Refactoring Module Structure

László Lövei Csaba Hoch Hanna Köllő Tamás Nagy
Anikó Víg Dániel Horpácsi Róbert Kitlei Roland Király

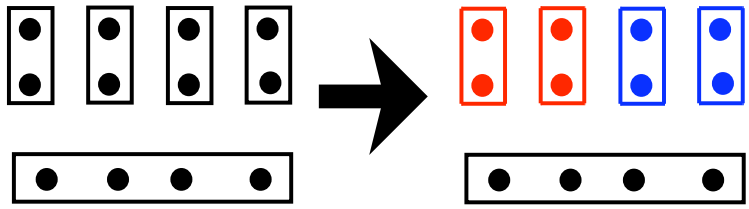
Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest

ACM SIGPLAN Erlang Workshop, 2008

Outline

Motivation

- We have a complex Erlang software
  - Consists of modules and functions
- Over time, it has grown to be even more complex
  - Maintenance became nearly impossible
- The modules should be grouped into blocks that are small enough to be maintained effectively



RefactorErl, an Erlang Refactoring Tool

- Refactoring is meaning-preserving source code transformation
- RefactorErl is a tool that refactors Erlang source code
- Refactoring needs static semantical analysis . . .
- . . . and module restructuring needs the same!

RefactorErl, an Erlang Refactoring Tool

- Refactoring is meaning-preserving source code transformation
- RefactorErl is a tool that refactors Erlang source code
- Refactoring needs static semantical analysis . . .
- . . . and module restructuring needs the same!

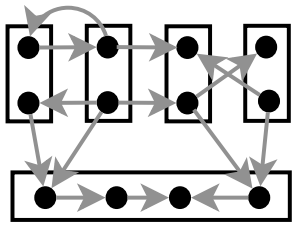
Idea

Use the infrastructure of RefactorErl for module restructuring

Module Restructuring Steps

1. Analysis
2. Clustering
3. Result selection
4. Splitting files
5. Transformation

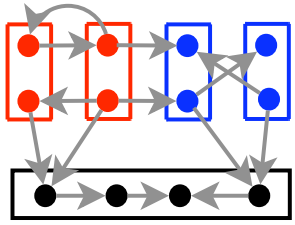
- Required information:
  - Function call graph
  - Record usage
- Provided by RefactorErl



Module Restructuring Steps

1. Analysis
2. Clustering
3. Result selection
4. Splitting files
5. Transformation

- Group related modules
  - Call each others functions
  - Use the same record
- Many results with different number of groups
- Many possible parametrization



Module Restructuring Steps

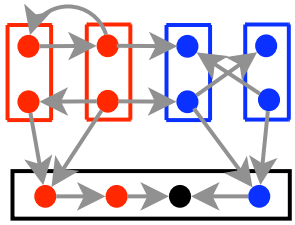
1. Analysis
2. Clustering
3. Result selection
4. Splitting files
5. Transformation

- Measure which is the best clustering
- Automatic selection from a big result set
- Uses the same analysis results

Module Restructuring Steps

1. Analysis
2. Clustering
3. Result selection
4. Splitting files
5. Transformation

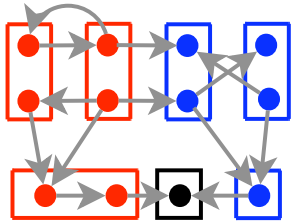
- Library modules: many incoming calls
  - Filtered before clustering
  - Used from more clusters
- Parts used from only one cluster are selected
- The same is done for header files



Module Restructuring Steps

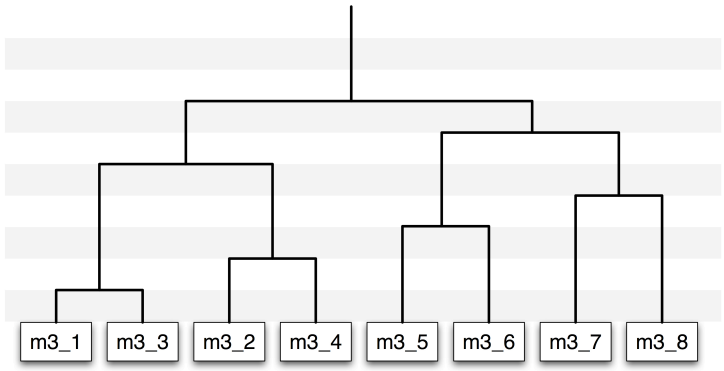
1. Analysis
2. Clustering
3. Result selection
4. Splitting files
5. Transformation

- The physical splitting of files is a refactoring
- Needed refactoring steps:
  - Move function definition
  - Move record definition
  - Move macro definition



Clustering

- Modules are sorted into clusters
- Hierarchical clustering algorithm



- The clustering algorithm can be parametrized with functions
  - e.g. how to compute the "distance" of two clusters

Fitness function

- Different clusterings are rated by a fitness function
- We used the MQ fitness function: the more internal and less external connection a clustering has, the better it is

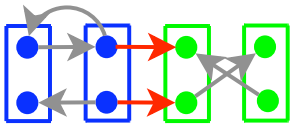


Figure: Clustering with better fitness value

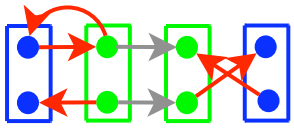
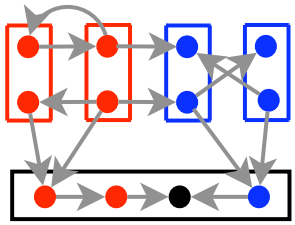


Figure: Clustering with worse fitness value

Splitting

- After a clustering is chosen, we have:
  - Clusters of modules
  - Stand-alone library modules
- Goal: splitting the library modules into smaller parts
  - Functions, records, and macros are assigned to clusters based on exclusive usage
  - Simple graph algorithm



Results with Industrial Software

- 188K lines of code, 6104 functions in 106 modules
- Clustering takes about 2 minutes – lots of parameters have been possible to try
- The resulting 4 clusters needed small manual corrections
- Module and header splitting, dead code elimination

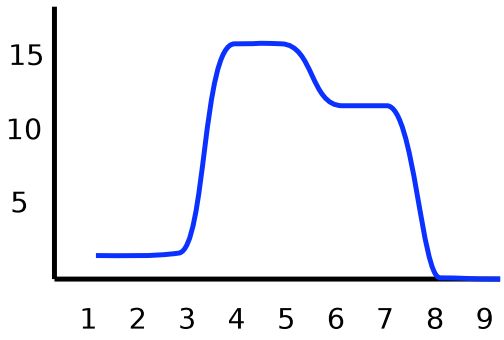


Figure: MQ fitness values for cluster numbers

Summary

- Developed a module restructuring method based on RefactorErl
- Successfully applied it on industrial software

Plans:

- Tool support for applying the results
- Clustering functions of one module