BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models

Sergio Segura, José A. Galindo, David Benavides, José A. Parejo and Antonio
Ruiz-Cortés
Department of Computer Languages and Systems
University of Seville, Spain
{sergiosegura, jagalindo, benavides, japarejo, aruiz}@us.es

ABSTRACT

The automated analysis of feature models is a flourishing research topic that has called the attention of both researchers and practitioners during the last two decades. During this time, the number of tools and techniques enabling the analysis of feature models has increased and also their complexity. In this scenario, the lack of specific testing mechanisms to assess the correctness and good performance of analysis tools is becoming a major obstacle hindering the development of tools and affecting their quality and reliability. In this paper, we present BeTTy, a framework for *BE*nchmarking and *T*es*T*ing on the anal*Y*sis of feature models. Among other features, BeTTy enables the automated detection of faults in feature model analysis tools. Also, it supports the generation of motivating test data to evaluate the performance of analysis tools in both average and pessimistic cases. Part of the functionality of the framework is provided through a web-based interface facilitating the random generation of both classic and attributed feature models.

Keywords

Testing, feature models, benchmarking, automated analysis, validation

1. INTRODUCTION

Software product line engineering is an approach to develop families of related software products by reusing a common set of features instead of building each product from scratch. Feature models play a key role in this paradigm by providing a high-level representation of all the products of the product line. The automated extraction of information from feature models is a thriving topic that has called the attention of researchers for the last twenty years [1]. During this time, numerous operations, techniques and tools to get information from feature models have proliferated and a whole community has been built around *the automated analysis of feature models* [1]. Currently, the rapid progress of this discipline is naturally leading to an increasing concern

about the quality of feature model analysis tools. The goal now is not simply developing basic research prototypes but solid and high quality analysis tools in terms of absence of bugs and good performance. In this context, current testing methods are mainly random and guided by intuition rather than by well-studied testing techniques. This makes testing conclusions rarely rigorous and verifiable, weakening the value and scope of research contributions and reducing the user's confidence in the correctness of analysis tools.

To address the problem of testing on the analysis of feature models, we have presented a number of techniques, algorithms and tools. These contributions are the results of the application of several classical and innovative testing techniques to the context of the analysis of feature models. Regarding functional testing, we have presented a test suite [10] and automated test data generator [11, 12] enabling the efficient detection of faults in analysis tools. Regarding performance testing, we have presented an evolutionary algorithm for the generation of computationally–hard feature models to reveal the deficiencies of tools in pessimistic cases [9, 13]. These contributions have been evaluated using extensive and rigorous experiments that reveal the efficacy and efficiency of our approaches. Among other results, we detected two faults in FaMa [3] and another two in SPLOT [17], two popular analysis tools widely used in the community of automated analysis of feature models. In order to make our contributions accessible to the feature modeling community and encourage other researchers and practitioners to use, evaluate and extend our work, we have integrated all these techniques in a framework called BeTTy. BeTTy is an extensible and highly configurable tool supporting BEnchmarking and TesTing on the AnalYsis of feature models. BeTTy features can be summarized as follows:

- Random generation of feature models. BeTTy enables the random generation of highly-customized feature models. These may be used to evaluate the performance of analysis tools in average cases.

- Random generation of attributed feature models. BeTTy also supports the random generation of attributed feature models (a.k.a. extended feature models) [1]. These models are commonly used to add non-functional attributes to the features. To the best of our knowledge, we are the first authors providing a random generator for this type of models.

- Automated generation of test data for functional testing. BeTTy supports the generation of inputs and expected outputs for a number of analysis operations on

feature models making the detection of faults straight-forward.

- Evolutionary feature model generator. BeTTy includes an evolutionary algorithm supporting the generation of feature models maximizing user-defined optimization criteria. This feature can be used to generate computationally–hard feature models that reveal the performance of tools in pessimistic cases.

- Benchmarking. The BeTTy framework integrates several components to facilitate the performance comparison of feature model analysis tools.

BeTTy is written in Java, released under LGPL3 license and distributed as a jar file facilitating its integration in external projects. Also, part of the functionality of the framework is offered through a web-based application enabling the generation of random feature models within a few clicks.

The rest of the paper is structured as follows: Section 2 presents the challenges addressed by BeTTy. Sections 3 and 4 ilustrate how BeTTy can be used for functional and performance testing of feature model analysis tools. The architecture and the web-based interface of the framework are presented in Sections 5 and 6, respectively. The related works found in the literature are presented in Section 7. Finally, we summarize our conclusions and plans for future work in Section 8.

2. CHALLENGES

As an illustrative example, suppose that we developed a tool for the automated analysis of feature models called VaMoSAnalizer. The tool implements some of the analysis operations on feature models reported in the literature such as the one that determines whether an input feature models is consistent or whether it contains dead features [1]. In the next sections, we introduce some of the challenges that should be addressed to assess the correctness and performance of the tool.

2.1 Challenge 1: Functional Testing

Functional testing is intended to check whether a program satisfies its functional requirements, i.e. whether the program does what the user expects it to do. In the context of our example, this means to assess whether the analysis operations implemented in VaMoSAnalizer are working as expected or whether they include faults. Figure 1 illustrates this challenge.

The key point when performing functional testing is to design test data (i.e. inputs and expected outputs) that help us to gain confidence in the correctness of the tool. Test data must fulfil certain requirements such as being complex enough, covering the input domain as much as possible and keeping the number of test cases manageable.

In order to gain confidence in the correctness of VaMoSAnalizer, we should therefore design effective test data to assess its functionality. More specifically, we must design specific test cases for each analysis operation under test. Consider, as an example, the operation that returns the dead features of a feature model (i.e. features that can never be selected due to a modelling error). To test this operation, we should run the tool with a number of input feature models and check whether the returned features correspond to the actual set of dead features of each model.
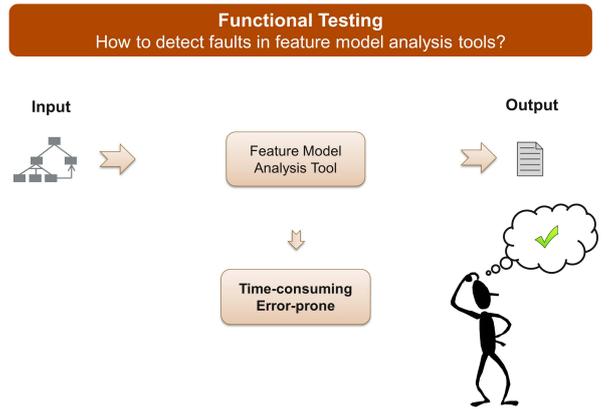

Figure 1: The challenge of functional testing

A first attempt to solve this challenge would be to manually build a set of test cases (i.e. test suite) as we did in the past [10]. We learnt, however, that the feasibility of this approach is only partial since it suffers from several practical limitations. First, the manual design of test cases relies on the ability of the tester to decide whether the output of an analysis is correct. This is time-consuming, error-prone and in most cases infeasible due to the combinatorial complexity of the analyses. As a result, manual test cases tend to use small and in most cases over-simplistic input models whose output can be calculated by hand. This limitation, also found in many other software testing domains, is known as the *oracle problem* [20], i.e. impossibility to determine the correctness of a test output. Another limitation of manual test data is that it is not a generic solution and therefore new test cases must be designed for each analysis operation which is tedious and time-consuming. The development of tool support for the automated detection of faults in feature model analysis tools overcoming these limitations remains as an open challenge.

2.2 Challenge 2: Performance Testing

Feature model analysis tool is also a complex software system in terms of computational complexity. It is well known in the literature that there are several analysis operations that are computationally–hard to perform in some cases. For instance, finding the set of products represented by a feature model is an operation with exponential complexity [1]. Benchmarking and performance testing has been recognized as a challenge in the domain of feature model analysis [1, 14, 16].

In order to evaluate the performance of VaMoSAnalizer, we should again design specific test data, i.e., a set of input feature models that show the behavior of tools in different situations. For an exhaustive evaluation of the performance, however, test data should include not only average feature models but also computationally–hard feature models that exploit the vulnerabilities of analysis tools in extreme situations. This would allow users and developers to know the behavior of tools in pessimistic cases providing a better idea of their real power. Figure 2 illustrates this challenge.

There are different complementary approaches for performance analysis of feature models. A first attempt would be to use real feature models but it is well known that compa-
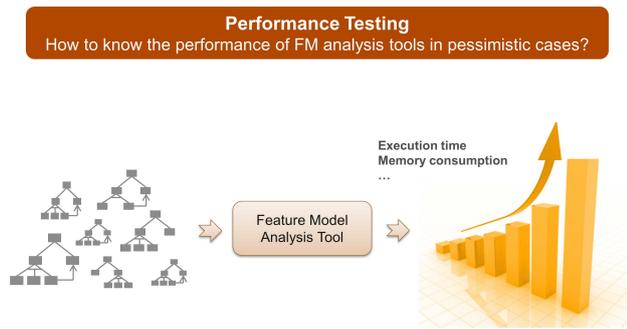
Figure 2: The challenge of performance testing

nies are reluctant to show their feature models since those very often include strategic information. There are some authors that have inferred variability models from open source software [4, 15]. Another approach is to generate feature models using random mechanisms. There are several works in the literature that follows this direction [2, 6, 8, 21]. An open challenge in the literature and existing tools is to randomly generate feature models including attributes.

Real and randomly generated feature models are a good option for gathering average results in terms of performance. However, there is a lack of testing mechanisms to assess the performance of analysis tools in extreme situations, i.e. check the performance with inputs that maximize the execution time or the memory consumed by the tool during the analysis. For instance, consider we intend to evaluate the performance of VaMoSAnalizer in hard cases. We could just use a random feature model of huge size, as it is usually done in the literature [1, 9, 13]. However, a negative consequence of using huge feature models to evaluate the performance of tools is that they frequently fall out of the scope of their users. Hence, both developers and users would probably be more interested to know whether their tool may crash with a computationally–hard model of small or medium size rather than knowing the execution times of huge random models out of their scope. Developing a configurable tool supporting the generation of computationally–hard feature models of pre-defined size is also an open challenge.

3. BeTTy FOR FUNCTIONAL TESTING

BeTTy enables the automated generation of test cases for the automated analysis of feature models using the approach presented in [11]. A test case is composed of a set of inputs (a feature model and some other optional parameters) and an expected output of the analysis operation under test. The key idea behind BeTTy is that most analysis operations on feature models can be answered by simply inspecting their set of products adequately. Figure 3 depicts an example of how the framework works. The process starts with a trivial input feature model and its set of products. The model is then extended progressively by adding to it random relationships and constraints. The set of products is also updated at each step by using so-called metamorphic relations, that is, relations between the operators of the model and the set of products. Once a feature model with the desired properties is created, it is used as nontrivial input for the tests. Also, its set of products is automatically inspected to get

the expected output of the analysis operations under tests. As an example, assume that we run VaMoSAnalizer using the model generated in Figure 3 as input. We could test the functionality of a number of operations by simply answering the following questions:

- *Is the model consistent?* Suppose that VaMoSAnalizer returns that the model is consistent, i.e. it represents at least one product. Looking at the set of products associated to the model generated by BeTTy we can easily conclude that the output is correct since the set of products is not empty. VaMoSAnalizer would pass the test.

- *Is P={A,C,F} a valid product?* Assume that our tool VaMoSAnalizer returns that the product is not valid, i.e. it is not included in the set of products represented by the model. Examining the set of products returned by BeTTy, however, we check that the product is included in the set and therefore it is valid. VaMoSAnalizer would therefore not pass the test.

- *How many different products represent the model?* Let us suppose that VaMoSAnalizer returns that the model represents 6 different products. We can easily check that the output is correct by simply counting the set of products generated by BeTTy. VaMoSAnalizer would pass the test.

- *What is the commonality of feature B?* The commonality of a feature is the percentage of products in which that features appears [1]. Suppose that VaMoSAnalizer returns 66%. According to BeTTy, feature B is included in 4 out of the 6 products of the set which means that the output is correct. VaMoSAnalizer would pass the test.

- *Does the model contain any dead feature?* Suppose that VaMoSAnalizer returns that feature F is dead. The set of products returned by BeTTy, however, include all the features of the model which means that there are no dead feature in the input model. Again, VaMoSAnalizer would not pass the test.

Using this technique, BeTTy allows users to test their analyses applications automatically using complex feature models representing thousands or even millions of products. BeTTy has proved to be effective detecting faults in real tools for the automated analysis of feature models such as the FaMa Framework [3] and SPLOT [11]. Also, it has shown to be much more effective than manually-designed test cases for the analysis of feature models [11, 12]. At the time of writing this paper, we have used BeTTy to automatically test up to 18 different analysis operations on feature models. See Appendix A (Figure 8) for a code example showing how to generate a feature model and its set of products using BeTTy.

4. BeTTy FOR PERFORMANCE TESTING

BeTTy supports the generation of random and computationally hard input feature models to be used in the performance evaluations of feature model analysis tools. These features are detailed in the next sections.
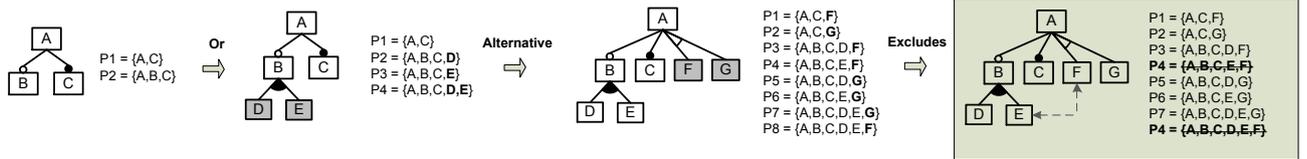
Figure 3: BeTTy approach for the generation of a random feature model and its set of products

4.1 Generation of random (attributed) feature models

BeTTy enables the generation of random feature models. The generation is performed in two steps. First, a feature tree is generated following the algorithm described by Thum et al. in [18]. Then, cross-tree constraints (depends and excludes) are added to the model fulfilling the following requirements: *i*) features with parent relations cannot be part of a constraint, and *ii*) two features cannot share more than one constraint. The generation can be highly configured through a number of input parameters such as number of features, percentage of cross–tree constraints, percentage of each type of relationship or maximum branching factor of the models to be generated. Users can optionally use a *seed number* to make the generation reproducible in later experiments. A coding example is provided in Appendix A (Figure 7).

In addition to classic feature models, BeTTy supports the random generation of extended feature models, i.e. feature models with attributes. These types of feature models are constructed in two steps. First, a classic feature model is generated as explained previously. Then, attributes are added to the leave features. The values for the attributes are added following pre–defined distributions using the library Apache Math [7]. The number of attributes per features and the type of distribution are configurable options. The current version of the tool only supports integer domains.

Figure 4 depicts an attributed feature model generated using BeTTy. As proposed by Benavides et al. [1], an attribute is defined by a name, a domain and a value. The piece of code used to generate this model is presented in Appendix A (Figure 9).
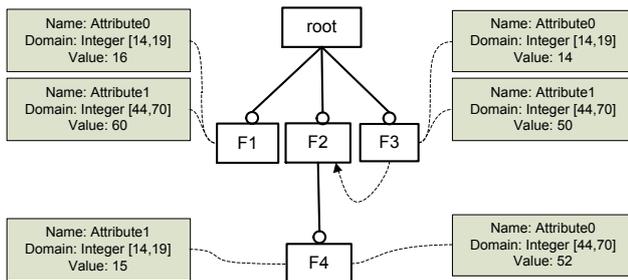

Figure 4: Sample attributed FM generated by BeTTy

4.2 Generation of computationally–hard feature models

BeTTy integrates the novel evolutionary algorithm for op-

timized feature models described in [9, 13]. This allows users to generate a feature model maximizing or minimizing certain properties of the model or their analyses. For instance, we could search for a feature model with a given number of features minimizing the height of the tree or maximizing the execution time required for its analysis. This is done by simply defining an objective function and using it as an input of the framework. This function determines the quality of a feature model with respect to a given optimization criteria.

The most appealing application of our evolutionary approach is the possibility of generating computationally–hard feature models. Given a tool and an analysis operation, BeTTy can generate input models of a predefined size maximizing aspects as the execution time or the memory consumption of the tool when performing the operation over the model. This allows users and developers to know the behaviour of tools in pessimistic cases providing an idea of their power in extreme situations. Experiments using our evolutionary algorithm on a number of analysis operations and tools have successfully identified input models causing much longer executions times and higher memory consumption than random models of identical or even larger size.

As an example, we compared the effectiveness of random search and BeTTy in generating feature models with up to 1,000 features maximizing the time required by a CSP solver to check their consistency. The results revealed that the hardest random model found required 0.2 seconds to be analyzed meanwhile BeTTy was able to find several models taking between 23.1 and 27.5 minutes to be processed. Not only that, we found the hardest feature models generated by BeTTy in the ranges 500-1,000 features were remarkably harder to process than random models with 10,000 features. For more details about the evaluation of our approach we refer the reader to [13]

Appendix A (Figure 10) shows a code example illustrating how to use BeTTy to generate a feature model maximizing the execution time required for its analysis.

5. ARCHITECTURE OF BETTY

Figure 5 presents a high level representation of the architecture of BeTTy. As illustrated, the framework is composed of two main blocks, the core and the extensions. The BeTTy core contains the set of interfaces and classes used to extend the framework and to build applications on top of it. This mainly consists of the following components:

- *FaMa feature model metamodel.* This is the set of classes used to represent and manipulate feature models. BeTTy integrates both, classic and attributed feature model metamodels located at the core of the FaMa Tool Suite [3]. This is a powerful implementation of
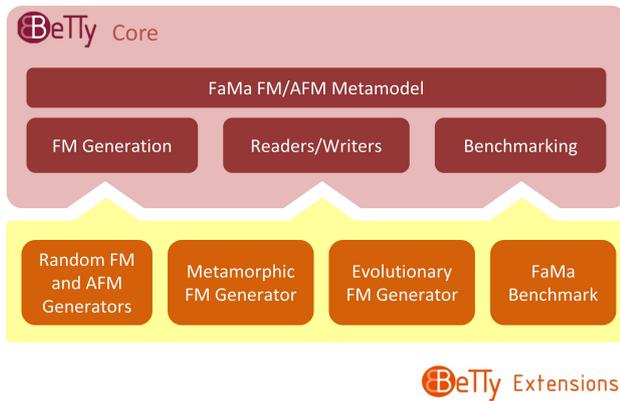
Figure 5: BeTTy framework architecture

a feature model metamodel highly tested and used in the feature modelling community.

- *FM generation.* This component contains support interfaces, methods and classes to facilitate the development of feature model generators.

- *Reader and writers.* This component consists of a set of ready–to–use feature model readers and writers for different formats. The formats currently supported by BeTTy are i) Simple XML Feature Model format (SXFM) [17], ii) Fama XML Format [3], iii) FaMa Textual Format [3], iv) X3D format [19] for 3D visualisation of feature models, and v) Dot format for the visual representation of the models in the graph visualization tool Graphviz [5].

- *Benchmarking.* This component contains the interfaces and basic classes to facilitate the generation, execution and saving of results during the performance evaluation of analysis tools.

The BeTTy extensions comprise the set of testing and performance tools developed on top of BeTTy. These mainly consist of the random feature model generators presented in Sections 3 and 4. Additionally, it integrates the FaMa Bechmarking component whose classes abstract FaMa users from low-level details making performance evaluations with the tool straightforward.

The BeTTy framework is freely distributed under LGPL v3 license and can be downloaded from the BeTTy Web site `www.isa.us.es/betty`.

6. BETTY ONLINE

Part of the functionality of BeTTy is provided through a web interface to make our work easily accessible to the community. In particular, the web application facilitates the generation of random feature models, both basic and extended. Figure 6 shows a screenshot of the application. Users must introduce several mandatory fields such as the number of models to be generated, the number of features and the percentage of cross-tree constraints of the models. Then, they can optionally specify a number of parameters such as the percentage of cross-tree constraints,

the percentage of relationships of each type (mandatory, optional, alternative and or), the maximum branching factor or the maximum number of child features in set relationships. Optionally, a user may ask for consistent (a.k.a. satisfiable) models only. The current version of the tool allows users to download the generated models in five different formats (see Section 5). The application is accessible at `http://www.isa.us.es/betty/betty-online`

7. RELATED TOOLS

We found a good number of approaches using random feature models to test the performance of their tools [1]. However, in most of the cases, these were generated using ad–hoc tools not publicly available. We found that only the SPLOT website [17] provides a standalone Java application for the generation of random feature models and their storage in Simple XML Feature Model format. The tool, not extensible, receives several parameters constraining the size and properties of the feature model to be generated (i.e. number of features or percentage of mandatory features).

When compared to related works, BeTTy is the first extensible framework specifically designed for functional and performance testing of feature model analysis tools. In addition to the random generation of feature models, it also provides extra and novel features including the generation of products using metamorphic relations, guided generation of feature models using optimization criteria and support classes for benchmarking. Also, BeTTy support most popular feature model formats and is distributed as a jar file facilitating its interoperability with other tools. To the best of our knowledge, BeTTy is the first tool enabling the online generation of random feature models facilitating its usage and accessibility.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented BeTTy, a benchmarking and testing framework for the automated analysis of feature models. Regarding functional testing, BeTTy supports the generation of random feature models and their set of products. This makes it possible to generate inputs and expected outputs for a number of analysis operations on feature models speeding the detection of faults. Regarding performance testing, BeTTy supports the random generation of both basic and attributed feature models to explore the behaviour of the tools under tests in average cases. More importantly, BeTTy enables the generation of computationally–hard feature models to reveal the weaknesses of analysis tools in extreme situations. As a part of our work, we also present an innovative web application facilitating the generation of highly customized random feature models. In short, we have motivated the need for testing on the analysis of feature models and we have presented a novel framework addressing the challenge.

A number of tasks remain for our future work. We plan to add new features to our random generator like the generation of complex cross-tree constraints (those involving more than two features). Also, we intend to extend our attributed feature model generator to support real domains and constraints among attributes. Finally, we would be glad to integrate BeTTy extensions from the community such as support for new formats or more complex generators.

Figure 6: BeTTy On-line feature model generator - http://www.isa.us.es/betty/betty-online

9. REFERENCES

[1] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010.

[2] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[3] FaMa Tool Suite. `http://www.isa.us.es/fama/`, accessed November 2011.

[4] J.A. Galindo, D. Benavides, and S. Segura. Debian packages repositories as software product line models. towards automated analysis. In *Proceeding of the First International Workshop on Automated Configuration and Tailoring of Applications (ACOTA)*, 2010.

[5] Graphviz. . `http://www.graphviz.org/`, accessed November 2011.

[6] G. Kapfhammer. *The Computer Science Handbook*, chapter Software Testing. CRC Press, 2nd edition, June, 2004.

[7] Apache Math. Apache math. `http://commons.apache.org/math/`, accessed November 2011.

[8] M. Mendonca, D.D. Cowan, W. Malyk, and T. Oliveira. Collaborative product configuration: Formalization and efficient algorithms for dependency analysis. *Journal of Software*, 3(2):69–82, 2008.

[9] S. Segura. *Functional and Performance Testing of Feature Model Analysis Tools. Extending the FaMa Ecosystem*. PhD thesis, Dept. of Computer Languages and Systems, University of Seville, 2011.

[10] S. Segura, D. Benavides, and A. Ruiz-Cortés. Functional testing of feature model analysis tools: a test suite. *Software, IET*, 5(1):70 –82, february 2011.

[11] S. Segura, Robert M. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology*, 53(3):245–258, 2011.

[12] S. Segura, Robert M. Hierons, D. Benavides, and A. Ruiz-Cortés. Mutation testing on an object-oriented framework: An experience report. *Information and Software Technology Special Issue on Mutation Testing*, 2011.

[13] S. Segura, JA. Parejo, Robert M. Hierons, D. Benavides, and A. Ruiz-Cortés. ETHOM: An evolutionary algorithm for optimized feature models generation. Tech Report ISA-2011-TR-03 (v. 1.0), Applied Software Engineering Research Group, 2011.

[14] S. Segura and A. Ruiz-Cortés. Benchmarking on the automated analyses of feature models: A preliminary roadmap. In *Third International Workshop on Variability Modelling of Software-intensive Systems*, pages 137–143, Seville, Spain, 2009.

[15] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The variability model of the linux kernel. In *Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS'10)*, Linz, Austria, January 2010.

[16] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the 27th International Conference on Software Engineering*, pages 461–470, 2011.

[17] S.P.L.O.T.: Software Product Lines Online Tools. `http://www.splot-research.org/`, accessed November 2011.

[18] T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *International Conference on Software Engineering*, pages 254–264, 2009.

[19] P. Trinidad, A. Ruiz-Cortés, D. Benavides, and S. Segura. Three-dimensional feature diagrams visualization. In *2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008)*, 2008.

[20] E.J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[21] J. White, B. Dougherty, and D. Schmidt. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, 82(8):1268–1284, 2009.

APPENDIX

A. BeTTy CODE EXAMPLES

Figure 7 illustrates how to use BeTTy to generate a random feature model and save it in FaMa Textual Format.

```
1  // STEP 1: Specify the user's preferences for the generation (so-called characteristics)
2  GeneratorCharacteristics characteristics = new GeneratorCharacteristics();
3  characteristics.setNumberOfFeatures(5);  // Number of features
4  characteristics.setPercentageCTC(100);   // Percentage of cross-tree constraints.
5  // STEP 2: Generate the model with the specific characteristics (FaMa FM metamodel is used)
6  IGenerator generator = new FMGenerator();
7  FAMAFeatureModel fm = (FAMAFeatureModel) generator.generateFM(characteristics);
8  // STEP 3: Save the model
9  FMWriter writer = new FMWriter();
10 writer.saveFM(fm, "./model.afm");
```

Figure 7: Automated generation of a random feature model

Figure 8 despicts a code example showing how to generate a random feature model and its set of products using BeTTy. The model is saved in FaMa XML format.

```
1  // STEP 1: Specify the user's preferences for the generation (characteristics)
2  GeneratorCharacteristics characteristics = new GeneratorCharacteristics();
3  characteristics.setNumberOfFeatures(30);    // Number of features.
4  characteristics.setPercentageCTC(10);      // Percentage of constraints.
5  // Max number of products of the feature model to be generated.
6  characteristics.setMaxProducts(10000);
7  // STEP 2: Generate the model with the specific characteristics (FaMa FM metamodel is used)
8  IGenerator generator = new MetamorphicFMGenerator(new FMGenerator());
9  FAMAFeatureModel fm = (FAMAFeatureModel) generator.generateFM(characteristics);
10 System.out.println("Number of products of the feature model generated: " + generator.
       getNumberOfProducts());
11 // STEP 3: Save the model and the products
12 FMWriter writer = new FMWriter();
13 writer.saveFM(fm, "./model.xml");
```

Figure 8: Automated generation of a random feature model and its set of products

Figure 9 illustrates the use of BeTTy for the generation of a random attributed feature model. The model is saved in FaMa Textual Format.

```
1  // STEP 1: Specify the user's preferences for the generation (so-called characteristics)
2  GeneratorCharacteristics characteristics = new AttributedCharacteristic();
3  characteristics.setNumberOfFeatures(5);  // Number of features
4  characteristics.setPercentageCTC(30);    // Percentage of cross-tree constraints.
5  // STEP 1.1: Add attributes to the model(5 attributes per feature)
6  Random random = new Random();
7  for (int i = 0; i < 2; i++) {
8   int domainMax = random.nextInt(100);
9   int domainMin = random.nextInt(domainMax);
10  characteristics.getAttrubitedModelList().add(new AttributedModel("Attribute" +
11 i, AttributedModel.TYPE_INTEGER, new Range(domainMin,domainMax), new
       IntegerUniformDistributionFunction(domainMin, domainMax)));
12 }
13 // STEP 2: Generate the model with the specific characteristics (FaMa Attributed FM is used)
14 IGenerator generator = new AttributedFMGenerator(new FMGenerator());
15 FAMAAttributedFeatureModel afm = (FAMAAttributedFeatureModel) generator.generateFM(characteristics
       );
16 //STEP 3: Save the model
17 FMWriter writer = new FMWriter();
18 writer.saveFM(afm, "./attributedModel.afm");
```

Figure 9: Automated generation of a random attributed feature model

Figure 10 shows how to use BeTTy to generate a feature model maximizing the execution time invested by FaMa when retrieving the set of products represented by the model. The optimization criteria (i.e. execution time) is provided to the generator as an input fitness function (shown in Figure 11)

```
1  // STEP 1: Specify the user's preferences for the generation (characteristics)
2  GeneratorCharacteristics ch = new GeneratorCharacteristics();
3  ch.setNumberOfFeatures(100);
4  ch.setPercentageCTC(30);
5  // STEP 2: Create a fitness function (i.e execution time in FaMa for is void question.)
6  IFitnessFunction fitnessFunction = new TimeFitness();
7  // STEP 3: Search for a feature model minimizing the branching factor ratio.
8  EvolutionaryFMGenerator generator = new EvolutionaryFMGenerator();
9  generator.setFitnessFunction(fitnessFunction);      // Set fitness function
10 FAMAFeatureModel fm = (FAMAFeatureModel) generator.generateFM(ch);
11 // STEP 4: Save the model
12 FMWriter writer = new FMWriter();
13 writer.saveFM(fm, "./model.xml");
```

Figure 10: Automated generation of a feature model maximizing the analysis time in FaMa

```
1  public class TimeFitness implements IFitnessFunction {
2   public double fitness(FAMAFeatureModel fm) {
3    double stime=System.currentTimeMillis();
4    QuestionTrader qt = new QuestionTrader();
5    qt.setSelectedReasoner("JaCoP");  // We use JaCoP Reasoner
6    ProductsQuestion pq = (ProductsQuestion) qt.createQuestion("Products");
7    qt.setVariabilityModel(fm);
8    qt.ask(pq);
9    return System.currentTimeMillis()-stime;
10  }
11 }
```

Figure 11: Fitness function for maximizing the execution time in FaMa