes the tracking of entity names hard or sometimes impossible, particularly when non-atomic composite refactorings are involved.

We resolve these problems in a number of ways:

- Each primitive refactoring has been extended with a *refactoring command generator* that can be used to generate refactoring commands in batch mode. A command generator searches the AST of a program for refactoring candidates according to the constraints on arguments specified, and returns a set of refactoring commands.
- A command generator can generate commands lazily, i.e., a refactoring command is generated only as it is to be applied, so we can make sure that the information gathered by the generator always reflects the latest status, including source locations, of the program under refactoring.
- Wrangler always allows a program entity to be referenced using its original name, as it performs name tracking behind the scenes.
- Finally, and most importantly, we provide a small domain-specific language (DSL) to allow composition of refactorings in a compact and intuitive way. The DSL allows users to have fine-grained control over the generation of refactoring commands and the interaction between the user and the refactoring engine so as to allow decision making during the execution of a composite refactoring.

4.1 Transactions: atomic vs non-atomic composition

We use the notion of *atomic* and *non-atomic* to control the propagation of failure during the execution of a composite refactoring. If a composite refactoring succeeds only if all of its direct constituent refactorings succeed, then we say this composition is an atomic composition; a non-atomic composite refactoring always succeeds. The failure of an atomic composite refactoring will leave the original program unchanged. Another way of expressing this would be to say that atomic refactorings are treated as *transactions*.

4.2 The Domain-Specific Language

We give an overview of the definition of the DSL now, but more details can be found in [10]. The DSL, as shown in

$$
\begin{aligned}
&RefacName ::= rename\_fun \mid rename\_mod \\
&\qquad \mid rename\_var \mid new\_fun \mid gen\_fun \mid ... \\
&PR ::= \{refactoring,\ RefacName,\ Args\} \\
&CR ::= PR \\
&\qquad \mid \{interactive,\ Qualifier,\ [PRs]\} \\
&\qquad \mid \{repeat\_interactive,\ Qualifier,\ [PRs]\} \\
&\qquad \mid \{if\_then,\ \text{fun}() \rightarrow Cond\ \text{end},\ CR\} \\
&\qquad \mid \{while,\ \text{fun}() \rightarrow Cond\ \text{end},\ Qualifier,\ CR\} \\
&\qquad \mid \{Qualifier,\ [CRs]\} \\
&PRs ::= PR \mid PRs,\ PR \\
&CRs ::= CR \mid CRs,\ CR \\
&Qualifier ::= atomic \mid non\_atomic \\
&Args ::= ...A\ list\ of\ Erlang\ terms... \\
&Cond ::= ...An\ Erlang\ expression\ that\ evaluates\ to \\
&\qquad a\ \texttt{boolean}\ value...
\end{aligned}
$$

Figure 1. The DSL for scripting composite refactorings

Figure 1, is defined in Erlang syntax, using tuples and atoms. In the definition, PR denotes a primitive refactoring, and CR denotes a composite refactoring.

- A primitive refactoring is, by definition, an atomic composite refactoring. A primitive refactoring is an elementary behaviour-preserving source-to-source program transformation that consists of a set of preconditions and a set of transformation rules. When a primitive refactoring is applied to a program, all the preconditions are checked before the program is actually transformed by applying all the transformation rules. A primitive refactoring fails if the conjunction of the set of preconditions returns false; otherwise the primitive refactoring succeeds.
- $\{interactive,\ Qualifier,\ [PRs]\}$ represents a list of primitive refactorings to be executed in an interactive way, that is, before the execution of every primitive refactoring, Wrangler asks the user for confirmation that he/she really wants that refactoring to be applied. The confirmation question is generated automatically by Wrangler.
- $\{repeat\_interactive,\ Qualifier,\ [PRs]\}$ also represents a list of primitive refactorings to be executed in an interactive way, but different from the previous one, it allows user to repeatedly apply one refactoring, with different parameters supplied by the user, until he/she decides to stop. The user-interaction is carried out before each run of a primitive refactoring.
- $\{if\_then,\ \text{fun}() \rightarrow Cond\ \text{end},\ CR\}$ represents the conditional application of CR, i.e., CR is applied only if $Cond$, which is an Erlang expression, evaluates to `true`. We wrap $Cond$ in an Erlang function closure to delay its evaluation until it is needed.
- $\{while,\ \text{fun}() \rightarrow Cond\ \text{end},\ Qualifier,\ CR\}$ allows CR, which is generated dynamically, to be continually ap-

plied until $Cond$ evaluates to `false`. $Qualifier$ specifies whether the refactoring is to be applied atomically or not.

- $\{Qualifier, [CRs]\}$ represents the composition of a list of composite refactorings into a new composite refactoring, where the qualifier states whether the resulting refactoring is applied atomically or not.

5. Generic Refactoring Behaviour

While every refactoring has its own pre-condition analysis and transformation rules, there are some parts of the refactoring process that are common to most refactorings, such as the generation and annotation of ASTs, the outputting of refactoring results, the collecting of change candidates, and the workflow of the refactoring process. We can use an Erlang *behaviour* to capture this genericity.

An Erlang behaviour is an application framework that is parameterized by a *callback* module. The behaviour implements the generic parts of the problem, while the callback module implements the specific parts. We have defined two behaviours, namely *gen_refac* and *gen_composite_refac*. *gen_refac* is the behaviour to use when defining a primitive refactoring, whereas *gen_composite_refac* is the behaviour for defining a composite refactoring.

With both behaviours, we aim to encapsulate those parts that are generic to all refactorings in the behaviour module, and let the user to handle the parts that are specific to the refactoring under consideration. Another advantage of having a refactoring behaviour is the ease of integration with an IDE. Since the integration only involves the IDE and the behaviour module, it is done by Wrangler; the developer of the callback module need not be concerned.

6. Examples

In this section, we demonstrate our approach by means of four examples. All the examples are available for download (as part of Wrangler).

Example 1. Remove an argument. Removing an unused argument from a function definition is a common refactoring supported by most refactoring tools. Part of the implementation of this refactoring is shown in Figs 2 and 3. The part of code not shown includes two more transformation rules, which handle the `M:F/A` format representation of function names and the function type specification respectively, and the function for refactoring modules that directly depend on the current module; however this should not affect one's understanding of the way in which refactorings are implemented in Wrangler. We explain the implementation now.

- **Line 2** declares that this module implements the *gen_refac* behaviour, and **lines 4-5** exports the *callback* functions that are required by *gen_refac*.
- This refactoring needs the user to specify the index of the argument to be removed, and **lines 6-7** defines the prompt used when asking the user to input the index.

```
1.  -module(refac_remove_an_argument).
2.  -behaviour(gen_refac).
3.  -include_lib("wrangler/include/wrangler.hrl").
4.  -export([input_par_prompts/0,select_focus/1,
5.           check_pre_cond/1,selective/0,transform/1]).

6.  -spec input_par_prompts() -> [string()]
7.  input_par_prompts() -> ["Parameter Index : "].

8.  -spec select_focus(Args::#args{}) ->
9.                {ok, syntaxTree()} |{ok, none}
10. select_focus(_Args=#args{current_file_name=File,
11.                          cursor_pos=Pos}) ->
12.   api_interface:pos_to_fun_def(File, Pos).

13. -spec check_pre_cond(Args::#args{})->
14.                  ok | {error, Reason}
15. check_pre_cond(Args=#args{focus_sel=FunDef,
16.                           user_inputs=[Ith]}) ->
17.   {_M,_F,A} = api_refac:fun_define_info(FunDef),
18.   case Ith>=1 andalso Ith=<A of
19.     true -> check_pre_cond_1(Args, Ith);
20.     false -> {error, "Index is invalid."}
21.   end.

22. check_pre_cond_1(#args{focus_sel=FunDef},Ith) ->
23.   IthArgs=?STOP_TD_TU(
24.              [?COLLECT(
25.                 ?T("f@(Args@@)when Guard@@-> Bs@@;"),
26.                 lists:nth(Ith, Args@@),
27.                 true)], FunDef),
28.   case lists:all(fun(A) ->
29.                    api_refac:type(A) == variable
30.                    andalso api_refac:var_refs(A) ==[]
31.                  end, IthArgs) of
32.     true ->  ok;
33.     false ->{error, "Parameter cannot be removed."}
34.   end.
```

Figure 2. Remove an argument (part 1)

- **lines 8-12** defines the callback function `select_focus`, which allows the user to select the function of interest by pointing the cursor to the function definition, and returns the AST representation of the function selected.
- **Lines 13-34** implements the pre-condition checking. The API function `fun_define_info` takes the function definition as input, and returns the module name, function name and arity of the function in a tuple. Apart from checking that the index value is valid, the pre-condition also checks that the argument to be removed is represented as a variable, and not used by the function definition. Since an Erlang function could consist of multiple function clauses, the argument from each clause needs to be checked; the collecting of these arguments is done by the code between **lines 23-27**, where a `COLLECT` macro is used together with an AST traversal strategy. i.e. `STOP_TD_TU`, to retrieve the argument from each clause.
- The callback function *transform* defined in Fig 3 performs the program transformation using the AST traversal strategy `FULL_TD_TP`, which does a full top-down traversal of

```
35. transform(Args=#args{current_file_name=File,
36.                      focus_sel=FunDef,
37.                      user_inputs=[Ith]}) ->
38.  MFA = api_refac:fun_define_info(FunDef),
39.  ?FULL_TD_TP([rule1(MFA,Ith),rule2(MFA,Ith)],[File]).

40. rule1({M,F,A}, Ith) ->
41.    ?RULE(?T("f@(Args@@) when Guard@@ -> Bs@@;"),
42.       begin
43.         NewArgs@@=delete(Ith, Args@@),
44.         ?TO_AST("f@(NewArgs@@) when Guard@@->Bs@@;")
45.       end,
46.       api_refac:fun_define_info(f@) == {M, F, A}).

47. rule2({M,F,A}, Ith) ->
48.   ?RULE(?FUN_APPLY(M,F,A),
49.         begin
50.           Args = api_refac:get_app_args(_This@),
51.           NewArgs=delete(Ith, Args),
52.           api_refac:update_app_args(_This@, NewArgs)
53.         end, true).

54. delete(Ith, Args) ->
55.   ... delete the Ith argument from a list...
```

Figure 3. Remove an argument (part 2)

```
transform(_Args=#args{search_paths=SearchPaths})->
    ?FULL_BU_TP([replace_bug_cond_macro_rule(),
                 logic_rule_1(),
                  .....
                 if_rule_1()], SearchPaths).

replace_bug_cond_macro_rule() ->
    ?RULE(?T("Expr@"), ?TO_AST("false"),
          is_bug_cond_macro(Expr@)).

logic_rule_1() ->
    ?RULE(?T("not false"),?TO_AST("true"),true).

if_rule() ->
    ?RULE(?T("if Conds1@@, false,Conds2@@ -> Body1@@;
                 true -> Body2@@
              end"), Body2@@, true).

is_bug_cond_macro(Expr) ->
    api_refac:type(Expr) == macro andalso
        is_bug_cond_name(?PP(Expr)).

is_bug_cond_name::string()-> boolean().
is_bug_cond_name(Str) -> ..check the macro name....
```

Figure 4. Eliminating Bug Pre-conditions

the AST while trying to apply the rules to the each of the nodes visited. The first rule, i.e. `rule1` defined between **lines 40-46**, removes the `Ith` parameter from the function clause definition, and the second rule, i.e. `rule2` defined between **lines 47-53** removes the `Ith` argument from an application site of the function. The pre-defined macro `FUN_APPLY` is used to capture the various ways a function can be called in Erlang. The function `delete` is a normal Erlang function that deletes the `Ith` element from a list.

Example 2. Bug pre-condition This example illustrates the implementation of a refactoring that is more application-specific, therefore not likely to be included in the distribution of refactoring tools. Without tool support, this kind of refactoring is likely to be applied manually. This example shows how little effort a user needs to invest to implement a particular refactoring that meets his/her own needs.

This particular example comes from testing, and specifically testing based on models. It is often necessary to patch a faulty model to be able to continue testing, otherwise the same faults keep being detected, and no new faults are found. The solution is to add macros to the code: if the bug is present in the code, the macro is true, otherwise it is false. However, when delivering the code after testing, it is necessary to remove uses of those macros. This is a repetitive process, and previously had to be done manually. One of the files we examined contains 43 use places of the bug macros; obviously, manual refactoring of all these use cases could be a time-consuming process, particularly if it had to be performed repeatedly.

The actual refactoring is simple: recognise the macro applications of syntax `?xxx_bug_nnn`, replace the macro application with `'false'`, and clean up the code accordingly. For example, the code fragment

```
if ?com_bug_006 -> set_raw_data(Id, Data, NewS);
           true -> NewS
end
```

should be refactored to the variable `NewS`.

To automate this refactoring process, we once again implement a *gen_refac* behaviour. This refactoring does not need user input, focus selection or pre-condition checking, therefore the only salient function is the `transform` callback function. The definition of `transform` together with some of the rules are shown in Fig 4. In total, 14 rules were defined to handle various scenarios. With this refactoring support, eliminating bug pre-condition macros becomes a matter of simply pressing a button.

The previous two examples demonstrate how Wrangler's template- and rule-based API makes it feasible for users to define refactorings themselves. With the next two examples, we will demonstrate how Wrangler's DSL makes it practical for users to script composite refactorings.

Example 3. Batch prefixing of module names. Renaming module names is one of the most common refactorings. When a collection of modules need to be renamed in a systematic way, refactoring support for batch renaming of module names becomes handy, but composite refactorings of this kind are not supported by most refactoring tools. Wrangler's DSL support for composite refactorings means that batch renaming of module names is easy to achieve: for example, Fig 5 shows the script that adds a given prefix to every Erlang module name in a project. The script implements the *gen_composite_refac* behaviour, i.e. all the callback functions required by this behaviour are defined by this module. This refactoring asks the user to input the prefix, as shown by the function `input_par_prompts`.

```
1. -module(refac_batch_prefix_module).

2.  -export([input_par_prompts/0, select_focus/1,
3.         composite_refac/1]).
4.  -behaviour(gen_composite_refac).
5.  -include_lib("wrangler/include/wrangler.hrl").

6.  input_par_prompts() -> ["Prefix: "].
7.  select_focus(_Args) -> {ok, none}.

8.  -spec composite_refac(Args::#args{})->
9.                        composite_refac().
10. composite_refac(_Args=#args{user_inputs=[Prefix],
11.                        search_paths=SearchPaths}) ->
12.  {non_atomic,
13.    {refac_, rename_mod,
14.       [{file, fun(File)->
15.                   filename:extension(File)==".erl"
16.               end},
17.        {generator, fun(File) ->
18.                        Prefix++filename:basename(File)
19.                    end},
20.        false, SearchPaths])).
```

Figure 5. Batch prefixing of module names

The script of this composite refactoring is defined in function *composite_refac*. **Lines 13-20** use the refactoring command generator for the *rename_mod* refactoring, represented as a tuple with `refac_` as the first element, to generate a collection of refactoring commands. The command generator says that for every file in the directories specified by `SearchPaths`, if this file is an Erlang file, i.e. has a file extension of `.erl`, then generate a *rename_mod* command for this file, and the new module name is generated by prefixing the original module name with the prefix given by the user.

The `'false'` in **line 20** tells the command generator that all the commands should be generated before the first refactoring is to be applied, i.e. no lazy command generation is used. Finally, the top-level qualifier *non_atomic* says that if one of the refactorings fails to rename a module, the refactoring process does not fail as a whole.

Example 4. Batch clone removal Wrangler's similar code detection functionality [8] is able to detect code clones in an Erlang program, and help with the clone elimination process. For each set of code fragments that are clones of each other, Wrangler generates a function, named `new_fun`, which represents the *least-general common abstraction* of the clones; cloned code fragments can then be replaced by applications of this function, thus eliminating duplicate code.

The general procedure to remove such a clone in Wrangler is to copy and paste the function `new_fun` into a module, then carry out a sequence of refactoring steps as follows:

- Rename the function to reflect its meaning.
- Rename variables if necessary, especially those of the form `NewVar`i, which are generated by the clone detector.
- Swap the order of parameters if necessary.
- Export the function if some clones are in other modules.

```
{atomic,
  [{interactive, atomic, RENAME_FUNCTION new_fun},
   {atomic, RENAME_VARIABLES NewVar*},
   {repeat_interactive, atomic, SWAP_ARGS},
   {if_then, NOT_EXPORTED, ADD_TO_EXPORT},
   {non_atomic, {refac, fold_expr, CLONE_INSTANCES}}
  ]}
```

Figure 6. Batch Clone Elimination

- For each module that contains one or more cloned code fragments, apply the 'fold' refactoring to replace the clones in that module with calls to the new function.

The process can be scripted as a *gen_composite_refac* behaviour, and full details are given in [10]; here we review the top-level structure, as given in Fig. 6.

- The whole refactoring is *atomic*, i.e. a single transaction.
- This does not stop parts of the refactoring not being transactions (i.e. *non-atomic*): if replacing a particular clone instance fails, this will not affect the correctness of the refactoring. On the other hand, swapping function arguments must be a transaction: we must have all the arguments in the correct order before we continue.
- We may we need to export the renamed function, but we don't know its new name. The infrastructure will track this, and allow us to refer to the function by its old name.
- Aspects of the refactoring are interactive: we prompt for names of variables and functions. Interactions can also be repeated, as in the case of swapping argument positions.

7. Related Work

Rule-based structural transformation is used by many other meta-programming paradigms, we discuss these now.

TXL [4] is a generalised source-to-source translation system. A TXL program takes as input a context-free grammar in BNF-like notation, and a set of transformation rules to be applied to inputs parsed using the grammar.

Stratego/XT [1] is a language for transformation of abstract syntax trees, and XT is a bundle of transformation tools that combine it with tools for other aspects of program transformation, such as parsing, pretty-printing, etc.

The ASF+SDF [19] meta-environment supports program analysis, transformation, generation of interactive program environments and pretty-printers, etc. As the successor of the ASF+SDF, the recently-released RASCAL [6] is a language that aims to provide a general framework for high-level integration of source code analysis and manipulation.

JunGL [20] is a DSL for implementing refactorings, combining an imperative core, ML-like algebraic data types with pattern-matching for representing syntax trees, and features for defining attributes on edges between AST nodes. Instead of providing declarative descriptions of transformations, JunGL relies on imperative modification of the AST.

The tools above give powerful support for program transformation, and infrastructure on which to build program analysis; others explicitly support the analysis of conditions for composite refactorings; we examine these now.

The idea of composite refactorings was proposed by Opdyke [16], and investigated by Roberts [17]. This work focused on the derivation of a composite refactoring's preconditions from the pre- and postconditions of its constituent refactorings. Ó Cinnéide [3] extends Roberts' approach in various ways including static manual derivation of pre- and postconditions for a composite refactoring.

`ContTraCT` is an experimental refactoring editor for Java [7], allowing composition of larger refactorings from existing ones. `ContTraCT` has two basic composition operations: AND and OR, that correspond to the atomic and non-atomic composition described in this paper. A formal model based on the notion of *backward transformation description* is used to derive the preconditions of an AND-sequence.

Extensibility has long been a question for those designing refactoring tools [13]. MOOSE [15] is a language-generic environment that provides support for re-engineering complex software systems. While it is clearly tailorable, and has been used successfully in a variety of projects, it is too heavyweight for the working programmer to use. Moreover, the effort of tuning a system of this sort to handle the specifics of a language (see e.g. [18] on renaming in Java) makes it difficult for it to perform as well as a system designed for a particular language.

Jbrx [12] is a Java refactoring tool that provides extensibility through giving users access to a program representation in XML, over which they can describe transformations and pre-conditions, but this requires users to learn both XML and the Sapid/XML tool platform.

RubyTL [5] is the closest to our work, in presenting a language for describing Ruby program transformations, embedded in Ruby. It does not combine this with the use of concrete syntax for specifying new atomic transformations; rather it concentrates on transformations on (meta-)models.

8. Conclusions and Future Work

We have presented the API and DSL based approach taken by Wrangler to make it practical for users to define for themselves elementary and composite refactorings. Through a series of examples, we have demonstrated the usability and power of this framework. We believe that making refactoring tools user-extensible is crucial step towards removing the barriers that inhibit the full take-up of refactoring tools.

In the future, we aim to carry out more case studies to see how the support for user-defined refactorings is perceived by users, and whether this changes the way they refactor their code. We will also explore how this work applies to other tools, such as HaRe, a refactoring tool for Haskell programs.

This research was supported by EU FP7 collaborative project 215868, ProTest (`www.protest-project.eu`).

References

[1] M. Bravenboer, K. T. Kalleberg, et al. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. of Computer Prog.*, 72(1-2), 2008.

[2] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, Inc., 2009.

[3] M. O. Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. PhD thesis, Trinity College Dublin, 2000.

[4] J. R. Cordy. Source Transformation, Analysis and Generation in TXL. In *PEPM*, 2006.

[5] J. S. Cuadrado, J. G. Molina, and M. Menarguez. Rubytl: A practical, extensible transformation language. In *2nd European Conference on Model Driven Architecture*, 2006.

[6] P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM*, 2009.

[7] G. Kniesel and H. Koch. Static Composition of Refactorings. *Sci. Comput. Program.*, 52, August 2004.

[8] H. Li and S. Thompson. Incremental Code Clone Detection and Elimination for Erlang Programs. In *Fundamental Approaches to Software Engineering (FASE'11)*, 2011.

[9] H. Li and S. Thompson. A User-extensible Refactoring Tool for Erlang Programs. Technical Report 4-11, School of Computing, Univ. of Kent, UK, 2011.

[10] H. Li and S. Thompson. A Domain-Specific Language for Scripting Refactoring In Erlang. In *Fundamental Approaches to Software Engineering (FASE'12)*, 2012.

[11] H. Li, S. Thompson, G. Orosz, and M. Töth. Refactoring with Wrangler, updated. In *ACM SIGPLAN Erlang Wkshp*, 2008.

[12] K. Maruyama and S. Yamamoto. Design and Implementation of an Extensible and Modifiable Refactoring Tool. In *Proceedings of IWPC '05*. IEEE Computer Society, 2005.

[13] T. Mens and A. V. Deursen. Refactoring: Emerging Trends and Open Problems, 2003.

[14] E. Murphy-Hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 99, 2011.

[15] O. Nierstrasz, S. Ducasse, and T. Gîrba. The Story of Moose: an Agile Reengineering Environment. In *Proceedings of the 10th European software engineering conference*. ACM, 2005.

[16] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Univ. of Illinois, 1992.

[17] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, Univ. of Illinois, 1999.

[18] M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for java. In *OOPSLA '08*. ACM, 2008.

[19] M. van den Brand et al. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *Compiler Construction*, volume 44, 2001.

[20] M. Verbaere et al. JunGL: A scripting Language for Refactoring. In *Proceedings of the 28th international conference on Software engineering(ICSE)*, 2006.