Parsing Variant C Code: An Evaluation on Automotive Software

Robert Heumüller
Universität Magdeburg
Magdeburg, Germany
Robert.Heumueller@st.ovgu.de

Jochen Quante and Andreas Thums
Robert Bosch GmbH, Corporate Research
Stuttgart, Germany
{Jochen.Quante, Andreas.Thums}@de.bosch.com

Abstract

Software product lines are often implemented using the C preprocessor. Different features are selected based on macros; the corresponding code is activated or deactivated using `#if`. Unfortunately, C preprocessor constructs are not parseable in general, since they break the syntactical structure of C code [1]. This imposes a severe limitation on software analyses: They usually cannot be performed on unpreprocessed C code. In this paper, we will discuss how and to what extent large parts of the unpreprocessed code can be parsed anyway, and what the results can be used for.

1 Approaches

C preprocessor (Cpp) constructs are not part of the C syntax. Code therefore has to be preprocessed before a C compiler can process it. Only preprocessed code conforms to C syntax. In order to perform analyses on unpreprocessed code, this code has to be made parseable first. Several approaches have been proposed for that:

- **Extending a C parser.** Preprocessor constructs are added at certain points in the syntax. This requires that these constructs are placed in a way compatible with the C syntax. However, preprocessor constructs can be added *anywhere*, so this approach cannot cover all cases [1].

- **Extending a preprocessor parser.** The C snippets inside preprocessor conditionals are parsed individually, e. g., using island grammars [4]. This approach is quite limited, because the context is missing, which is often important for decisions during parsing.

- **Analyzing all variants separately and merging results.** This approach can build on existing analysis tools. However, for a large number of variance points, it is not feasible due to the exponential growth in the number of variants.

- **Replacing Cpp with a better alternative.** A different language for expressing conditional compilation and macros was for example proposed by McCloskey et al. [3]. Such a language can be designed to be better analyzable and better integrate with C. However, it is a huge effort to change a whole code base to a new preprocessing language.

We chose to base our work on the first approach. We took ANTLR's standard ANSI C grammar[1] and extended it by preprocessor commands in well-formed places. This way, we were already able to process about 90% of our software. In order to further increase the amount of successfully processable files, it was necessary to discover where this approach failed, and to come up with a strategy for dealing with these failures. An initial regex-based evaluation indicated that the two main reasons for failures were a) the existence of conditional branches with incomplete syntax units, and b) the use of troublesome macros.

2 Normalization

To be able to deal with incomplete conditional branches, we implemented a pre-preprocessor as proposed by Garrido et al. [1]. The idea is to transform preprocessor constructs that break the C structure to semantically equivalent code that fits into the C structure. The transformation basically adds code to the conditional code until the condition is at an allowed position. Figure 1 shows a typical example of unparseable code and its normalized equivalent.

The code is read into a tree that corresponds to the hierarchy of the input's conditional compilation directives. The normalization can then be performed on this tree using a simple fix-point algorithm:

1. Find a Cpp conditional node with incomplete C syntax units in at least one of its branches. "Incompleteness" is checked based on token black and white lists. For example, a syntactical unit may not start with tokens like `else` or `&&`.

2. Copy missing tokens from before/after the conditional into all of the conditional's branches. This way, some code is duplicated, but the resulting code becomes parseable by the extended parser.

3. Delete the copied tokens at their original location.

[1] `http://www.antlr3.org/grammar/list.html`

Original:	Normalized:	Pruned:
`#ifdef a`	`#ifdef a`	`#ifdef a`
`if (cond) {`	`  #ifdef a`	`  if (cond) {`
`#endif`	`    if (cond) {`	`     foo();`
`  foo();`	`      foo();`	`  }`
`#ifdef a`	`    }`	`#else`
`}`	`  #else`	`  foo();`
`#endif`	`    if (cond) {`	`#endif`
	`      foo();`	
	`  #endif`	
	`#else`	
	`  #ifdef a`	
	`      foo();`	
	`    }`	
	`  #else`	
	`    foo();`	
	`  #endif`	
	`#endif`	

Figure 1: Normalization and pruning example.

4. Repeat until convergence.

This step introduces a lot of infeasible paths and redundant conditions in the code. For example, the code in Figure 1 contains many lines that the compiler will never see – they are not reachable because of the nested check of the negated condition. Such infeasible paths may even contain syntax errors, like `foo();}` in the example. Such irrelevant parts are thrown away in a postprocessing step (*pruning*). It symbolically evaluates the conditions, identifies contradictions and redundancy, and removes the corresponding elements.

3 Macros and User-Defined Types

In unpreprocessed code, macro and type definitions are often not available. They are usually only resolved by included header files, and this inclusion is done by the preprocessor. Therefore, our parser cannot differentiate between macros and user-defined types or functions. Kästner et al. [2] have solved this problem by implementing a partial preprocessor that preprocesses #include and macros, but keeps conditional compilation. We decided to use a different approach: We added a further preprocessing step that collects all macro definitions and type declarations from the entire code base. This information is then used by the parser to decide whether an identifier is a macro statement, expression, or call, or whether it is a user-defined type. Additionally, naming conventions are exploited in certain cases.

4 Results

The approach was evaluated on an engine control software of about 1.5 MLOC. It consists of about 6,700 source files and contains about 150 variant switching macros. The following share of files could be successfully parsed due to the different parts of the approach:

- 90% could be parsed by simply extending the parser to be able to deal with preprocessor constructs in certain well-formed positions.
- 4% were gained by providing the parser with precollected macro and type information.
- 3% were gained by normalization.
- 1% was gained by adding information about type naming conventions.

In summary, the share of code that can now be parsed could be increased from 90% to 98% at an acceptable cost. This enables meaningful analyses, for example collecting metrics on variance. These can in future be used to come up with improved variance concepts. Another use case is checking if the product line code complies with the architecture. We also think about transforming `#if` constructs to corresponding dynamic checks to allow using static analysis tools like Polyspace on the entire product line at once.

References

[1] A. Garrido and R. Johnson. Analyzing multiple configurations of a C program. In *Proc. of 21st Int'l Conf. on Software Maintenance (ICSM)*, pages 379–388, 2005.

[2] C. Kästner, P. G. Giarrusso, and K. Ostermann. Partial preprocessing C code for variability analysis. In *Proc. of 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 127–136, 2011.

[3] B. McCloskey and E. Brewer. ASTEC: A new approach to refactoring C. In *Proceedings of the 13th Int'l Symp. on Foundations of Software Engineering (ESEC/FSE)*, pages 21–30, 2005.

[4] L. Moonen. Generating robust parsers using island grammars. In *Proc. of 8th Working Conference on Reverse Engineering (WCRE)*, pages 13–22, 2001.