Refactoring Support for Modularity Maintenance in Erlang

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

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

Abstract

Low coupling between modules and high cohesion inside each module are key features of good software architecture. Systems written in modern programming languages generally start with some reasonably well-designed module structure, however with continuous feature additions, modifications and bug fixes, software modularity gradually deteriorates. So, there is a need for incremental improvements to modularity to avoid the situation when the structure of the system becomes too complex to maintain.

We demonstrate how Wrangler, a general-purpose refactoring tool for Erlang, can be used to maintain and improve the modularity of programs written in Erlang without dramatically changing the existing module structure. We identify a set of "modularity smells" and show how they can be detected by Wrangler and removed by way of a variety of refactorings implemented in Wrangler. Validation of the approach and usefulness of the tool are demonstrated by case studies.

1 Introduction

Modular programming, as a software design technique, improves the maintainability and reusablity of software by enforcing well-defined boundaries between components or modules. A module captures a set of design decisions which are hidden from other modules, and the interaction among the modules should primarily be through module interfaces [9].

Low coupling between modules and high cohesion inside each module are the key aspects of modular programming [13]. Unlike monolithic legacy application systems written in programming languages that did not support modular programming, most recent systems are structured in a modular way.

However, without proper maintenance, software structure gradually deteriorates over years of feature additions, changes and bug fixes, and finally gets to a state that the program structure is too complex for anyone to fully understand it. The ageing of software architecture could be avoided by reviewing the system structure regularly and refactoring it whenever the symptoms of modularity problems start to show. This kind of incremental modularity improvement slows down such deterioration and improves the maintainability of the system. By carrying out small changes each time, we avoid having to make dramatic changes to existing module structures in a single step.

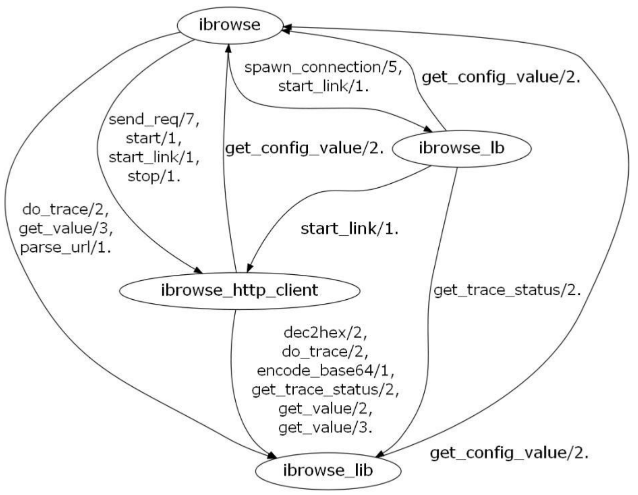
When a software system is large, detecting modularity flaws and refactoring module structure are both hard without proper tool support.

- Detecting modularity flaws and working out steps to eliminate them both need an overall analysis of the system under consideration.
- Restructuring a system, no matter at what scale, usually involves module interface changes and affects multiple modules in the system. This is a tedious process, and bugs can be introduced very easily, potentially without being noticed.

Erlang is a modern functional programming language supporting modular programming. Erlang's module system is simple, allowing the export of functions defined in a module; calls of these in other modules are usually in fully qualified `Module:Function` form. Our case studies shown that the problem of modularity deterioration is not uncommon in existing Erlang programs. Figure 1 shows an example module graph generated from an Erlang program. The nodes in the graph are labelled with module names, the edges are labelled with the names and arities of the functions called by the client module, pointed to by the arrow. Although there are only four modules in this module graph, there are already a number of cyclic module dependencies, and no layered architecture for the system is apparent.

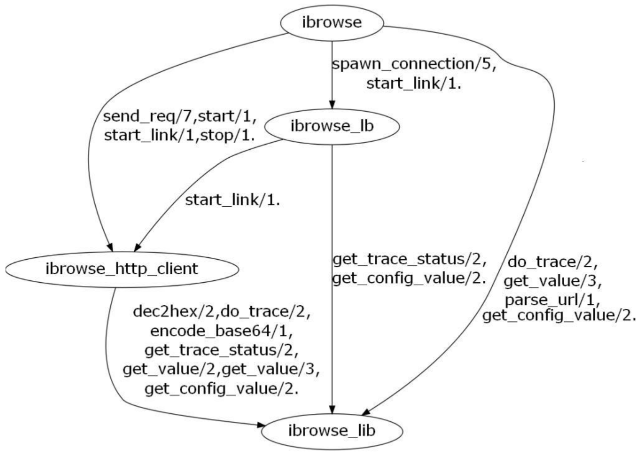
Further examination of the graph shows that all the cyclic dependencies are actually caused by the same function, namely `get_config_value/2`. This obviously indicates some sort of "module structure bad smell". We took a closer look into the source code, and found that this function is commented as "internal export", which means that the function is an internal function, but also exported by the module. Exporting of functions that are meant to be internal is a programming practice not recommended.

Refactoring the code by moving the function `get_config_value/2` from its current module `ibrowse` to module `ibrowse_lib` results in the module structure as shown in Figure 2, which is clearer and more obviously layered.



ibrowse
spawn_connection/5, start_link/1.
get_config_value/2.
send_req/7, start/1, start_link/1, stop/1.
get_config_value/2.
ibrowse_lb
do_trace/2, get_value/3, parse_url/1.
start_link/1.
ibrowse_http_client
get_trace_status/2.
dec2hex/2, do_trace/2, encode_base64/1, get_trace_status/2, get_value/2, get_value/3.
get_config_value/2.
ibrowse_lib

Figure 1. Module graph before refactoring



ibrowse
spawn_connection/5, start_link/1.
send_req/7,start/1, start_link/1,stop/1.
ibrowse_lb
start_link/1.
ibrowse_http_client
get_trace_status/2, get_config_value/2.
do_trace/2, get_value/3, parse_url/1, get_config_value/2.
dec2hex/2,do_trace/2, encode_base64/1, get_trace_status/2, get_value/2,get_value/3, get_config_value/2.
ibrowse_lib

Figure 2. Module graph after refactoring

Wrangler [5, 6] is a general-purpose refactoring tool for Erlang developed by the authors. To make incremental modularity improvement feasible for Erlang programs, we have extended Wrangler with support for modularity maintenance. This is achieved in three steps:

- Firstly, a number of most common modularity flaws, or "modularity smells", are identified, and automatic detection of these modularity flaws is implemented.
- Secondly, for each modularity smell detected, Wrangler gives refactoring suggestions in the format of refactoring commands, which would eliminate the smell detected without introducing new modularity smells into the system.
- Thirdly and finally, the original refactoring to move a function from one module to another has been extended to allow a collection of functions to be moved in one single step.

These functionalities together make incremental modularity maintenance much easier. The rest of the paper is organised as follows. In Section 2, we give a brief introduction to Erlang and Wrangler; in Section 3, we identify a number of modularity smells that are common in Erlang programs. In Section 4, we explain Wrangler's support of modularity smell detection and elimination, and in Section 5, we discuss Wrangler's refactoring support for module restructuring; a case study demonstrating the usefulness of the tool is shown in Section 6. Section 7 discusses implementation considerations of the tool, Section 8 gives an overview of related work, and finally, Section 9 concludes the paper and briefly discusses future work.

2 Erlang and Wrangler

Erlang [1, 3] 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.

Erlang comes with a simple, non-hierarchical module system. An Erlang program typically 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; furthermore, a module may only export functions which are defined in the module itself.

Calls to functions defined in other modules should qualify the function name with the module name: the function `foo` from the module `bar` is called as: `bar:foo(...)`. Despite the fact that this can be avoided by means of `import` directives, practice within the Erlang community recommends the use of this *fully qualified* notation. Figure 3 shows an Erlang module containing the definition of the factorial function. In this example, `fac/1` denotes the function `fac` with arity of 1. In Erlang, a function name can be defined with different arities, and the same function name with different arities can represent entirely different functions computationally.

```
-module (fact).
-export ([fac/1]).
fac(0) -> 1;
fac(N) when N > 0 -> N * fac(N-1).
```

Figure 3. Factorial in Erlang

Wrangler [5, 6] is a tool that supports interactive refactoring of Erlang programs. It is integrated with Emacs and XEmacs as well as with Eclipse through the ErlIDE plugin. Wrangler itself is implemented in Erlang. Wrangler supports a variety of elementary structural refactorings, process refactorings, as well as a set of "code smell" inspection operations, together with facilities to detect and eliminate duplicated code [4]. Wrangler is downloadable from `http://www.cs.kent.ac.uk/projects/wrangler/Home.html`.

3 Modularity Smells

Modules are the basic unit of code in Erlang. Functions in an Erlang program are grouped together into modules, each containing functions that logically belong together. Each module makes itself available to the other modules by exporting a list of functions defined in this module. In a well-designed system, each module should serve a clear purpose or goal, and only export functions that are expected to be part of a well-defined interface or API (Application Programming Interface). While we could assume that most programmers follow some kind of design principles when deciding to which module a function should belong, and which functions should be made visible to other modules, there are still some modularity smells which we found to be common in real-world Erlang programs, and these are the subject of this section.

3.1 Improper Inter-Module Dependency

As mentioned earlier, each Erlang module exports a set of functions available for use by other modules. Ideally, the functions that are exported should constitute a well-designed API, which represents the services that the module has to offer. All the remaining functions are internal functions meant to be used only within the module itself.

During the course of system development, it is often the case that the developer finds that a function which he or she wants to implement already exists in another module, but is not exported by that module. At this stage, the developer could refactor the code to include that function in the API for a module to which the function logically belongs.

However, the alternative of adding the function name to the export list of the module is often chosen, without thinking of questions such as: should this function be an API function provided by this module? does its functionality conform to the purpose of this module? should the module that needs this function be dependent on this module? If the answer to any of these questions is 'no', then a code smell is being introduced by the export. A dependency introduced in this way cries out for further inspection.

3.2 Cyclic Dependent Modules

Cyclic dependent modules are a set of modules in which each module calls functions defined in every other module in the set, directly or indirectly. Cyclic dependency between two or more modules should always be avoided whenever it is possible because it affects the understandability and maintainability of the system; on the contrary, a tree-structured or acyclic module dependency gives a layered view of the program structure, which is much easier to understand and maintain as illustrated in Figures 1 and 2.

3.3 Modules Serving Multiple Goals

High cohesion inside each module is one of the key features of good software architecture. A module should ideally contain a collection of functions and data structures that are logically grouped together, and offer some well-defined common service to the rest of the system. It is therefore a modularity smell if a module provides a large collection of API functions that logically serve more than one goal. [1]

Modules serving multiple goals are in general harder to understand. Apart from that, modules serving multiple goals are more likely to depend heavily on other modules, or *vice versa*; in other words, they tend to have high *in* or *out* degree in the module graph. Such modules are more difficult to maintain: each time the module's interface is changed, all the places in the program where this module is used have to be checked. Also, each time a change is made to the interface of one of the modules that this module depends on, this module also needs to be checked.

Modules with multiple goals should be partitioned into smaller modules, each of which provides a clearly-defined service to its clients.

3.4 Very Large Modules

Large monolithic modules containing many lines of code potentially obscure the architecture of the system. Large modules are often modules with multiple goals, but not necessarily: for instance, it could be that a set of functions

[1] It has been conjectures that this would lead to a partition in the set of modules using this one: we will investigate this further.

within a module form an internal library that is used by other functions: this is a candidate to become a separate module which exports its services. This additional structure makes the system easier to understand, to maintain and to evolve, as well as increasing the opportunities for reuse of the library functions.

While it is difficult to give a hard and fast rule about module size, a recent Erlang text *Erlang Programming* [3] has suggested: "A manageable module should have no more than 400 lines of code, comments excluded."

4 Modularity Smell Detection and Refactoring Suggestions

Detecting modularity smells is the very first step to modularity improvement. Once a modularity smell has been detected, the next question is how to remove it. While eliminating modularity smells mostly involves moving functions from one module to another, it is not always clear which functions to move, and to where. With Wrangler, we aim to not only find modularity smells, but also give the user suggestions as to how to eliminate them.

In this section, we focus on Wrangler's detection of the modularity smells discussed in the previous section. For each kind of modularity smell, we also discuss the strategy used by Wrangler to work out how to eliminate it. The fundamental principles used by Wrangler when suggesting refactorings steps are that the refactoring steps suggested, if executed, should not introduce new modularity smells to the system, and the new module dependencies introduced to the system because of the refactoring should be minimal.

4.1 Improper Inter-Module Dependency

Detection. In principle a module should only export functions that are designed to be API functions; however, this principle can be very easily violated in practice. Erlang's module system allows any function defined in a module to be exported and used by the rest of the system. For an Erlang module to use functions exported by other modules, there is no need to import that module first as is required by other programming languages like Haskell, as long as the function name is qualified with its defining module. This is flexible, but also means that module dependency can be introduced in a rather ad-hoc way.

Syntactically, the export lists of an Erlang module make no distinction between functions that are designed to be API functions and functions that are meant to be internal but are also exported by the module, which we call *non-API* functions (there might by comments indicating this, as mentioned earlier). To be able to detect the export of non-API functions, and module dependency introduced by function calls to non-API functions, we try to mark each function exported by a module as an API function or a non-API function using static analysis and heuristics.

For the purposes of discussion we denote an Erlang program P as a collection of Erlang modules $\{M_1, ...M_n\}$, and each Erlang module as a collection of functions exported by the module $M_i = \{f_{i1}, ..f_{in}\}$.

For a given Erlang module $M_i = \{f_{i1}, ..f_{in}\}$, we use F_i^{ext} to denote the subset of M_i representing those functions that are not called by any other functions in the module that do not contribute to its definition; in other words, a function in F_i^{ext} is only called, directly or indirectly, by other functions in its *Strongly Connected Component (SCC)*. We use F_i^{int} to denote those functions in M_i that do not belong to F_i^{ext}, that is $M_i - F_i^{ext}$. As a convention, each function in F_i^{ext} is considered as an API function, therefore the major task is to classify each function from F_i^{int} as an API, or non-API function.

Within this context, for each function f_{ij} belonging to module $M_i = \{f_{i1}, ..f_{in}\}$, the probability score for it being an API function is calculated as:

$$APIScore(f_{ij}) = 1 - \min\{dist(f_{ij}, f^e) | f^e \in F_i^{ext}\}$$

where $dist\{f_{ij}, f_{ik}\}$ is a function calculating the *distance* between two functions f_{ij} and f_{ik}. If A and B denote the set of nodes reachable from f_{ij} and f_{ik} in the function callgraph of the Erlang program in question, then we define

$$dist(f_{ij}, f_{ik}) = 1 - \frac{2*|A \cap B|}{|A|+|B|}$$

Function f_{ij} is considered to be an API function if its $APIScore(f_{ij})$ is greater than a specified threshold, ϕ, which should be between 0 and 1. Obviously, a function belonging to F_i^{ext} always has an $APIScore$ of 1, therefore is always marked as an API function. We mark a function as a *non-API* function if its $APIScore$ is less than the required threshold, ϕ.

Given two Erlang modules M_i and M_j, we say M_j *directly depends on* M_i if some of the functions exported by M_i are directly called by functions defined in M_j. We represent this kind of direct module dependency as a three-element tuple $\{M_i, M_j, F\}$, where F is the set of functions that are exported by M_i and used by M_j, which we denote by $use(M_i, M_j)$. Given an Erlang program P, and an API score threshold ϕ, the set of improper module dependencies reported by Wrangler is given by:

$$\{\ \{M_i, M_j, use(M_i, M_j)\} \quad | \quad \forall f \in use(M_i, M_j),\ APIScore(f) < \phi\ \}$$

Refactoring Suggestion. An improper inter-module dependency introduced by non-API inter-module function calls could be eliminated by moving the non-API functions, together with those internal functions on which the non-API functions depend. They are moved to a third module, so that the non-API functions become API functions exported by that module.

For each improper inter-module dependency $\{M_i, M_j, use(M_i, M_j)\}$ detected, Wrangler gives possible options for the third module, to which the non-API functions could be relocated. The following constraints are applied when choosing the target module.

- Moving a non-API function to the target module should not introduce a cyclic module dependency.
- Moving a non-API function to the target module should make the function an API function of that module as measured by its *APIScore*.
- The number of new module dependencies introduced should be minimal.
- A target module that is closer to M_i and M_j in terms of the length of the shortest paths connecting them in the module graph is favoured over a target module that is further away.

4.2 Cyclic Module Dependency

Detection. A *cyclic module dependency* is a minimal set of modules $\{M_1, M_2, ...M_n\}$, in which $M_{i(1<i\leq n)}$ directly depends on M_{i-1}, and M_1 directly depends on M_n. Modules that are cyclically dependent are straightforward to locate given the module graph and function callgraph generation functionalities provided by Wrangler. For a set of cyclically dependent modules, Wrangler reports not only the module names, but also the functions called between each pair of dependent modules. To take the function calls between each pair of directly dependent modules into account, we denote a cyclic module dependency relation as:

$$M_1 \xrightarrow{F_1} M_2 \xrightarrow{F_2} ... \xrightarrow{F_{n-1}} M_n \xrightarrow{F_n} M_1 \quad (n >= 2)$$

where $M_i \xrightarrow{F_i} M_j$ means that module M_j directly depends on module M_i, and F_i represents those functions that are exported by module M_i, and called by functions defined in module M_j, i.e. $use(M_i, M_j)$.

Elimination. Give a cyclic module dependency,

$$M_1 \xrightarrow{F_1} M_2 \xrightarrow{F_2} ... \xrightarrow{F_{n-1}} M_n \xrightarrow{F_n} M_1 \quad (n >= 2),$$

the following steps are used to work out how to break the cyclic module dependency.

Step 1: we mark each function in $F_{i(i=1,..n)}$ as either an API function or a non-API function using the same approach as described in Section 4.1. Let us use F_i^{api} to represent the subset of F_i representing functions marked as API functions.

Step 2: if there exists j, such that $F_j^{api} = \phi$, then the dependency between module M_j and M_{j+1} (M_1 when $j = n$) is regarded as improper inter-module dependency, and Wrangler would suggest the moving of functions in F_j the same way as discussed in Section 4.1; otherwise continue to Step 3.

Step 3: We try to distinguish two kinds of cyclic module dependency, namely *intra-layer* and *inter-layer*. To do so, for each $F_{i(i=1,..n)}$, we use $Callers(M_{i+1}, F_i^{api})$ to denote the set of functions from M_{i+1} (M_1 when $j = n$), each of which calls one or more of the functions from F_i^{api} either directly or indirectly.

Step 4: we say that the cyclic module dependency is an *intra-layer* cyclic module dependency if, and only if, for each module $M_{i(i=1,..n-1)}$, $Callers(M_{i+1}, F_i^{api}) \cap F_{i+1}^{api} \neq \phi$, and $Callers(M_1, F_n^{api}) \cap F_1^{api} \neq \phi$.
Otherwise the cyclic module dependency is considered as *inter-layer* cyclic dependency.

Generally speaking, an intra-layer cyclic module dependency is caused by mutually recursive functions across modules, and intra-layer cyclic module dependency could be eliminated by merging those mutual-dependent functions into a single module.[2] An inter-layer cyclic module dependency, on the other hand, is usually caused by the coexistence of API functions that belong to different logical layers of the systems in the same module. This kind of cyclic module dependency can be removed by splitting the module into two, or more, so that each module only exports functions that belong to the same logic layer. For example, Figure 4 shows two contrived Erlang modules that are mutually dependent on each other. The dependency between modules can be represented as :

```
-module(m1).
-export ([foo/0, bar/0]).

foo() ->1.
bar() -> m2:blah().

-module(m2).
-export([blah/0]).

blah() -> m1:foo().
```

Figure 4. Cyclic module dependency

[2]There are, of course, exceptions to this: we might want to split a mutually recursive set of language processing functions into one module per language category, say.

$$m_1 \xrightarrow{F_1=\{foo/0\}} m_2 \xrightarrow{F_2=\{blah/0\}} m_1$$

This cyclic module dependency is treated as an inter-layer cyclic module dependency as:

$$Callers(m_1, F_2) \cap F_1 = \phi$$

It is obvious in this case that the cyclical dependency can be removed by splitting module `m1` into two modules, so that functions `foo/0` and `bar/0` are not in the same layer of the architecture.

4.3 Modules Serving Multiple Goals

Detection. Detecting modules serving multiple goals by static analysis is less straightforward simply because the service offered by a function is hidden in its implementation logic, and there is no standard way to measure the commonality of purpose between functions in a quantitative way.

Based on the observation that functions serving the same goal are more likely to share a number of things, including nodes in the function callgraph, data structures and macros as well as words used in function, variable, or process names, similarity metrics based on these features could serve as an indicator as to whether two functions, or two function groups, share similar goals.

We make use of an existing agglomerative hierarchical algorithm [12, 8] to cluster the functions exported by a module into clusters based on the similarity metrics between functions clusters. Clustering is a key technique used in reverse engineering to gather software components into modules that might well be considered significant to the software engineers who designed the original system or those who will have to work with the results. To evaluate the similarity score between two functions, we represent each function by a feature list. The four features we choose are:

- Calls to functions, which corresponds to those nodes in the function callgraph of the system that are reachable from the function under consideration.
- Use of records, which is the only data structure that can be named in Erlang.
- Use of macros.
- Reference to words. Identifiers, including function, module, process and variable names, used in a function definition are decomposed into words.

Given two functions, or function clusters, and their list of entity references, X and Y say, their similarity score is calculated using the Jaccard similarity metrics [11] as follows,

$$sim(X,Y) = a/(a+b+c)$$
where $a = |X \cap Y|$, $b = |X \setminus Y|$, and $c = |Y \setminus X|$

An agglomerative hierarchical algorithm starts from the individual entities, gathers them into small clusters which are in turn gathered into larger clusters up to one final cluster that contains every entity. The result is a binary tree of clusters. However, if the aim is to detect modularity smells, there is no need to continue the clustering process until there is only one cluster left. In fact, we only group two clusters into one if their similarity score is above a specified threshold, and the clustering process stops when there are no more clusters whose similarity scores are above the threshold.

The clusters generated are then further analysed regarding to the size of the clusters, and their usage by the client modules. A ‘multi-goal module’ smell is only reported if we find more than one cluster, and each of them is of a reasonable size, i.e. the number of lines of code contained is greater than a threshold given.

Elimination. A ‘multi-goal module smell’ can be eliminated by partitioning the module into two or more parts based on the number of clusters reported. With Wrangler’s support for moving functions from one module to another, which we will discuss in more detail in Section 5, splitting a module is straightforward. All the user needs to do is to select the functions that are to be moved to another module, and provide Wrangler with the target module name, which in general is a fresh module name.

4.4 Very Large Modules

The number of lines in a module is the major factor used to identify very large modules. In general, it is likely that a large module is providing too many services to the rest of the system, and therefore should be partitioned into smaller modules using the cluster techniques discussed in Section 4.3; however, it is also possible that a module containing a monolithic piece of code serves only one goal; in this case, it still could be possible to extract one or more sub-components of the module into another module, so that the module size and complexity can be reduced.

To guide Wrangler’s searching for sub-components to be moved to anther module, two parameters can be specified:

- the minimal number of lines of code, and
- the maximal number of functions shared between the sub-components and the rest of the module.

5 Refactoring Support

The most important refactoring when refactoring module structure is concerned is moving functions from one module to another. For this purpose, we have extended Wrangler’s original refactoring for moving a *single* function between modules, so that a collection of functions can be moved in one single step. Moreover, in the case that a function to be moved depends on other functions defined in the original module, those functions are also automatically moved to the target module if no other functions in the original module depend on them. The target module can be an existing module or a new module to be created.

The refactoring *move functions from one module to another* is complex because both the pre-condition checking and program transformation involved are nontrivial. For this refactoring to be behaviour preserving, various pre-conditions need to be checked before the program can be actually transformed. For example, the refactoring needs to make sure that the functions to be moved do not conflict with the existing functions in the target module, also macros and records used by the functions to be moved should not be defined differently in the target module, and so forth. The program transformation step needs not only to remove the functions from the original module, to add them to the target module, but also needs to check the call sites of these functions – potentially across the whole system when the functions are exported – to make sure that all the references to the functions are changed to use the target module as the defining module of functions moved.

Tool support for this kind of complex, though elementary, refactorings is essential, as both manual program analysis and transformation are tedious and error prone.

Apart from *move functions from one module to another*, other refactorings from Wrangler can also help with the modularity maintenance process. These include renaming of module and function names, cleaning up the module export lists, cross module duplicated code detection and elimination [4].

6 A Case Study

To examine the usefulness of Wrangler's support for modularity maintenance, we applied the tool to the Wrangler system itself, a number of other open-source Erlang systems, as well as some industrial code from Ericsson, Sweden. In this paper we mainly discuss the results of applying the tool to Wrangler itself. As authors of most of the code in Wrangler, we can play the role of domain experts in judging the usefulness of the results returned by the tool.

The version of Wrangler we used in the case study is Wrangler-0.8.7, which consists of 56 Erlang files and 40K lines of code, comments included. Due to the compactness of program written in functional programming languages, Wrangler is by no means a small Erlang program.

Along with the development of Wrangler's modularity maintenance support, substantial number of structural refactorings have been made to Wrangler after the release of Wrangler-0.8.7. In what follows, we discuss the case study results for each modularity smell in the same order as they are introduced in the previous sections.

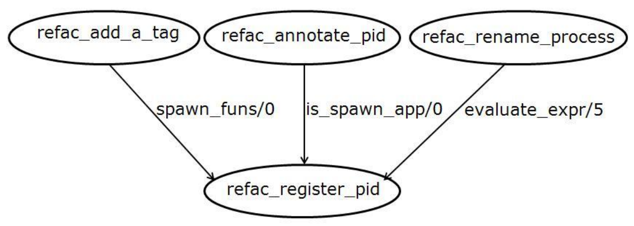


Figure 5. Improper inter-module dependency

6.1 Improper Inter-module Dependency

With an *API Score* threshold of 0.4, Wrangler reports 13 *improper inter-module dependencies*. We examined each of these and concluded that 11 of them should be removed. The remaining 2 involve sharing of functions between two different versions of the Erlang tokenizer, which we decided to leave them as they are.

Most of the 11 dependencies that we chose to remove were caused by sharing of functionalities between the implementation of different refactorings. For instance, Figure 5 shows 3 of the improper inter-module dependencies reported involving 4 modules. Among the four modules shown in the graph, three of them (`refac_add_a_tag`, `refac_rename_process` and `refac_register_pid`) each implements a single refactoring, whereas `refac_annotate_pid` is a infrastructure module providing services to be used by the other three.

The module graph shows that three non-API functions defined in module `refac_register_pid` are exported by it and used by other modules. As a matter of fact, the three non-API functions are all the non-API functions exported by that module. These inter-module dependencies are clearly against the authors' intention, as ideally a module implementing a refactoring should only export functions that serve as refactoring commands, and there should be no dependency between modules implementing different refactorings (whenever it is possible).

The dependency between `refac_annotate_pid` and `refac_register_pid` is even more undesirable, as it actually constitutes a cyclic dependency between the two modules.

For the module dependencies shown in Figure 5, the three refactoring commands suggested by Wrangler are:

```
move_fun(refac_register_pid,[{evaluate_expr,5}],
    [refac_util,refac_syntax,
     refac_annotate_pid]).
move_fun(refac_register_pid,[{is_spawn_app,1}],
    [refac_annotate_pid]).
```

```
move_fun(refac_register_pid,[{spawn_funs,0}],
    [refac_util,refac_syntax,
     refac_annotate_pid,refac_syntax_lib]).
```

Take the first refactoring command as an example, it suggests to move the function evaluate_expr/5 defined in module refac_register_pid to one of the modules given in the list. When multiple target modules are suggested by Wrangler, the user has to select one target module, and remove the others from the list. Of course, if none of the modules suggested makes sense to the user, he or she could always specify another existing module or a completely new module. For the three refactorings above, we chose the first module suggested as the target module.

Performing these refactorings in either an IDE or directly on the command line is straightforward with Wrangler's refactoring support.

6.2 Cyclic Module Dependency

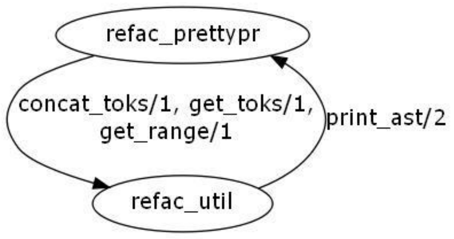
After having removed the inter-module dependencies reported, we applied Wrangler's cyclic module dependency detection to Wrangler-0.8.7, and this reveals 8 cyclic module dependencies, among which one is reported as intra-layer cyclic module dependency, and the other seven are reported as inter-layer cyclic module dependencies. Two of the cyclic module dependencies reported consist of 3 modules, and all the others consist of two modules. Our manual inspection of the results reported completely agrees with Wrangler's automatic intra-layer/inter-layer classification result.

Most of the 8 cyclic module dependencies reported involve a module named refac_util, which provides a collection of utility functions to the other parts of the system. However, over years of development, too much functionality has been added to this module, some functions no longer qualify as utility functions.

For instance, Figure 6 shows one of the cyclic module dependencies reported, and the refactoring command suggested by Wrangler. What the refactoring command says is that the four functions enclosed in the list defined in module refac_util should be moved into a separate module, and the user needs to choose the module name, which in general is a fresh module name, and substitute it for user_supplied_mod_name. Indeed, after the release of Wrangler-0.8.7, this module has been refactored significantly, and the single big module has been divided into 5 smaller modules, each of which provides a specific kind of service.

6.3 Modules Serving Multiple Goals

With a Jaccard similarity threshold of 0.2, we applied Wrangler's multi-goal module detector to Wrangler-0.8.7, and this process reported that 12 of the 56 modules serve multiple goals.



```
move_fun(refac_util,
    [{write_refactored_files,1},
     {write_refactored_files,3},
     {write_refactored_files,4},
     {write_refactored_files_for_preview,2}],
     user_supplied_target_mod_name).
```

Figure 6. Cyclic module dependency

The clustering result provides valuable information regarding the commonality between different functions in each module, however given the fact that there is not standard way to compare the goals of different functions, the user still has to experiment with different threshold values, and inspect the results using his or her domain knowledge to decide what to do next.

For example, for the module refac_syntax_lib from Wrangler, which is a modified version of the erl_syntax_lib module from Erlang Syntax Tools library [2], seven clusters were reported, as shown in Figure 7. Together with each cluster, *InDegree* represents the number of modules that make use of the functions from the cluster, and *OutDegree* represents the number of modules on which the cluster depends. Clearly, this module provides functionalities that cover a number of themes including functionalities for abstract syntax tree (AST) traversal, for AST annotation, for AST analysis, etc. It might not be a good idea to put each cluster into a separate module, because some of the clusters are actually too small to form a new module, however, it would be preferable to move cluster 1, probably also cluster 2, to a separate module especially designed for AST traversal APIs, because of the large number of modules that depend on it, and the clearness of the service it provides.

6.4 Very Large Modules

The average module size of Wrangler is 450 lines of code, comments excluded. In general, a module implementing a refactoring only contains the implementation of

```
Module: refac_syntax_lib
Cluster 1, Indegree:25, OutDegree:1,
[{map,2}, {map_subtrees,2},
 {mapfold,3},{mapfold_subtrees,3},
 {fold,3}, {fold_subtrees,3}]

Cluster 2, Indegree:0, OutDegree:0,
[{foldl_listlist,3},{mapfoldl_listlist,3}]

Cluster 3, Indegree:0, OutDegree:0,
[{new_variable_name,1},{new_variable_names,2},
 {new_variable_name,2},{new_variable_names,3}]

Cluster 4, Indegree:4, OutDegree:1,
[{annotate_bindings,2},{annotate_bindings,3},
 {var_annotate_clause,4},{vann_clause,4},
 {annotate_bindings,1}]

Cluster 5, Indegree:4, OutDegree:1,
[{analyze_function_name,1},
          ...13 items omitted here
 {analyze_attribute,1}]

Cluster 6, Indegree:0, OutDegree:1,
[{to_comment,1},{to_comment,2},
 {to_comment,3}]

Cluster 7, Indegree:0, OutDegree:1,
[{limit,2},{limit,3},
 {function_name_expansions,1}]
```

Figure 7. Clusters identified

a single refactoring, and only the refactoring command is exported by that module. However, it is not uncommon that a module gets too big because of the complexity of the refactoring implemented, and in this case, extracting a sub-component from the implementation into a separate module is the general practice. Wrangler's support for automatic searching of sub-components proved to be very helpful.

7 Implementation Considerations

Erlang is a general-purpose functional programming language. While Erlang shares some basic properties, such as *referential transparency*, with other functional programming languages, it also has its own characteristics and programming idioms, which we need to address when building Erlang-specific program analysis tools in general. Since the work investigated in this paper is built on top of Wrangler's program analysis and transformation infrastructure, and most of the issues were already handled by Wrangler, we only summarise the major issues here without going into details.

- In Erlang, function and module names are normal Erlang atoms, and can be generated dynamically. Moreover, a function can be called using the built-in function `apply/3` in the form of `apply(Module, Function, Args)`, where `Module` and `Function` are expressions that evaluate to an Erlang atom, and `Args` is an expression that evaluates to a list of terms. These features make the generation of function callgraph and module graph more complex than expected, and data flow analysis techniques are needed for generation of accurate, or nearly accurate, function callgraphs and module graphs.
- Erlang is a programming language with built-in support for concurrency. In Erlang, functions can commute with each other in two different ways, that is *parameter passing* and *message passing*. The primitives **spawn**, "**!**" (send) and **receive** allow a process to create a new process and communicate with other processes through asynchronous message passing. When message passing comes into play, traditional callgraph-based program analysis has to been extended to take the process structure information into account.
- OTP behaviour callback modules. Erlang comes with the Open Telecom Platform (OTP) middleware platform, which provides a number of ready-to-use behaviours, such as finite state machines, generic servers, etc, embodying a set of design principles for Erlang systems. To use these components, the user has to define a behaviour callback module and implement a number of pre-specified callback functions. Static analysis of callback modules in a normal way could produce results below expectation due to the fact that the actual interaction between functions in the module are hidden away in the components from Erlang.

8 Related Work

Various approaches have been proposed in the literature on system (re)modularization, with *software clustering* [12] being the most commonly used approach. Clustering algorithms model the similarity between entities in a quantitative way, and group entities that are similar together. Clustering-based software (re)modularization approaches differ mainly in their choice of three parameters: how the entities are described, how the similarity metrics between the entities is computed and what clustering algorithm is used. A comparative study of the influence of different parameter choices on the clustering results when doing software (re)modularization has been done by N. Anquetil et. al. [8], and their experimental results gave us insight into the choice of the clustering algorithm used by Wrangler to detect modules with multiple goals.

The work most closely related to ours is the Erlang refactoring tool, RefactorErl [7], developed by researchers at the Eötvös Loránd University in Budapest, Hungary. Like Wrangler, RefactorErl is also a general refactoring tool for Erlang. Unlike Wrangler, which use abstract syntax tree (AST) as the internal representation of Erlang programs, RefactorErl follows a different approach to refactoring and works by creating a formal semantical graph model from Erlang source code and storing this graph in a relational database. RefactorErl also provides support for refactoring the module structure of an existing Erlang application, however unlike Wrangler's modularity smell directed incremental modularity maintenance, RefactorErl's support for module restructure is solely based on clustering techniques, and is mainly used to split a large software into smaller loosely coupled components.

In [10], G.M.Rama also proposed the idea of refactoring based modularity improvement, targeting at programs written in imperative or OO programming languages. While there is some overlapping between the modularity smells detected by Rama and us, different techniques were used to detect these modularity smells, and apart from that, refactoring support for modularity smell elimination lies in Rama's future work.

9 Conclusions and Future Work

In this paper, we have identified a number of modularity smells that are common in programs written in Erlang, and also presented techniques taken to support automatic detection and semi-automatic elimination of those modularity smells. The tool is built on top of the infrastructure of Wrangler, a general purpose refactoring tool for Erlang, and also integrated within the Wrangler environment. Instead of carrying out fully-automatic program restructuring – which could produce a program that is too different from the original one to be recognised and accepted by the user – we aim to help the user to identify and solve existing modularity flaws in a step-by-step way, so that the user can justify, and be fully aware of, the changes made to the system. The tool is designed to be regularly used during the software development process, so that modularity smells can be detected and eliminated early. Case studies carried out with real-world code demonstrated the usefulness of the tool.

Our future work will go in a number of directions. We are going to do more empirical studies of modularity smells from different Erlang systems, and extend the tool to help the detection and elimination of more modularity smells. Although Erlang is a relatively simple programming language, the concepts presented in this paper would also be useful in attacking the same problem for other languages, given the fact that moving functions from module to module is a common refactoring across a number of programming languages. To justify this, we would like to explore the application of the approach investigated here to other function programming languages like Haskell, which has a more complex module system than Erlang.

10 Acknowledgements

This research is supported by EU FP7 collaborative project ProTest (`http://www.protest-project.eu/`), grant number 215868; we thank our funders and colleagues for their support and collaboration.

References

[1] J. Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007.

[2] R. Carlsson. Erlang Syntax Tools. `http://www.erlang.org/doc/apps/syntax_tools/`, 2004.

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

[4] H. Li and S. Thompson. Similar Code Detection and Elimination for Erlang Programs. In M. Carro and R. Pena, editors, *Practical Aspects of Declarative languages 2010*, LNCS, pages 104–118. Springer, January 2010.

[5] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, and T. Nagy. Refactoring Erlang Programs. In *EUC'06*, Stockholm, Sweden, November 2006.

[6] H. Li, S. Thompson, G. Orosz, and M. Töth. Refactoring with Wrangler, updated. In *ACM SIGPLAN Erlang Workshop 2008, Victoria,Canada*, 2008.

[7] L. Lövei, C. Hoch, H. Köllő, T. Nagy, A. Nagyné-Víg, D. Horpácsi, R. Kitlei, and R. Király. Refactoring Module Structure. In *Proceedings of the 7th ACM SIGPLAN workshop on Erlang*, Victoria, Canada, Sep 2008.

[8] C. F. Nicolas Anquetil and T. C. Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, Washington, DC, USA, 1999.

[9] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[10] G. M. Rama. A Desiderata for Refactoring-Based Software Modularity Improvement. In *Third India Software Engineering Conference*, Los Alamitos, CA, USA, 2010.

[11] P. H. SNEATH and R. R. SOKAL. *Numerical Taxonomy.* Series of books in biology. W.H. Freeman and Company, San Francisco, 1973.

[12] T. A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 33, Washington, DC, USA, 1997.

[13] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall, Inc., NJ, USA, 1979.