Transformation of Preprocessor Variance to Post-Build Variance

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

Olga Pikuleva
Universität Tübingen
Tübingen, Germany
Olga.Pikuleva@student.uni-tuebingen.de

1 Introduction

Software product lines are widely used to efficiently develop similar variants of a software. In automotive software development, they are often implemented based on the C preprocessor. Even tools for model based development generate such code. The advantage of the preprocessor approach is that only the needed code ends up in the control unit. This way, resource consumption is minimized. On the other hand, for testing and creating meta program versions, post build variance is needed or at least easier to handle. *Meta program version* means that several of the possible static configurations are integrated in a single binary file, and the concrete configuration can be dynamically selected at engine start. We have developed an approach and tooling to transform existing C code that contains preprocessor variance to post-build variance code.

2 Example

For illustration of the desired transformation, take a look at the following piece of code:

```
#if SC1 && ((SC2 && (SC3>0)) || (SC4 && SC5))
  foo();
#else
  bar();
#endif
```

Now we want to create a meta program version for this from two customer projects. Unfortunately, system constants (i. e., special macros) SC1-3 each have different values in the two projects. However, SC4 and SC5 have the same values in both projects. To integrate both projects into one meta program version, we want to transform system constants SC1, SC2 and SC3 to runtime variables, while we want to keep SC4 and SC5 as build time variance points. Furthermore, we only want to have the code that is really required in the control unit due to limited resources. The result should look something like shown in Figure 1.

For preprocessor directives in declaration parts of the code, the transformation looks a bit different – since there is no `if` there. This means that *all* declarations that are guarded by system constants have to be kept. This in turn means that there may be multiple or conflicting declarations. In this case, identifiers have to be renamed – not only in the declaration, but also in all places where they are used.

```
if (SC1) {
  if (SC2 && (SC3 > 0)) {
    foo();
  } else {
    #if (SC4 && SC5)
      foo();
    #else
      bar();
    #endif
  }
} else {
  bar();
}
```

Figure 1: Transformed code.

3 Related Work

There has been quite some research on analysis and transformation of C code with preprocessor directives. However, there is no approach that supports our requirements. Most closely related is von Rhein's work [4]. He implemented such a transformation, but only supports a specific usage pattern of preprocessor variance – the one that is found in the Linux kernel. It requires configurations and only supports binary system constants (`#ifdef`). The approach is based on TypeChef [3], which transforms the stream of tokens to conditional tokens in a very early processing step, which eliminates our system constants. Therefore, these existing tools (TypeChef/Hercules) could not be used.

4 Approach

We have developed and implemented the following approach for performing such transformations. It is also depicted in Figure 2. We refer to SCs as the system constants that are to be transformed, and NSCs as the system constants that shall be kept as they are.

1. Normalize the unpreprocessed C code using Garrido's approach [1, 2]. This basically means moving `#if`s to the appropriate higher level and duplicating the code inbetween.

2. Parse the normalized code, using a C code parser that is extended by preprocessor constructs. Thanks to Garrido's normalization, preprocessor constructs only need to be provided at a few

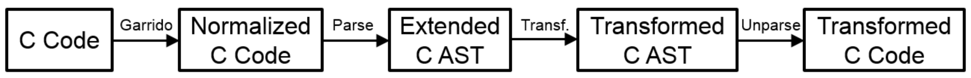

Figure 2: Processing chain for preprocessor variance transformation.

points in the C grammar. Without that, preprocessor directives can occur at arbitrary points in the code.

3. Build a corresponding abstract syntax tree (AST) that contains C code and preprocessor constructs.
4. Perform transformations on the AST:
   (a) Split `#elif`s to nested `#if`-`#else` constructs (only when both SCs and NSCs occur in the different conditions).
   (b) Transform `#if` conditions into a splittable form, which means that only system constants of either class SC or class NSC are contained in one `#if` condition.
   (c) Transform preprocessor constructs using specific rules. For example, transform a `#if` to an `if` if the condition only contains SCs.
5. Unparse the transformed AST, i. e., generate C code.

The transformation of step 4(b) is performed in the following way. First, the operands of the condition expression (with the same operator) are sorted by pure SC, pure NSC, and mixed expressions. The latter are also sorted in the same way. Then, the `#if` directive is transformed to a lambda expression using Church encoding:

$$\texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ f \equiv c\ t\ f \tag{1}$$

$$a \wedge b \equiv a\ b\ \texttt{false} \tag{2}$$

$$a \vee b \equiv a\ \texttt{true}\ b \tag{3}$$

After the whole conditional directive (including `#else`s and nested `#if`s) is encoded this way, the following two rules can be applied to split it in case of mixed (SC and NSC) expressions:

$$(a \wedge b)\ t\ f \equiv a\ (b\ t\ f)\ f \tag{4}$$

$$(a \vee b)\ t\ f \equiv a\ t\ (b\ t\ f) \tag{5}$$

Finally, the resulting lambda expression is transformed back to an `#if-#else-#endif` construct. This kind of transformation provides a splittable form with a minimal number of `#if` branches.

5 Evaluation

We evaluated the approach on 650 C files, which were taken from a typical embedded software from Bosch. The prototype tool was applied on them, and they were manually checked for correctness. The result was that we can sucessfully transform about 93% of the files fully automatically. The remaining files could not be processed due to the following main reasons:

- The code contains C constructs that are not yet supported by our prototype (e. g., structs and unions). This could be addressed by adding these features to the prototype.
- `#define`s or `#undef`s on the variables that are to be transformed. However, this is irrelevant for the meta program version use case.
- Usage of macros for code generation. If macros are not only used like expressions or functions, they may break the preprocessor-aware parser.

Given these restrictions, the success rate of 93% is quite good, and there is potential for further improvement.

6 Summary

We have introduced and evaluated a new approach for transforming preprocessor variance to post-build variance. Our prototypical implementation and evaluation shows that the majority of our code can be transformed. However, there will always remain some code with special kind of preprocessor usage that cannot be dealt with.

The transformation allows us to simplify software validation and to create meta program versions on demand. Validation is supported by checking all variants of the code with a single executable. Static code provers can also be applied on this transformed code. Project-specific meta program versions can be created with just those variance points transformed that are actually needed in dynamic form for a given project. The approach therefore helps to reduce implementation and test effort as well as resource usage.

References

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

[2] R. Heumüller, J. Quante, and A. Thums. Parsing variant C code: An evaluation on automotive software. *Softwaretechnik-Trends*, 34(2), 2014.

[3] 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.

[4] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, and S. Apel. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming*, 85(1):125–145, 2016.