Colligens: A Tool to Support the Development of Preprocessor-based Software Product Lines in C

Flávio Medeiros[1] , Thiago Lima[2] , Francisco Dalton[2] , Márcio Ribeiro[2] , Rohit Gheyi[1] , Baldoino Fonseca[2]

[1]Departamento de Sistemas e Computação
Universidade Federal de Campina Grande (UFCG) – Campina Grande – Brasil

[2]Instituto de Computação
Universidade Federal de Alagoas (UFAL) – Maceió – Brasil

flaviomedeiros@copin.ufcg.edu.br, rohit@dsc.ufcg.edu.br

{marcio, teol, fdbd, baldoino}@ic.ufal.br

Abstract. *A Software Product Line (SPL) is a set of products developed from reusable assets. These products share a common platform and we can customize them for specific customers. In an SPL, developers use the concept of features. They are the semantic units by which we can differentiate products. In C programming, we can implement features by mapping them to preprocessor directives, such as* `#ifdef` *and* `#endif`*. In this context, although there are tools to deal with C preprocessors and SPLs, they are not integrated, which hinders their usage in product line development. In this paper, we propose a tool that provides an integrated environment based on the Eclipse platform to support the development of preprocessor-based SPLs in C. Thus, developers can create feature models, see SPL metrics, and generate SPL products automatically after checking for the absence of invalid products.*

1. Introduction

A Software Product Line (SPL) consists of software systems that share common features that satisfy the needs of a specific market segment [Clements and Northrop 2001]. In this context, features are the semantic units by which we can differentiate programs in an SPL [Trujillo et al. 2006]. In C programming, developers may use the C preprocessor to implement features by mapping them to preprocessor directives, such as `#ifdef` and `#endif` [Thüm et al. 2012].

Although there are tools to deal with C preprocessors and SPLs, they are not integrated, which hinders their usage in product line development. For example, FeatureIDE [Thüm et al. 2012] enables developers to create and gather a feature model to their software projects in some languages like Java, but there is no mapping between this feature model and C preprocessors. As another example: despite recognizing preprocessors, C compilers such as GCC do not check all products to identify potential errors before generating them. Last but not least, TypeChef [Kästner et al. 2011], a variability-aware parser for C, outputs the results in console, making the tasks of locating and analyzing errors difficult. Therefore, to have the benefits of each tool, developers may need to write scripts to use the output of a tool as input of another and use different environments, technologies and languages, increasing their effort.

To minimize these problems and provide an environment containing the functionalities of these tools, we present in this paper *Colligens*,[1] an Eclipse *plug-in* tool to support C developers during preprocessor-based SPL implementations. When using our tool, developers can create feature models and map them to C preprocessor directives, check invalid products before generating them, reach erroneous code points by just clicking on the tool views, generate valid products according to both feature model and C language rules (with respect to syntax and type), analyze SPL metrics, and so forth.

We applied our tool to real software families that contain feature constraints defined in the configure files. In particular, we use the *libssh* project,[2] a multiplatform C library that implements the Secure Shell (SSH) protocol on client and server side, and *libpng*, the official PNG reference library[3]. Using the *Colligens* tool, we find syntax errors that the project developers confirmed as bugs.

2. Motivating Example

To implement SPLs and software families in C, developers may use preprocessors to implement features. For example, Figure 1 presents a real code snippet from the *libssh* software family that verifies signatures using the cryptographic libraries libgcrypt and libcrypto.

```
...  ...
510. static int sig_verify(SSH_SESSION *session, PUBKEY *pubkey, SIGNATURE *signature){
...    // Code here..
524.   switch (pubkey->type){
525.     case TYPE_DSS:
526.       #ifdef HAVE_LIBGCRYPT
...          // Code here..
532.         if (gcry_err_code (valid) != GPG_ERR_BAD_SIGNATURE){
533.           ssh_set_error(2, "DSA error : %s", gcry_strerror(valid));
534.       #elif defined (HAVE_LIBCRYPTO)
...          // Code here..
539.         if (valid == -1){
540.           ssh_set_error(session, 2, "DSA error : %s", ERR_get_error());
541.       #endif
542.           return -1;
543.         }
544.         ssh_set_error(session, 2, "Invalid DSA signature");
545.         return -1;
...    // Other case options
571.   }
572.   return -1;
573. }
...  ...
```

Figure 1. Code Fragment from the *libssh* project.

We may use TypeChef as illustrated in Figure 2 to detect errors. It identifies a syntax error, i.e., a missing bracket, in line 11500 when configuration (not `HAVE_LIBGCRYPT` and not `HAVE_LIBCRYPTO`) is set. However, in this scenario we face two problems. Firstly, the line number, pointed by TypeChef, that contains the error does not correspond to the original code (line 543 of dh.c file). Thus, developers need to spend additional time to manually identify the actual problematic line. Also, according to the configure file of the *libssh* project, the configuration where TypeChef identified the

[1] `https://sites.google.com/a/ic.ufal.br/colligens/`
[2] `http://www.libssh.org/`
[3] `http://www.libpng.org/`

error is not valid (not HAVE_LIBGCRYPT and not HAVE_LIBCRYPTO). In particular, one of the features must be always enabled. Again, developers would spend extra effort to investigate an error that actually does not exist. Notice that this kind of error does not arise in case we integrate a feature model to our project, since we can make TypeChef aware of such a feature model. Further, the *Colligens* tool can check problems like this one in several files.

```
MacBook-Pro: Typechef Flavio$ java -Xmx1024m -jar lib/TypeChef-0.3.5.jar -h platform.h --parse libssh/dh.c
parsing.
(def(HAVE_LIBGCRYPT) | def(HAVE_LIBCRYPTO))      succeeded
(!def(HAVE_LIBGCRYPT) & !def(HAVE_LIBCRYPTO))    failed: end of input expected at file libssh/dh.c:11500:12 (List())
```

Figure 2. Analyzing a *libssh* file with TypeChef using a command line tool.

Besides, we execute TypeChef using command line tools. To minimize these problems, we present a tool that not only integrates other tools, i.e., FeatureIDE and TypeChef, but also provides new functionalities such as mapping features and preprocessor directives, feature renaming, and SPL metrics.

3. A Tool to Support the Development of Software Product Lines in C

In this section, we present how our tool deals with the aforementioned problems. Then, we describe its architecture in Section 3.1, and the main functionalities in Section 3.2.

Colligens is an Eclipse *plug-in* distributed under the GNU General Public License (GPL) that provides support for the development of preprocessor-based SPLs using the C language. It integrates the TypeChef and FeatureIDE tools on the Eclipse platform. When using our tool, developers can identify invalid products (regarding syntax and type errors) without preprocessing the source code and generating binary code. In Figure 3, we present a view of the tool showing a type error that is present in all products with the feature HAVE_LIBGCRYPT. This view shows the file that contains the problem, so that developers can reach the original erroneous code by just clicking on the tool view.

In addition, *Colligens* also uses a feature model editor as we can see on the right hand side of Figure 3. Using the *libssh* feature model, our tool do not identify the syntax error in the invalid configuration that we mention in Section 2 (not HAVE_LIBGCRYPT and not HAVE_LIBCRYPTO). In this context, the tool checks the feature model constraints and TypeChef analyzes only valid configurations. Also, since the tool maps features to preprocessor directives, the renaming of a feature implies modifications on the source code, keeping the feature model and the source code consistent, which might reduce developers effort.

Even after checking the presence of errors and invalid products, we can force the *Colligens* tool to generate the SPL products. The generation of invalid products can ease the identification of problems, since the developers can focus on a single product, which contains no preprocessor directives on the source code. In Figure 4, we depict a view that lists the invalid products.

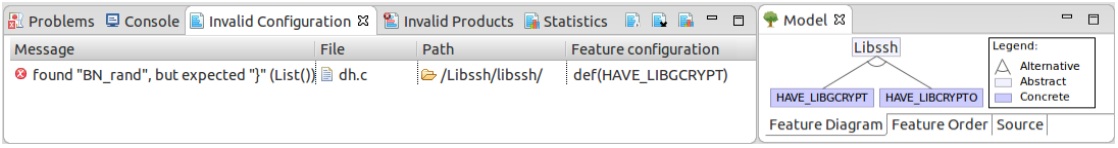

Figure 3. *Colligens* view to show syntax and type errors found by TypeChef, and the partial feature model of the *libssh* project.

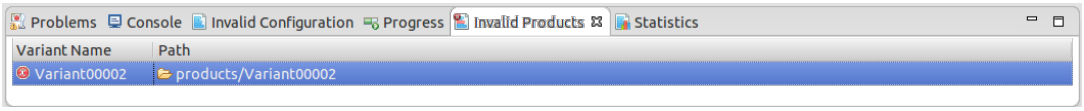

Figure 4. *Colligens* view to list invalid products.

Moreover, *Colligens* provides a view that shows metrics about the SPL (see Figure 5). Thus, developers can view data, such as the number of features, number of products, number of files, number of files with preprocessor directives, average number of directives per file, and the Lines of Code (LOC). The tool executes its functionalities on the Eclipse development environment without the need of additional scripts or command line tools. Hence, SPL developers may not need to perform tasks manually.

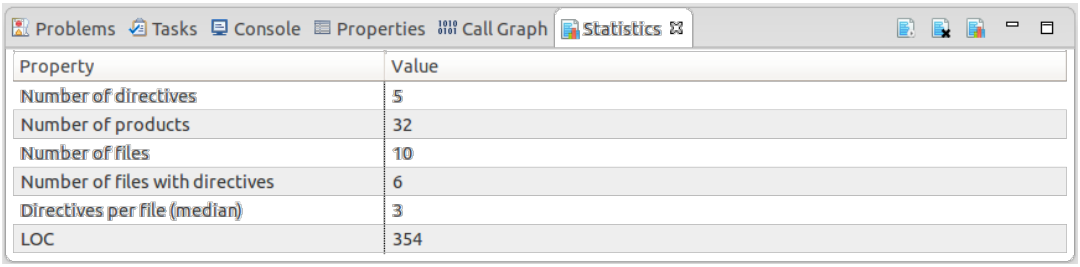

Figure 5. *Colligens* view to show SPL metrics information.

3.1. Architecture

In this section, we present the architecture of our tool. It is an Eclipse *plug-in* that integrates the TypeChef and FeatureIDE tools. As we can see in Figure 6, the tool architecture is composed by the Eclipse infrastructure, Eclipse C/C++ Development Tooling (CDT) *plug-in*, FeatureIDE, TypeChef, and the modules that we implemented using the Model-View-Controller (MVC) pattern as explained next:

- *Core*: it integrates TypeChef and FeatureIDE, maps features to preprocessor directives, and so forth;
- *View*: It provides views to show invalid products and configurations, and metrics, such as the number of features and products, median of preprocessor directives per file, and LOC;
- *Model*: it contains classes to represent, for example, syntax errors, invalid configurations and metric values;
- *Controller*: it is responsible to connect the model and view components, it notifies the model to update changes and the views to update the presentation of the model.

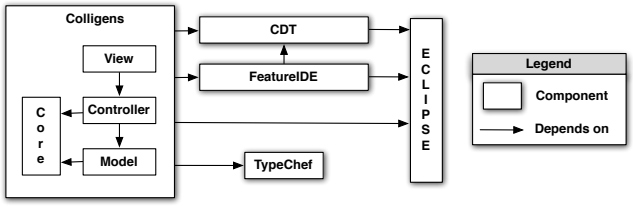

Figure 6. The Architecture of *Colligens*.

Colligens tool works in the following way. The user clicks on the tool views, and the controller calls TypeChef and FeatureIDE functionalities. Then, the controller calls the model methods to get the results and passes it back to the views. The controller also uses the CDT *plug-in* to get information, such as project name, paths, files selected on the project explorer, and source roots. As we can see in Figure 6, our tool depends on the Eclipse platform, CDT, FeatureIDE and TypeChef. FeatureIDE also depends on the CDT functionalities, such as source code editors, preference screens, and its building infrastructure.

3.2. Main Functionalities

In this section, we present the main functionalities of the *Colligens* tool to support the development of SPLs in C. Its main functionaries are:

- Since we implement a mapping between the FeatureIDE feature model and the C preprocessor directives, our tool provides a feature renaming functionality that keeps both feature model and code consistent;
- Generation of all SPL valid products in C, which is possible due to the aforementioned mapping;
- A list of the errors pointed by TypeChef in terms of an Eclipse view. Using this view, developers can click on its elements and our tool makes the cursor points to the actual erroneous line, improving code navigation;
- When trying to generate products, our tool warns whether there are erroneous ones (according to C language syntax and type rules). In case the user decides to continue, our tool provides a view that lists invalid products (see Figure 4);
- A metric view that provides SPL information;
- A preferences screen, allowing developers to set some FeatureIDE and TypeChef parameters.

4. Related Work

Beyond the TypeChef and FeatureIDE tools, we relate our work with the following tools. The FeatureC++ tool extends the syntax of C++ with new keywords to support Feature-Oriented Programming (FOP) [Apel et al. 2005]. It focuses on providing techniques to express features in a modular way, e.g., using aspects. Our tool has a different focus, it aims to support existing software families that implement features in a non-modular way using preprocessor directives.

FeatureHouse is a general approach to the composition of software artifacts [Apel et al. 2009]. It is language-independent in that we can compose artifacts written in various languages. It integrates the *FSTComposer*, *FSTMerge* and *FSTGenerator* tools.

The AHEAD tool suite provides several tools for FOP [Batory 2006]. The AHEAD tool also focuses on compositional SPL, and implements features as modular units, e.g., using C++ classes. Thus, these tools are appropriated to compositional SPL, different from our work that focuses on preprocessor-based product lines.

The CIDE tool implements the concept of virtual separation of concerns, where we can associate a feature with a specific background colour [Kästner and Apel 2009]. It focuses on preprocessor-based SPL. CIDE also integrates a variability-aware parser that checks for syntax and type errors. However, different from our work, it only accepts directives on structured code fragments.

5. Concluding Remarks

This paper presented *Colligens*, a tool to support the development of preprocessor-based SPLs in C. The tool consists mainly of an Eclipse *plug-in* that provides an integrated environment to create feature models, map features to preprocessor directives, and generate executable codes for the SPL products. Moreover, we applied our tool to real software families, the *libssh* and *libpng* projects. Our tool is based on FeatureIDE, a tool for feature-oriented programming, and TypeChef, which allows the checking of SPL properties without generating products. As a future work, we intend to implement extension points to integrate other tools, such as SuperC [Gazzillo and Grimm 2012], which is another variability-aware parser.

References

Apel, S., Kastner, C., and Lengauer, C. (2009). FEATUREHOUSE: Language-independent, automated software composition. In *Proceedings of the ICSE*. IEEE Computer Society.

Apel, S., Leich, T., Rosenmüller, M., and Saake, G. (2005). FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *In Proceedings of the GPCE*. Springer.

Batory, D. (2006). A tutorial on feature oriented programming and the ahead tool suite. In *Proceedings of the GTTSE*. Springer-Verlag.

Clements, P. C. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley.

Gazzillo, P. and Grimm, R. (2012). SuperC: parsing all of c by taming the preprocessor. In *Proceedings of the PLDI*. ACM.

Kästner, C. and Apel, S. (2009). Virtual separation of concerns – a second chance for preprocessors. *Journal of Object Technology (JOT)*, 8(6).

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the OOPSLA*. ACM.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2012). FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*.

Trujillo, S., Batory, D., and Diaz, O. (2006). Feature refactoring a multi-representation program into a product line. In *Proceedings of the GPCE*, pages 191–200. ACM.