Type: Weekly reading report **Date:** June 7, 2017

Paper: [Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation] Christian Kaestner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. *In Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (Portland, OR), New York, NY, October 2011. ACM Press.*

Summary: In this paper, authors proposed a novel variability-aware parser that can parse almost all unpreprocessed code without heuristics in practicable time.

What problem(s) are they solving? In practice, it is unrealistic to parse code without preprocessing it. This paper proposed a variability-aware parser to parse, analyse, and process unpreprocessed code to further support variability-aware error detection, program understanding, reengineering, refactoring and other code transformations.

Motivation(Why are these problems important): Current attempts to parse un-preprocess C code are:

- **unsound:** A variability-aware parser is unsound if it produces a parse result which does not correctly represent the result from preprocessing and parsing in all variants. (***not*** *collect the results from all variants*)
- **incomplete:** A variability-aware parser is incomplete if it rejects a code fragment even though preprocessing and parsing would succeed for all variants. (***Reject code without preprocessing*** *even if there is one valid configuration*);
- **exponential explosion:** when the number of feature is large.

What is the contribution of the work?(i.e. what is interesting or new?) The main contribution of this work is a variability-aware parsing framework.

- Comparing with three common strategies to parse un-preprocessed C code:

  brute force: parse and analyze the preprocessed variants in isolation, it suffers from exponential explosion and quickly becomes infeasible in practice when the number of features grows;(break **exponential explosion**)

  manual code preparation: support only a subset of possible preprocessor usage;(break **complete**)

  heuristics and partial analysis: exploit repeating patterns and idioms, such as the common include-guard pattern or capitalised letters for macro names;(break **soundness**)

What methods are they using?

Variability-aware Lexer

Step 1. Conditional-token Stream

1. Token-based lexer;
2. Each token has a *presence condition*; resulting in *conditional-token stream*;
3. *Presence condition* serves as the **subscript** to tokens, and separate tokens by "$\cdot$" and empty token sequences as "$\emptyset$". For example, a conditional-token stream "$3 \cdot * \cdot 7 \cdot + \cdot 1_A \cdot 0_{\neg A}$"

Step 2. File inclusion and macros

1. When including a header file, continue reading tokens from that file. *Only when there is a macro, look up for expansion in the header file.*
2. When reading a macro token <=> expand it based on the macro;
3. **Multiple definitions of a macro** <=> return all possible expansions with corresponding presence conditions. For example, a conditional-token stream with macro expansion :

$$3 \cdot *_X \cdot 7_X \cdot + \cdot 1_{Y \wedge Z} \cdot 4_{Y \wedge \neg Z} \cdot *_Y \cdot 1_{Y \wedge X} \cdot 2_{Y \wedge \neg X} \cdot 0_{\neg Y}$$

A Library of Variability-aware Parser Combinators

1. Transfer the *conditional token stream* into abstract syntax tree;
2. "$\diamond$" to represent a `Choice`;
3. **Split** the parser context on conditional tokens (split when necessary);
4. **Join** the parser contexts again to produce choice nodes in the abstract syntax tree (join early to avoid parsing tokens repeatedly);
5. **Context Splitting.**
   use the function `next` to determine whether there is a need on splitting
   | next: VParser[Token]
   | next$(ctx, \emptyset)$ = FAIL<"unexpected end of file">
   | next$(ctx, t_{pc} \cdot rest)$ =
   | $--$ SUCC$< t_{pc}, rest >$, if ctx implies pc
   | $--$ next$(ctx, rest)$ =, if ctx contradicts pc
   | $--$ SPLIT
6. **Filtering.**
   `Filter` function that checks all successful parse results from next and replaces them by failures in case the expected token does not match.

7. **Joining contexts.**
   Joining of parse results that attempts to join the results of another parser. The key idea is to move variability out of a split parse result into the resulting abstract syntax tree.

8. **When to join?**
   Joining after typical fine-grained program structures seems to be a good balance between computation overhead and low amount of repeated parsing.

9. **Sequencing**
   A sequence parser combinator ($p \sim q$), The produced parser continues all successful results (and only successful results) of the first parser with the second parser.

10. **Alternatives**
    The parser combinator for alternatives ($p \mid q$), it replaces all failures with the result of the second parser, called with the corresponding context.

11. **Repetition**
    Parser combinators for repetition (p or p^+) can be constructed with sequencing and alternatives.

12. **Function application**
    This paper adopts **Repetition** combinators.
    - Split parser context as late as possible and provide facilities to join parser context early.
    - If we attempt to join after each parsing step, we guarantee to consume each token only once in token streams with only disciplined annotation.
    - Although we need to reason about the relationship be- tween parser context and presence condition for every single token, this can be done efficiently with contemporary SAT solvers, even for complex formulas with hundreds of features.

Summary

Would you have solved the problem differently? [TODO]

Paper: [A Variability-Aware Module System] Christian Kstner, Klaus Ostermann, and Sebastian Erdweg.*In Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 773–792, New York, NY: ACM Press, October 2012.*

Summary:

What problem(s) are they solving?

Motivation(Why are these problems important):

What is the contribution of the work?(i.e. what is interesting or new?)

What methods are they using?

Would you have solved the problem differently? [TODO]

Do all the pieces of their work fit together logically?

Paper: [Scalable Analysis of Variable Software] J. Liebig, A. von Rhein, C. Kstner, S. Apel, J. Drre, and C. Lengauer.*In Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), New York, NY: ACM Press, August 2013*

Summary: In this paper, authors build a *module system* to cope with scale problem in real-world projects.

What problem(s) are they solving? In this paper, it try to solve the problems in following perspectives:

- sampling and variability-aware analysis(Variable abstract syntax tree);
- scalable variability-aware analysis;
- variable control-flow graph;
- variability-aware liveness analysis;
- variability-aware type checking;

Motivation(Why are these problems important): Previous work, including parsing, type checking, data-flow analysis, model checking, and deductive verification. While these work are promising, variability-aware analyses have not been applied to large-scale, real-world systems so far; previous work concentrated mostly either on formal foundations or is limited with respect to practicality.

What is the contribution of the work?(i.e. what is interesting or new?) Comparing with the sampling analysis

1. **Single-conf heuristic.** to analyze only a single representative variant that enables most.
2. **Random heuristic.** a simple approach to select samples is to generate them randomly. For *n* features, make *n* random, independent decisions whether to enable the corresponding configuration option.
3. **Code-coverage heuristic.** select a minimal sample set of variants, such that every lexical code fragment of the systems' code base is included in, at least, one variant.
4. **Pair-wise heuristic** using the pair-wise heuristic, the sample set contains a minimal number of samples that cover all pairs of configuration options, whereby each sample is likely to cover multiple pairs.

What methods are they using? Variable abstract syntax trees

1. Add `Choice` node expresses the choice between two or more alternative subtrees;
2. `Choice(Prop,A,B)` to represent the with the condition `Prop`, if `Prop` is enabled, it will jump to A, otherwise B;

Variability-aware type checking

1. Extend the symbol table, a symbol is not mapped to a single type, it map to a conditional type;
2. During expression typing, assign a variable type (choice of types) to each expression, where already looking up a name in a symbol table may return a variable type;
3. Use variability model(if available) to filter all type errors that occur only in invalid variant;

Variable control-flow graphs

1. define a variability-aware successor function that may return different successor set for different variants, or, equivalently, but with more sharing.
2. using the result of successor function, we determine for every possible successor, a corresponding presence condition, which we store as annotation of the edge in the variable CFG.

Variability-aware liveness analysis

1. `uses` computes all variables read;
2. `defines` computes all variables written to;
3. based on `uses` and `defines` to compute the live variables and along with presence conditions

Principle: Keeping variability local

1. *late splitting*: perform the analysis without variability until we encounter it.
2. *local variability representation*: keeping variability local in intermediate results. Instead of copying the entire symbol table for a single variable entry, we have only a single symbol table with conditional entries.
3. *early joining*: attempts to join intermediate results as early as possible.

Paper: [Mining Configuration Constraints: Static Analyses and Empirical Results] S. Nadi, T. Berger, C. Kstner, and K. Czarnecki.*In Proceedings of the 36th International Conference on Software Engineering (ICSE), pages 140–151, June 2014.*

Summary: Extracting the configuration constraints from the source code and mapping the constraint from feature model.

Motivation(Why are these problems important):

- Many systems have no documented variability model or rely on information textual descriptions of constraints;
- Identifying the sources of configuration constraints is essential to support automatically creating variability models.

What is the contribution of the work?(i.e. what is interesting or new?)

- an extension and combination of existing analyses to extract configuration constraints;
- a novel constraint extraction technique based on feature use and code structure;
- a quantitative study of the effectiveness of such techniques to recover constraints;
- a qualitative study of sources of constraints in existing models.

What methods are they using?

Problem Space enforced configuration constraints can stem from technical restrictions present in the solution space such as dependencies between two code artifacts.

- Constraints from the existing variability models;
- | - hierarchy constraints
- | - cross-tree constraints

Solution Space extract global configuration constraints from the code (in all variants, any configuration *violating* this constraint is *ill-defined* by some specification)

- **Build-time errors**: Every valid configuration of the system must not contain build-time errors, such that it can be successfully preprocessed, parsed, type checked, and linked.
- **Feature Effect**: Every valid configuration of the system should yield a lexically different program.

Build-time errors

- **Preprocessor Errors**: test #ERROR directives and extract the relation(configuration) reach this directive;
- **Parser Errors**: syntax error, but TypeChef reports syntax error with corresponding presence condition;
- **Type Errors** report type error with presence condition;
- **Constraints** == extracted from each error(preprocessor, parser, type error)
- **Linker Constraints** (1) check only linkage within the application and discard symbols defined in libraries (2) extract global across file constraints, derive linker errors and corresponding constraints;

Feature effects

- **feature effect** detecting nesting among #IFDEFs, that is, a nested feature should not be selected without the outer feature, then create a constraint.
- (1) *collect all unique PCs* of all code fragments
- (2) compute a feature's effect. $==>$ if a feature $f \in F$ has an no effect in a PC ϕinP if $\phi[f \leftarrow True]$ is equivalent to $\phi[f \leftarrow False]$. Otherwise, using $\phi[f \leftarrow True] \oplus \phi[f \leftarrow False]$ to show substituting f is different.
- (3) from (2), we can extract the constraints like: $f \rightarrow \vee\phi\,[f \leftarrow True] \oplus \phi\,[f \leftarrow False]$;

Would you have solved the problem differently? [TODO]

Do all the pieces of their work fit together logically?

Paper: [Variational Data Structures: Exploring Tradeoffs in Computing with Variability] E. Walkingshaw, C. Kstner, M. Erwig, S. Apel, and E. Bodden.*In Proceedings of the 13rd SIGPLAN Symposium on New Ideas in Programming and Reflections on Software at SPLASH (Onward!), pages 213–226, New York, NY: ACM Press, 2014.*

Summary: First research on variational data structures will benefit not only customisable software, but many other application domains that must cope with variability.

What problem(s) are they solving?

Motivation(Why are these problems important):

What is the contribution of the work?(i.e. what is interesting or new?)

What methods are they using?

Would you have solved the problem differently? [TODO]

Do all the pieces of their work fit together logically?

Paper: [Morpheus: Variability-Aware Refactoring in the Wild] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer.*In Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE), pages 380391. IEEE Computer Society, May 2015.*

Summary:

What problem(s) are they solving?

Motivation(Why are these problems important):

What is the contribution of the work?(i.e. what is interesting or new?)

What methods are they using?

Would you have solved the problem differently? [TODO]

Do all the pieces of their work fit together logically?

Paper: [Partial Preprocessing C Code for Variability Analysis] Christian Kstner, Paolo G. Giarrusso, and Klaus Ostermann*In Proceedings of the Fifth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS) (Namur, Belgium), pages 137-140, New York, NY, USA, January 2011. ACM Press.*

Summary:

What problem(s) are they solving? It is hard to analyze code that was not already preprocessed

Motivation(Why are these problems important): - It is hard to process C code due to preprocessor:

- The C preprocessor is token-based and uses lexical macro and no mechanism to detect potential problem in the underlying code.(**Token based no error detection**)
- Deeply interwined with the file-inclusion mechanism (#include) and macro facilities(#define and #undef)(**A lots interaction**)
- Conditional compilation is not only used for variability implementation also used for include guards(**Distinguish macro for variability and normal issue**)

What is the contribution of the work?(i.e. what is interesting or new?)

- A comprehensive description of the problem of analysing variability in pre-cpp code;
- partially preprocessing code by evaluation file inclusion and macros but not conditional compilation;
- Implementation on top of C preprocessor;
- Evaluation on top of a C preprocessor;

What methods are they using?

- **Presence condition**:

Would you have solved the problem differently? [TODO]

Do all the pieces of their work fit together logically?

Paper: [Presence-Condition Simplification in Highly Configurable Systems] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger. *In Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE), pages 178188. IEEE Computer Society, May 2015.*

Summary:

What problem(s) are they solving?

Motivation(Why are these problems important):

What methods are they using?

Would you have solved the problem differently? [TODO]

Do all the pieces of their work fit together logically?

Paper: [TypeChef: Toward Type Checking #ifdef Variability in C] Andy Kenner, Christian Kstner, Steffen Haase, and Thomas Leich.*In Proceedings of the Second Workshop on Feature-Oriented Software Development (FOSD) (Eindhoven, The Netherlands), pages 25-32, New York, NY, USA, October 2010. ACM Press.*

Summary:

What problem(s) are they solving? An initial solution for type checking. To guarantee that all potential variants of a product line are well-typed without generating all variants. (check types before running the preprocessor with a specific feature combination)

Motivation(Why are these problems important):

What is the contribution of the work?(i.e. what is interesting or new?)

- outline the difficulties of product-lineaware type checking
- an initial solution with TypeChef
- partial preprocessor to handle file inclusion and macro substitution
- how TypeChef can detect inconsistencies

What methods are they using?

Partial Preprocessor

- A *partial preprocessor*: combine macro expansion and file inclusion;
- *include guard* is not considered as variability factor;

Expansion to Disciplined Annotations In contrast, TypeChef considers conditional compilation directives at finer granularity or around partial elements as undisciplined. In general it is always possible to expand undisciplined annotations to disciplined ones.

Parsing Implement a parser with the parser generate ANTLR based on an existing GNU C grammar.

Reference Analysis

- TypeChef looks up references that should be checked.
- cond', create a set of formulas that we can later feed into a solver;
- cond',

Would you have solved the problem differently? [TODO]

Do all the pieces of their work fit together logically?

References