Application of a Syntax-based Testing Method and Tool to Software Product Lines*

Anamaria Martins Moreira[1], **Cleverton Hentz Antunes**[1], **Viviane de Menezes Ramalho**[1]

[1]Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte (UFRN)
CEP 59.078-970 – Rio Grande do Norte – RN – Brasil

anamaria@dimap.ufrn.br, chentz@ppgsc.ufrn.br,viviane@ppgsc.ufrn.br

Abstract. *Automatic support to the generation and execution of tests is an important research topic, given the need to have quality testing of products within reasonable costs. The use of systematic testing techniques and adequate coverage criteria are means through which this support and quality may be obtained. The LGen (Lua Language Generator) tool aims to contribute to this scenario by generating sentences from the language of a given grammar in such a way that these sentences satisfy some given coverage criteria. In the current paper, we study the applicability of LGen to the domain of software product lines. More specifically, we propose to use LGen to generate testing configurations (specific products) for a family of products in a product line specified by a feature model. We applied the tool to some publicly available feature models of the SPLOT repository and generated test configurations from grammars which formally describe those feature models. We have attained full coverage of the used classical coverage criteria in all examples with a small number of test configurations, a crucial condition for such tests to be practical.*

1. Introduction

It is widely known that testing is a very important and expensive activity of software engineering. Many works then, look for ways of supporting test activities and of bringing more quality to them while reducing costs. Automatic support for the identification of test cases and for the generation of test data are themes of many of these works. In our group, we have developped the LGen (Lua Language Generator) tool [Hentz 2010, Hentz and Moreira 2009] which generates sentences from the language of a given grammar in such a way that these sentences satisfy some given coverage criteria. Each generated sentence is then the input data of a test case. If the program to be tested is a parser, for instance, and the grammar corresponds to the language that should be accepted by this parser, then the expected outcome would be parser success. The coverage cirteria aim, in this case, to provide some confidence that the different constructs of the language have been tested and parsed succesfully. Programming languages are often, however, infinite sets of sentences or programs. Additionally, semantic constraints which are not expressed by the grammar make that many sentences of the language described by the grammar are not actual programs. The consideration of these semantic restrictions is ongoing work.

*This work is partly supported by CAPES and CNPq grants 560014/2010-4 and 573964/2008-4 (National Institute of Science and Technology for Software Engineering-INES, www.ines.org.br).

Nevertheless, it is interesting to note that, in addition to programming languages, there are other categories of languages that are used in the description of problems in specific areas where only the syntactic constraints are necessary for the description. So, we can use the tool to generate tests in these areas. Some examples from the literature are: (1) Software Product Line (SPL) configuration descriptions [Bagheri et al. 2012], (2) Wireless sensor networks settings[Lymberopoulos et al. 2006] and (3) XML(eXtensible Markup Language) specification files [Hoffman et al. 2009].

The basic idea of software product lines is that from a set of features of a family of applications are extracted sub-sets that represent the characteristics of each application. The specification of the available features and restrictions on how they may be composed to define an application is called a *feature model*. From a feature model it is possible to define an application in a process called *feature model configuration*. Thus, the number of possible feature model configurations increase exponentially with the size of the feature model. Furthermore, the effort for testing these configurations concerns running a whole set of tests for each. In this sense, it is necessary to have a reduced set of configurations to be tested [Bagheri et al. 2012]. The present work aims to present an evaluation of the use of LGen in the area of software product lines.

The work presented in this paper studies the application of the LGen tool to the generation of SPL test configurations. Sections 2 and 3 respectively present the necessary information on the SPL application domain and an overview of LGen. In section 4 the scenario is detailed for the case study, its methodology and results, in section 5 related work is presented and in section 6 concluding remarks are made on the results presented in the article.

2. Software Product Line and Feature Models

A software product line is a set of software systems that share a common the set of features. From software product lines is possible to create program families of related programs for a domain. Software product lines aim at generating programs from a set of features. Each feature represents an increment in functionality relevant to stakeholders. Different programs can be created by selecting a particular subset of features [Kästner et al. 2008]. To compose a product family all the features of a set of similar systems are composed into a feature model [Bagheri et al. 2012] which describes the possible configurations of a product in a product line.

A feature model is a set of features hierarchically organized and can typically be classified as: *Mandatory*, *Optional*, *Alternative* and *Or*. An *optional* feature describes functionality that may or may not be present in products from a family of systems. A *mandatory* feature is a feature that is present in all the products of a family of systems. *Alternative* features represent mutually exclusive features, in other words, alternative features define a group in which one, and only one may be present in a product. A *Or* construct indicates that at least one of the sub-features must be selected. In addition to relationships between features, a feature model can also contain constraints between features, these are typically: *Requires* and *Excludes*. In Requires, if a feature A requires a feature B, the inclusion of A in a product implies the inclusion of B in such product. For the second case, if a feature A excludes a feature B, both features cannot be part of the same product [Benavides et al. 2010].

The feature models are represented by a tree whose root node represents a domain, and the other nodes and leafs depict the features [Bagheri et al. 2012]. Figure 1 shows a classical example of a feature model in the software product line community, a set of applications on graphs. One mandatory feature is represented by an edge terminated by a circle filled in black. An optional feature is represented by an edge terminated by a empty circle. An alternative feature (exclusive-or) is represented by edges which are connected by an empty arc. If the arc is full is allowed to choose more than one alternative (OR).

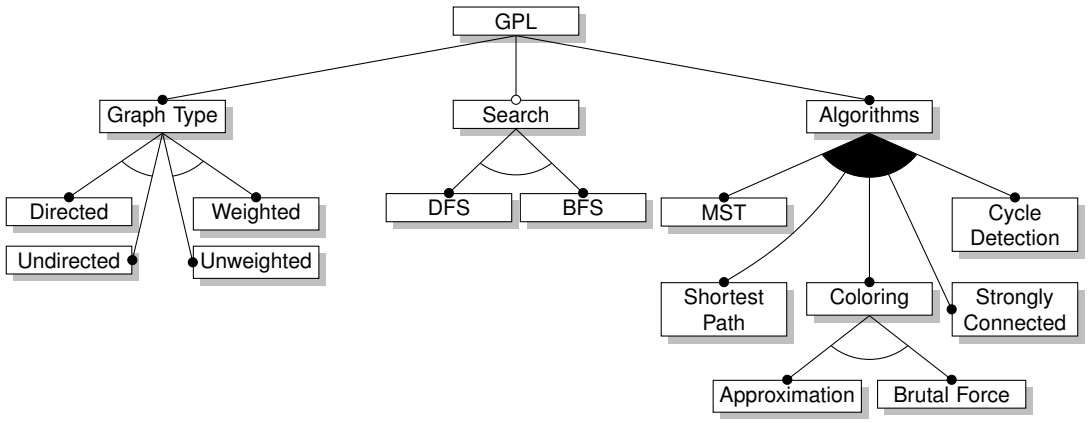

Figure 1. Feature Model GPL [Bagheri et al. 2012].

3. Syntax-Based Testing and LGen

Testing based on grammar descriptions is called *Syntax-Based Testing*, and LGen is a syntax-based testing tool. The main characteristic of syntax-based testing is the use of a syntactic description of some data (usually input data) of the software under test to define the test cases. Grammars are used to formally describe this syntax.

Syntax-based testing has a specific set of test coverage criteria. Three fundamental coverage criteria for grammars are [Ammann and Offutt 2008]:

Terminal Symbol Coverage: The test suite must contain every terminal symbol of grammar.

Production Coverage: The test suite must contain every production rule of grammar.

Derivation Coverage: The test suite must contain every possible string derivable from grammar.

LGen implements two coverage criteria: *Terminal Symbol Coverage* and *Production Coverage*. These criteria are used to limit the number of generated sentences keeping a minimum quality and seeking a good set of tests. Derivation Coverage is attained, when possible, when no other coverage criterion is used, but although it is of theoretical interest, this criterion is impractical, because the number of derivations is often too big or even infinite.

3.1. LGen

The Lua Language Generator (LGen) [Hentz and Moreira 2009] [Hentz 2010] is a sentences generator based on syntax description and which uses coverage criteria to restrict the set of generated sentences. This generator takes as input a grammar described in a

notation based on Extended BNF (EBNF) and returns a set of sentences of the language corresponding to this grammar. The process of generating sentences is divided into two phases. The first is the translation of the grammar described in EBNF to a specification language described in Lua [Ierusalimschy 2006]. In the second, the generated Lua specification is used to generate a set of sentences. Each phase is performed by one component: the first by component **translator**, presented briefly below, responsible for phase translation, and the second by component **generation engine** responsible for the generation itself. Figure 2 shows the architecture of the process of sentences generation, where their components are illustrated by circles and information of input/output is denoted by squares.

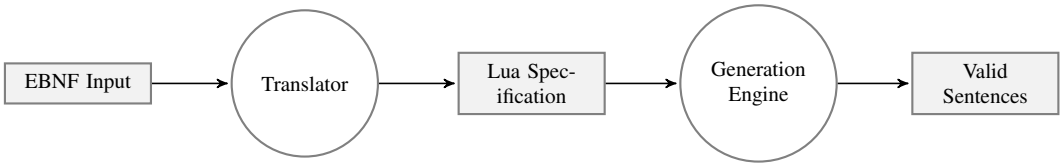

Figure 2. LGen architecture.

The input of the translator component uses a notation based on EBNF. This notation is basically described by the operators of sequence, alternative, repetitions and optional. Table 1 summarize and describe these operators. Each production rule of the grammar is defined by the notation `<NTerm> = <Exp>;`, where `NTerm` is a non-terminal being defined and `Exp` is an expression that defines it.

Operator	**Syntax**	**Description**
Sequence	`Term1 , Term2`	A term `Term1` is generated and then `Term2` will be generated.
Alternative	`Term1 | Term2`	A term `Term1` or `Term2` can be generated.
Optional	`[Term1]`	A term `Term1` may or not be generated.
One Or More	`Term1+`	A term `Term1` can be generated one or more times.
Zero Or More	`{Term1}`	A term `Term1` can be generated zero or more times.

Table 1. Summary of operators supported by the input notation.

The input of LGen is a file containing the description of a context-free grammar in EBNF notation. This file is passed to the translator component that translates this content and generates a second file. This file contains the Lua specification which is required for execution of the generation engine. After this, we can run the generation engine providing execution parameters. These parameters will set limitations on the generation process which are necessary due to the possibility of working with grammars that generate infinite languages and coverage criteria used in process.

The first execution parameter used is *maxCycles* that controls the maximum number of recursion cycles of a non-terminal used in the process of sentences generation. The second parameter is *maxDerivLen* that controls the height of the derivation tree associated with the generated sentences. Finally, the parameter *coverCrit*, configures a coverage criterion to direct the generation process. Currently, the options are the classical ones already presented (terminal, production and no criteria/derivation). The tools has been designed,

however, to be easily adapted with the inclusion of new criteria.

Once the parameters passed in the execution, the generation engine is executed and it results in two files. The first file has an extension *out* that contains the sentences, and second, with extension *prf*, contains information about the generation process. In addition, information such as coverage criteria, coverage rate of the selected criteria, number of generated sentences, memory used and elapsed run time of the generation engine are presented on the screen.

4. Case Study

For our work we have selected three feature models that are available through the SPLOT website. SPLOT is a repository containing feature models for Software Product Lines [Mendonca et al. 2009]. SPLOT allows the user to edit, analyze, debug, configure, share and download feature models. Since its launch in May 2009, it has been visited by several research groups from over 15 countries.

The feature models used for our study were *GPL*, *HIS* and *Model transformation*, which were also used for the generation of product configurations for testing in [Bagheri et al. 2012]. The Graph Product Line (GPL) is the feature model of graphs presented in section 2. The Home Integration System (HIS) is a feature model representing integrated systems of a house [Kang et al. 2002] and Model transformation represents a feature model of transformations for taxonomy [Czarnecki and Helsen 2003]. These feature models were converted to a EBNF grammar through a tool developed in the work of [Bagheri et al. 2012]. To perform our study, we used these converted grammars. We performed a case study for the three grammars, however, we only present details of the study the grammar GPL.

Figure 3 shows the grammar EBNF of the feature model GPL. Minor changes were made to the grammar generated by the original tool to adapt the notation generated to the one accepted by LGen, e.g. empty symbol. This grammar has sixteen (16) non-terminals, thirteen (13) terminals and twenty eight (28) production rules. Once the specification Lua is generated, it is sent to the generator that will generate a set of sentences.

4.1. Results and Analysis

This case study was performed on a MAC OS 10.8.4, configured with one Intel Core 2 Duo 2.26 Ghz and 8GB of memory. Each grammar has been executed in three modes: without the coverage criteria, using the terminal coverage criterion and the production coverage criterion.The results of the tests are presented in tables 2 and 4. For each table we presented the following information: (1) coverage criterion used, (2) configuration of parameter *maxCycles*, (3) configuration of parameter *maxDerivLen*, (4) number of sentences generated and (5) run time of generation. The run time of generation is presented to give a notion of the cost of sentences generation. This is presented in the format *mm:ss.zzz* where *mm* is minutes, *ss* is seconds and *zzz* is milliseconds.

In every run of LGen with the three studied models the specified coverage criterion was 100% satisfied by the generated sentences, except for the derivation criterion when the language is infinite. This is one the goals of our work. Another important quality of a test set is its size, which should not be too big to be practical. To evaluate this

```
GPL = GraphType , Algorithms |
      GraphType , Algorithms , Search;
GraphType = _r_1_2 |
            _r_1_5;
LHS-directed = "t_directed" , "t_Strongly_Connected";
_r_1_2 = LHS-directed |
         "t_undirected";
_r_1_5 = "t_weighted" |
          "t_unweighted";
search = _r_8_9;
LHS-DFS = "t_DFS" , "t_Cycle_Detection";
_r_8_9 = LHS-DFS |
         "t_BFS";
Algorithms = _r_12_13;
_r_12_13-sec = |
               _r_12_13;
LHS-Cycle Detection = "t_Cycle_Detection" , "t_DFS";
LHS-Strongly_Connected = "t_Strongly_Connected" , "t_directed";
_r_12_13 = "t_Shortest Path" , _r_12_13-sec |
            Coloring , _r_12_13-sec |
            LHS-CycleDetection , _r_12_13-sec |
            "t_MST" , _r_12_13-sec |
            LHS-Strongly_Connected , _r_12_13-sec;
Coloring = _r_12_13_16_20;
_r_12_13_16_20-sec = |
                     _r_12_13_16_20;
_r_12_13_16_20 = "t_Approximation" , _r_12_13_16_20-sec |
                  "t_Brute_Force" , _r_12_13_16_20-sec;
```

Figure 3. Grammar extracted from GPL feature model.

caractheristic, for each of the studied models the number of generated sentences and the generation time for the derivation coverage criterion was used as a reference for comparison. In the case of models GPL and Model Transformation the language generated by the input grammar is infinite, so it is not possible to cover the derivation coverage criterion. Thus, we use as limit the parameters *maxCycles* and *maxDerivLen* with the highest values obtained from the terminal criterion and production criterion of each model.

Criterion	Cycles	MaxDerivLen	Sentences	Run time
Derivations	1	8	840	0:00.18
Productions	1	8	13	0:00.03
Terminals	0	6	9	0:00.02

Table 2. Results of running with grammar GPL

Taking as an example GPL grammar, for the terminal criterion, the parameter *maxCycles* shown in table 2 is 0 and *maxDerivLen* is 6, generating 9 sentences. This is the minimum configuration to generate a set of sentences that satisfies the terminal criterion, and increasing values of these settings would not alter the satisfaction of the criterion.

As it can be observed, with the use of terminal or production coverage, there is a reduction on the number of generated sentences and of LGen execution time. For the terminal coverage criterion the number of generated sentences was reduced by 99% and execution time was reduced by 89%, when compared to the generation of all possible

Generated sentences	Terminals covered	Production rules covered
t_directed t_Strongly_Connected t_Shortest_Path	t_directed t_Strongly_Connected t_Shortest_Path	1,3,6,5,14,19,15
t_directed t_Strongly_Connected t_Approximation	t_directed t_Strongly_Connected t_Approximation	1,3,6,5,14,20,27,25
t_directed t_Strongly_Connected t_Brute_Force	t_directed t_Strongly_Connected t_Brute_Force	1,3,6,5,14,20,28,25
t_directed t_Strongly_Connected t_Cycle_Detection t_DFS	t_directed t_Strongly_Connected t_Cycle_Detection t_DFS	1,3,6,5,14,21,17,15
t_directed t_Strongly_Connected t_MST	t_directed t_Strongly_Connected t_MST	1,3,6,5,14,22,15
t_undirected t_Shortest_Path	t_undirected t_Shortest_Path	1,3,7,14,19,15
t_weighted t_Shortest_Path	t_weighted t_Shortest_Path	1,4,8,14,19,15
t_unweighted t_Shortest_Path	t_unweighted t_Shortest_Path	1,4,9,14,19,15
t_directed t_Strongly_Connected t_Shortest _Path t_BFS	t_directed t_Strongly_Connected t_Shortest_Path t_BFS	2,3,6,5,14,19,15,10,13

Table 3. Generated sentences with terminals criterion

derivations up to the given size limit. And for the production coverage criterion the number of generated sentences was reduced by 98% and execution time was reduced by 83%, when compared to the generation of all possible derivations up to the given size limit.

Table 3 details generated sentences to GPL grammar using the terminal criterion. Furthermore, we present the terminals covered and the production rules used in each generated sentence. Table 4 we present the results for HIS and *Model Transformation* grammars. As it can be observed, with the use of terminal or production coverage, there is a reduction on the number of generated sentences and of LGen execution time for two grammars. In all cases, in both coverage criteria the number of generated sentences was reduced by 99.9% and execution time was reduced by 99.9%, when compared to the generation of all possible derivations up to the given size limit. For the *Model Transformation* grammar, the number of derivations is 1.639087061102e+26. But it would never be able to generate these sentences.

Grammar	Criterion	Cycles	MaxDerivLen	Sentences	Run time
HIS	Derivations	0	8	17920	0:14.16
	Productions	0	8	17	0:00.08
	Terminals	0	8	13	0:00.04
Model Transformation	Derivations	1	12	1.639087061102e+26	-
	Productions	1	12	59	0:00.99
	Terminals	0	10	40	0:00.11

Table 4. Results of running with grammars HIS and *Model Transformation*.

As expected, then, we observed that the use of coverage criteria lead to a significantly reduced number of sentences and execution time, when compared to the generation of all possible derivations or all possible derivations up to the given size limit.

5. Related Work

The work presented by [Batory 2005] and [Czarnecki et al. 2004] shows the relationship between grammars and feature models. However, we identified only one work that relates grammars with test generation for software product line feature models [Bagheri et al. 2012]. In this work, the feature models are represented using context-free grammars. The authors proposed eight coverage criteria for product line feature models. Furthermore, they test suite generation for each coverage criteria, using feature models

available through the SPLOT website. In work presented by [Hoffman et al. 2011] the grammar was applied to the Voip software configuration generation.

Other works also use grammars and are potential areas for the use of LGen which will be exercised as future work. Some examples are firewall testing [Hoffman et al. 2010], network protocols [Kaksonen 2001] and bioinformatic [Weinberg and Nebel 2010].

6. Conclusions

In this study we exercised the application of a syntax-based test generation tool to the problem of defining products to be tested in a family of a software product line. Given the high number of possible products in most real software product lines, this is an important problem to be dealt with. We showed that this approach is successful in significantly reducing the number of configurations to be tested, while completely covering traditional coverage criteria. Terminal coverage, for instance, insures that every feature implementation will appear in at least one tested product.

Future work on this area is to implement new criteria, as, for instance the specific ones proposed in [Bagheri et al. 2012], which tries to include in the test configurations situations where the interaction among features may be faulty and the absence of features may also lead to a faulty product. LGen has been designed so that the implementation of new criteria is orthogonal to the generation kernel, making it easier to include such extensions.

Furthermore, on the specific SPL application domain, we could probably improve the grammar generated from the feature model. We have noticed that when the feature model includes the OR construct on features the grammar generated by the tool in [Bagheri et al. 2012], which we used, is recursive, as it happened in the GPL model. This is not however the only solution that can be adopted. We expect a non recursive grammar for the same model to give better results in the generation process.

References

Ammann, P. and Offutt, J. (2008). *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA.

Bagheri, E., Ensan, F., and Gasevic, D. (2012). Grammar-based test generation for software product line feature models. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '12, pages 87–101, Riverton, NJ, USA. IBM Corp.

Batory, D. (2005). Feature models, grammars, and propositional formulas. In *Proceedings of the 9th international conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg. Springer-Verlag.

Benavides, D., Segura, S., and Cortés, A. R. (2010). Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636.

Czarnecki, K. and Helsen, S. (2003). Classification of model transformation approaches.

Czarnecki, K., Helsen, S., and Eisenecker, U. (2004). Staged configuration using feature models. In *Software Product Lines: Third International Conference, SPLC 2004*, pages 266–283. Springer-Verlag.

Hentz, C. (2010). Geração automática de testes a partir de descrições de linguagens. Master's thesis, UFRN, Natal, RN.

Hentz, C. and Moreira, A. M. (2009). Geração de sentenças para testes a partir de descrições de linguagens. In *SAST*, pages 51–60, Gramado, RS, Brasil.

Hoffman, D., Wang, H.-Y., Chang, M., and Ly-Gagnon, D. (2009). Grammar based testing of html injection vulnerabilities in rss feeds. In *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, TAIC-PART '09, pages 105–110, Washington, DC, USA. IEEE Computer Society.

Hoffman, D., Wang, H.-Y., Chang, M., Ly-Gagnon, D., Sobotkiewicz, L., and Strooper, P. (2010). Two case studies in grammar-based test generation. *J. Syst. Softw.*, 83(12):2369–2378.

Hoffman, D. M., Ly-Gagnon, D., Strooper, P., and Wang, H.-Y. (2011). Grammar-based test generation with yougen. *Softw. Pract. Exper.*, 41(4):427–447.

Ierusalimschy, R. (2006). *Programming in Lua*. Lua.org, Rio de Janeiro, RJ, Brazil, 2 edition.

Kaksonen, R. (2001). A functional method for assessing protocol implementation security.

Kang, K. C., Lee, J., and Donohoe, P. (2002). Feature-oriented project line engineering. *IEEE Softw.*, 19(4):58–65.

Kästner, C., Apel, S., and Kuhlemann, M. (2008). Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 311–320, New York, NY, USA. ACM.

Lymberopoulos, D., Ogale, A. S., Savvides, A., and Aloimonos, Y. (2006). A sensory grammar for inferring behaviors in sensor networks. In *IPSN*, pages 251–259.

Mendonca, M., Branco, M., and Cowan, D. (2009). S.p.l.o.t.: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 761–762, New York, NY, USA. ACM.

Weinberg, F. and Nebel, M. E. (2010). Extending stochastic context-free grammars for an application in bioinformatics. In *Proceedings of the 4th international conference on Language and Automata Theory and Applications*, LATA'10, pages 585–595, Berlin, Heidelberg. Springer-Verlag.