Variability Code Analysis Using the VITAL Tool

Bo Zhang
Software Engineering Research Group
University of Kaiserslautern
Kaiserslautern, Germany
bo.zhang@cs.uni-kl.de

Martin Becker
Fraunhofer Institute Experimental
Software Engineering (IESE)
Kaiserslautern, Germany
martin.becker@iese.fraunhofer.de

ABSTRACT
As a product line evolves over time, variability realizations become overly complex and difficult to understand. This causes practical challenges in product line maintenance. To solve this issue, the VITAL tool is developed to automatically extract a variability reflexion model from variability code and conduct further analyses. In this paper, variability code analysis process using the VITAL tool is introduced, and each step of the analysis is demonstrated with an example product line.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering.* D.2.8 Metrics – *complexity measures.* D.2.9: Management – *software quality assurance.* D.2.13: Reusable Software – *domain engineering.*

General Terms
Measurement, Design, Experimentation.

Keywords
Variability Code Analysis, Product Line Evolution, Reverse Engineering Variability, Conditional Compilation.

1. INTRODUCTION
Nowadays product line development is often conducted in an incremental way, in which the variability artifacts evolve both in space and in time [3] [9]. On the one hand, existing variability realizations need to be adapted in order to satisfy the changing requirements of current product variants in the product line (i.e., evolution in time). On the other hand, new variability realizations need to be developed as new product variants are introduced into the product line (i.e., evolution in space). As a result, variability realizations tend to erode in the sense that they become overly complex and difficult to understand [13]. In particular, the code layout in variability code using conditional compilation is obfuscated by #ifdef directives, which are nested, tangled, and scattered in the code [8] [10]. The overly complex variability realizations cause a serious impact on productivity of SPL maintenance, and put the expected SPL advantages at risk.

In order to understand complex variability realizations and their evolution in large product lines, static code analysis and measurement with tool support is required. To this end, we have considered existing technology as well as commercial analysis tools for their suitability in this context. Factors that we have considered are the accuracy of the results, performance and interoperability of the tools, and the adoption and integration efforts. Moreover, as analysis and assessment demands differ considerably among different product line settings, the tool support should provide sufficient customization and extension mechanisms. However, to the best of our knowledge there is no full-fledged tool that supports variability code analysis in different settings.

Therefore, we have developed the VITAL (Variability ImprovemenT AnaLysis) tool set. VITAL is centered around a simple and extensible analysis data model that represents various variability elements and code assets, and it integrates different tool components for respective analysis and measurement purposes. The VITAL tool parses variability code that may potentially use different variability implementation mechanisms, and extracts a variability reflexion model (cf. Section 3.2) into a database. Then different variability code measurement can be conducted by running database queries. Besides, based on the database further code analysis and visualization are conducted cost-effectively by standard analysis tools such as MS Excel, R [16] and Treeviz [19]. VITAL has been used in our previous studies [22] and [7] for analyzing the variability code evolution of a large industrial product line (31 versions over four years).

The paper is organized as follows: Section 2 introduces the basic concepts of various variability elements especially in the context of conditionally compiled code. Then section 3 presents the variability code analysis using the VITAL tool. Finally the conclusion and future work is presented in section 4.

2. BASIC CONCEPTS
In variability realizations, we consider four types of variability elements, i.e., Variability (Var), Variation Point (VP), Code Variant (CV), and Variation Point Group (VPG). A Var represents a variable feature in the problem space and is realized in VPs (solution space) to select enclosed CVs. Moreover, VPs with equivalent predicate of selecting variant code fragments are grouped as a VPG, and thus they will be instantiated by the same Var with the same logic. As a Var may have cross-cutting concerns in variability realizations, VPs in the same VPG may be scattered in different code locations. The concept of VPG helps to understand the alignment of variability elements between problem space and solution space.

In practice, Conditional Compilation (CC) is a frequently used variability mechanism [5] [14]. In this context, macro constants

FOSD '14, September 14 2014, Västerås, Sweden

ACM 978-1-4503-2980-4/14/09…$15.00.
http://dx.doi.org/10.1145/2660190.2662113

are first defined in #define statements, and then they are used in #ifdef statements (in this paper #ifdef also subsumes other similar directives such as #if, #ifndef, and #elif) in code core assets. Based on the defined values of macro constants, a C-preprocessor adapts the core code by including or excluding enclosed code fragments into code compilation. Therefore, in CC each #ifdef block is considered as a VP with enclosed CV fragments, while each macro constant used in #ifdef statements is considered as a Var. Moreover, VPs with logically equivalent #ifdef statements are considered as a VPG.

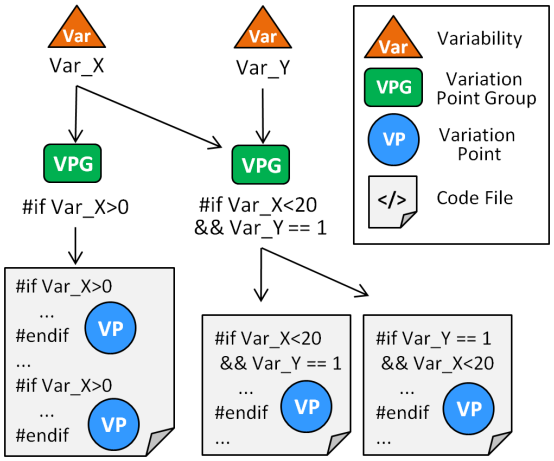

Figure 1. Variability Elements in Conditional Compilation.

Figure 1 shows an example of variability elements and their relationships implemented in CC. The example contains three files with variability code, where there are two Vars used in four VPs. These VPs are grouped into two VPGs, and VP in the same VPG has equivalent #ifdef logics. As shown in Figure 1, each Var can be used in multiple VPGs, while each VPG may contain multiple VPs. These VPs may be scattered in either one or multiple code files.

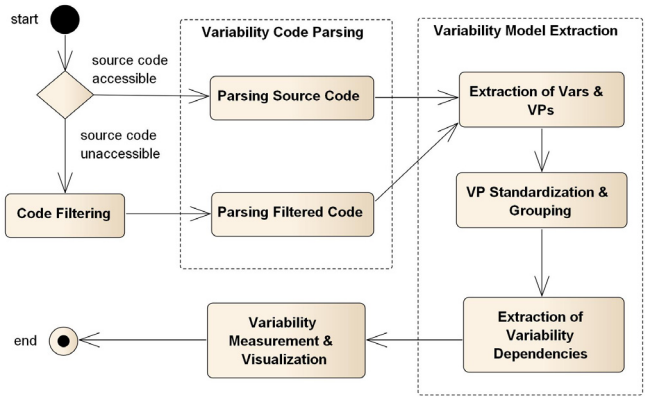

Figure 2. Variability Code Analysis Process.

3. VITAL VARIABILITY CODE ANALYSIS

In order to analyze variability code, the VITAL tool extracts a variability reflexion model from code and then conducts further measurement and visualization. An automated analysis process is presented as illustrated in Figure 2. First, either original or filtered variability code is parsed on the syntactic level. Second, the parsing result is used to extract a variability reflexion model from code. Third, variability code measurement and visualization are conducted based on the variability reflexion model. Each analysis step is introduced in following subsections.

Moreover, the variability code analysis is demonstrated using an example product line called FreeRTOS [4]. FreeRTOS is an open source real-time operating system that can be customized for different microcontrollers in the embedded system domain. In this paper, three versions of FreeRTOS source code (developed between 2004 and 2014) are analyzed by the VITAL tool.

3.1 Variability Code Parsing

As product line code can be implemented with various variability mechanisms (e.g., conditional compilation, module replacement, and aspect orientation) [5], different parsing techniques may be used to identify variability code elements. Since conditional compilation is one of the most frequently used mechanism, currently a C-preprocessor (CPP) code parser has been developed. The parser recognizes all CPP directives and their syntax in product line code. If the variability code is implemented by other mechanisms, then other parsing techniques and tools (e.g., the Fraunhofer SAVE tool [2] and SciTools Understand [17]) can be adopted and extended into the VITAL tool.

```
1  def CPP_Parser(code)
2  {
3      nameList, vpList, tempVPList;
4      for each line in code  {
5          if(line.match("#define"))  {
6              name = new Macro(line);
7              nameList.append(name);
8          }
9          if(line.match("#ifdef|#ifndef|#if|#elif")) {
10             ... // positive branch
11             vp = new VP(line);
12             vp.parent = tempVPList[-1]; // last element
13             tempVPList.append(vp);
14         }
15         else if(line.match("#else"))
16             ... // negative branch
17         else if(line.match("#endif"))
18         {
19             vp = tempVPList[-1];
20             do {
21                 vp.end = line.number;
22                 vpList.append(vp);
23                 tempVPList.delete(vp);
24                 vp = tempVPList[-1];
25             } while(vp.directive == "elif");
26         }
27     }
28     return nameList, vpList;
29 }
```

Figure 3. Algorithm for Parsing CPP Code.

The algorithm of the CPP parser is shown in Figure 3, which scans each line of code, parsing each macro constant as a potential Var and parsing each #ifdef block as a potential VP. In particular, the nesting structure of all #ifdef blocks is identified by parsing the #endif directives at the end of each #ifdef block. Note that an #ifdef block and its enclosed #elif directives share the same #endif directive at the end. While parsing each #ifdef statement, the CPP parser technically used the external tools of Pyparsing [15] and SrcML [18] to syntactically recognize code elements (e.g., #ifdef directives and macro constants). Currently it

is sufficient to use this CPP parser for parsing #ifdef statements. However, if it is necessary to parse the other normal source code in the future (e.g., for the purpose of extracting variability interdependencies), then more powerful variability code parsers (e.g., the parser used in TypeChef [20]) can be adopted.

A prerequisite of using the code parsing is that the source code is accessible. However, it is often difficult to get access to entire source code of a large product line in industrial case studies. To cope with this restriction, a source code filter is developed in order to extract only CPP directives from entire source code. With this, an industrial company can run the code filter and provide only CPP directives for this variability code analysis. These filtered "CPP skeletons" can be reconstructed into normal source code (filled with empty lines) for following code parsing and analysis.

```
1  def cppFilter(dir)
2  {
3     for file in traverseFile(dir)
4     {
5       ext = getFileExtention(file)
6       if ext in ['.h', '.c', '.hpp', '.cpp']
7       {
8          print '<file>%s</file>\n' % file
9          lineno = 0
10         for line in traverseLine(file)
11         {
12            lineno += 1
13            regex = '#\s*(if|else|elif|endif)'
14            if re.search(regex, line):
15               print '%d:%s\n' % (lineno, line)
16         }
17      }
18   }
19 }
```

Figure 4. Algorithm for Filtering CPP Code.

An abstract algorithm of code filtering is presented in Figure 4. For each C/C++ code file, the file name is saved in a "file" tag (line 8), and each CPP directive and its line number is also saved (line 15). Note that since both #ifdef and #ifndef directives starts from "#if", they are already included in the regular expression at line 13. In this paper, the analysis of the FreeRTOS system did not use this filtering algorithm since FreeRTOS is open source. However, the CPP filtering was applied successfully in our previous industrial case studies [22] and [7].

After parsing the source code of three versions of FreeRTOS, all CPP directives and their syntax are recognized. Then variability code (containing variability elements) is identified from this CPP code (will be explained in section 3.2). As shown in Table 1, during product line evolution the number of files with variability code has increased from 19 to 172, while the variability code size (i.e., size of corresponding #ifdef blocks) has increased from 627 LOC to 9461 LOC.

Table 1. Variability Code in FreeRTOS

Version	v2.4	v5.2	v8.0
# files with Variability Code	19	108	172
Variability Code Size (LOC)	627	3480	9461

3.2 Extraction of Variability Reflexion Model

After parsing (potential) variability code, further code analyses are conducted on semantic level in order to extract the variability reflexion model. The concept of reflexion model is initially introduced by Murphy [11] to reverse engineer source code and check its compliance with architecture. Similarly, the variability reflexion model contains various variability elements and their interdependencies.

A variability reflexion meta-model is shown in Figure 5, which contains the variability elements of Var, VP, and VPG as explained in Section 2. In the model, a VPG may contain a group of equivalent VPs (cf. Figure 1), and its predicate may involve multiple Vars. Each VP contains one or multiple CVs, so that they can be selected in product code depending on the predicate. Both Vars and VPs may have interdependencies respectively. The variability reflexion model was also introduced in our previous paper [22]. The following subsections will introduce how these variability elements and their dependencies can be extracted in the context of CPP code.

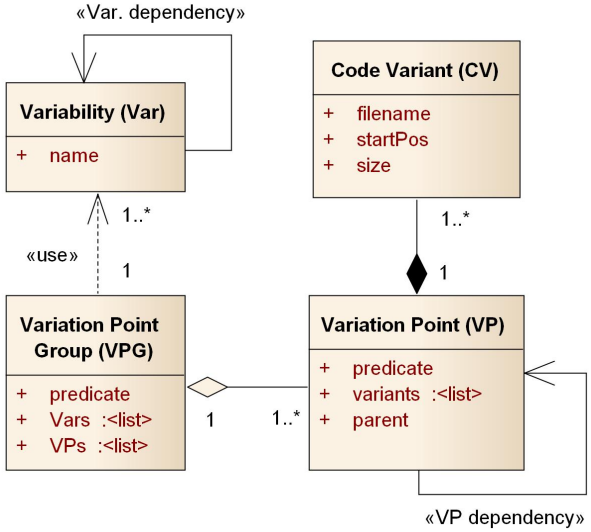

Figure 5. Variability Reflexion Meta-Model.

3.2.1 Extraction of Vars & VPs

In Conditional Compilation (CC), a Var is implemented as a macro constant, while a VP is implemented as an #ifdef block using Vars in its #ifdef statement. While CC is one of the most frequently used variability mechanism, CC is also used for other reasons in general (e.g. constant definition, include guards, and macro substitution). In order to analyze variability code, it needs to be distinguished, whether the CPP directives are used for variability realizations or something else. This complicates the analysis in general and typically requires extensive involvement of domain knowledge.

However, in practice the identification of Vars and VPs can be automated if there is a clear and strictly followed naming convention in variability code (e.g., the prefix "HAS_FEATURE" for all Vars in our previous studies [22] and [7]). Given a naming convention of Vars, the Vars can be automatically recognized by selecting extracted macro constants that follows the naming convention, and VPs can be automatically recognized by selecting #ifdef blocks that use Vars in their #ifdef statements. Note that

the parent link of a VP (specified in the variability reflexion model) needs to be updated if its parent #ifdef block is not a VP.

The source code of FreeRTOS does not have such naming convention for Vars. However, the include guards have a naming convention like "XXX_H", and thus they can be automatically identified and filtered out from variability code. Assuming the rest macro constants being used in #ifdef statements are Vars, the results of identified Vars and VPs are shown in Table 2. Although the results may still need further validation based on domain knowledge, they provide a first overview of variability on the code level.

Table 2. Variability Elements in FreeRTOS

Version	v2.4	v5.2	v8.0
# Vars	21	156	296
# VPs	52	503	1088
# VPGs	24	207	400

3.2.2 VP Standardization and Grouping

While a Var can be used in multiple VPs for selecting corresponding variant code fragments, VPs are implemented in different code locations and files. If these VPs contain logically equivalent predicates, they will be instantiated in the same way, and therefore they often implement the same functional or non-functional requirement. During variability code maintenance, a change in one VP will often affect other logically equivalent VPs. According to the variability reflexion model, VPs (i.e., #ifdef blocks) with logically equivalent #ifdef statements are grouped as a VPG. The VPG concept helps to identify VPs with equivalent logic and facilitates code change impact assessments in product line maintenance. However, comparing the logical equivalence of VPs with arbitrary #ifdef statements is theoretically undecidable.

To solve this issue, the #ifdef statements of all VPs are standardized syntactically into a canonical form. Specifically, the standardization process includes: 1) rewriting #ifdef statements as "#if defined(...)" and #ifndef statements as "#if !defined(...)"; 2) removing extra spaces and parentheses in #ifdef expressions; 3) rewriting negations like "!(X==Y)" to "X!=Y". Besides, the standardized #ifdef statements are further analyzed logically by a SMT solver Z3 [21] to identify inconsistent term orders in equivalent #ifdef statements, e.g., "#if X>Y" and "#if Y<X" (currently Z3 can be used to only recognize conjunctions and disjunctions where the associativity is not affected by parentheses). This provides an "approximate solution", which has been able to automatically identify VPs with equivalent #ifdef statements as a VPG in all of our variability code analyses conducted so far.

After analyzing all VPs in the three versions of FreeRTOS, the VPGs in each version are derived as shown in Table 2. It shows that all variability elements have been significantly increased during product line evolution.

3.2.3 Extraction of Variability Dependencies

Besides extracting variability elements, the dependencies between Vars and VPs also can be extracted based of code parsing results. In CC these dependencies can be extracted based on #ifdef nesting. For instance, given a VP "#if Var_Y>0" nested in another VP "#ifdef Var_X", the nested VP depends on the parent VP because the nested VP is included in core code only if the parent VP condition "#ifdef Var_X" is true. Besides, the Var_Y depends on Var_X because Var_Y needs to be defined if Var_X is also defined. In VITAL both of these dependencies are automatically extracted using Python scripts. Besides variability dependencies based on #ifdef nesting, there are other kinds of dependencies even in C preprocessor code (e.g., based on definition and usage of a variable or method in different VPs [12]). However, extraction of these dependencies is probably not always performed fully automatically but requires domain knowledge sometimes.

3.3 Measurement and Visualization

As a result of variability reflexion model extraction, informations of variability elements are identified and stored in a database (current MS Access is used). Then they are analyzed to measure the complexity of variability code and its evolution. Table 3 shows a list of six variability code metrics, which are supported in the VITAL tool and indicate variability code complexity in different ways. In our previous study [22], these metrics have been defined to measure different variability elements, and the code measurement is automatically conducted by the VITAL tool using database queries. Further variability code measurement can be easily extended by creating additional queries or using other external tools.

Table 3. Variability Code Metrics Supported in VITAL

Metric	Description
VP Nesting Degree	#ifdef nesting level of a given VP
Var Tangling Degree	# Vars used in a given VP
Var Fan-out on VPG	# VPGs that contain a given Var
Var Fan-out on File	# files that contain a given Var
Var Fan-in on File	# Vars included in a given file
VP Fan-in on File	# VP included in a given file

Table 4 shows the measurement results of these six metrics on the three versions of FreeRTOS. It shows that the variability code of FreeRTOS becomes complex in different perspectives including variability nesting, scattering (on VPG and on file), and file complexity (in terms of Vars and VPs). One exception is variability tangling, where the Avg. tangling degree has decreased along the three versions.

Table 4. Variability Code Measurement in FreeRTOS

Version	v2.4	v5.2	v8.0
Avg. VP Nesting Degree	1.40	1.59	1.61
Avg. Var Tangling Degree	1.08	1.07	1.06
Avg. Var Fan-out on VPG	1.33	1.47	1.51
Avg. Var Fan-out on File	1.86	2.24	2.48
Avg. Var Fan-in on File	2.05	3.23	4.27
Avg. VP Fan-in on File	2.74	4.66	6.33

Moreover, the VITAL analysis also supports visualization of variability measurement results. For instance, the measurement results of Var Fan-in on File (defined as number of used Vars in each code file) can be visualized as a rectangular treemap using Treeviz [19]. Figure 6 shows such visualization of FreeRTOS v8.0. Each block in the treemap represents a variability code file in a hierarchical structure, and the size of the block indicates size

of the file. The file blocks are colored in gradient from blue, yellow, to red, which indicate their metric value of Var fan-in on File. The orange and red file blocks represent "hotspot" files with a large number of Vars (Max. 138), while the blue file blocks represent files with a small number of Vars (Min. 1). If necessary, other visualization tools (e.g., Cytoscape [1] and Gephi [6]) can also be integrated into the VITAL tool.

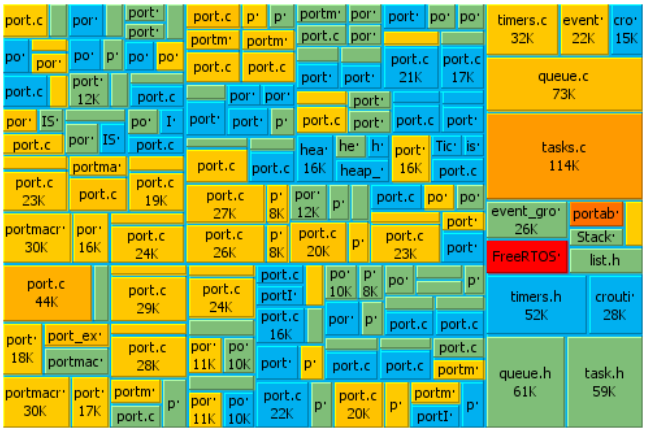

Figure 6. Visualization of Var Fan-in on File.

4. CONCLUSION AND FUTURE WORK

In this paper, the variability code analysis using our VITAL tool is introduced particularly in the context of variability code using conditional compilation. The VITAL tool can automatically extract a variability reflexion model from variability code, and then conduct various code measurement and visualization. Each step of VITAL analysis has been demonstrated with an example product line.

Currently the VITAL tool can parse variability code using conditional compilation. In the future other parsing techniques can be extended into the tool in order to analyze variability code using other mechanisms such as aspect orientation or module replacement.

5. ACKNOWLEDGMENTS

This work was sponsored by the Innovation Center Applied System Modeling, which is funded by Fraunhofer and the state Rhineland Palatinate of the Federal Republic of Germany.

6. REFERENCES

[1] Cytoscape tool. http://www.cytoscape.org/ (Last visit: August 2014)

[2] S. Duszynski, J. Knodel, and M. Lindvall, "SAVE: Software architecture visualization and evaluation," in Software Maintenance and Reengineering, 2009. CSMR 2009. 13th European Conference on, Mar. 2009, pp. 323-324.

[3] C. Elsner, G. Botterweck, D. Lohmann, and W. Schröder-Preikschat, "Variability in time - product line variability and evolution revisited," in Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems, ser. VaMoS '10, 2010.

[4] FreeRTOS. http://www.freertos.org/. (Last visit: August 2014).

[5] C. Gacek and M. Anastasopoules, "Implementing product line variabilities," SIGSOFT Softw. Eng. Notes, vol. 26, no. 3, pp. 109-117, May 2001.

[6] Gephi tool. http://gephi.org/ (Last visit: August 2014).

[7] Robert Hellebrand, Adeline Silva, Martin Becker, Bo Zhang, Krzysztof Sierszecki, Juha Erik Savolainen, "Coevolution of Variability Models and Code: An Industrial Case Study", In Proceedings of the 18th International Software Product Line Conference, ACM, New York, NY, USA, 2014.

[8] C. Kästner, "Virtual separation of concerns: Toward preprocessors 2.0," Ph.D. dissertation, 2010.

[9] C.W. Krueger, "New Methods behind a New Generation of Software Product Line Successes", In K.C. Kang, V. Sugumaran, and S. Park, "Applied Software Product Line Engineering", Auerbach Publications, 2010, pp. 39-60.

[10] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 105-114.

[11] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," IEEE Trans. Softw. Eng., vol. 27, no. 4, pp. 364-380, Apr. 2001.

[12] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in 36th International Conference on Software Engineering (ICSE'14), 2014.

[13] T. Patzke, M. Becker, M. Steffens, K. Sierszecki, J. E. Savolainen, and T. Fogdal, "Identifying improvement potential in evolving product line infrastructures: 3 case studies," in Proceedings of the 16th International Software Product Line Conference - Volume 1, ser. SPLC '12. New York, NY, USA: ACM, 2012, pp. 239-248.

[14] T. Patzke, "Sustainable evolution of product line infrastructure code," Ph.D. dissertation, 2011.

[15] Pyparsing. http://pyparsing.wikispaces.com/ (Last visit: August 2014)

[16] R. http://www.r-project.org/. (Last visit: August 2014)

[17] SciTools Understand V2.6. http://www.scitools.com/. (Last visit: August 2014)

[18] SrcML.http://www.sdml.info/projects/srcml/. (Last visit: August 2014)

[19] Treeviz. http://www.randelshofer.ch/treeviz/. (Last visit: August 2014)

[20] TypeChef. http://ckaestne.github.io/TypeChef/. (Last visit: August 2014)

[21] Z3 Tool. http://z3.codeplex.com/ (Last visit: August 2014).

[22] B. Zhang, M. Becker, T. Patzke, K. Sierszecki, and J. E. Savolainen, "Variability evolution and erosion in

industrial product lines: a case study," in Proceedings of the 17th International Software Product Line Conference, ser. SPLC '13. New York, NY, USA: ACM, 2013, pp. 168-177.