Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis

José A. Galindo, David Benavides and Sergio Segura
Department of Computer Languages and Systems
Av Reina Mercedes S/N, 41012 Seville, Spain
{jagalindo, benavides, sergiosegura} at us.es

***Abstract*—The automated analysis of variability models in general and feature models in particular is a thriving research topic. There have been numerous contributions along the last twenty years in this area including both, research papers and tools. However, the lack of realistic variability models to evaluate those techniques and tools is recognized as a major problem by the community. To address this issue, we looked for large–scale variability models in the open source community. We found that the Debian package dependency language can be interpreted as software product line variability model. Moreover, we found that those models can be automatically analysed in a software product line variability model-like style. In this paper, we take a first step towards the automated analysis of Debian package dependency language. We provide a mapping from these models to propositional formulas. We also show how this could allow us to perform analysis operations on the repositories like the detection of anomalies (e.g. packages that cannot be installed).**

I. Introduction

Software Product Line (SPL) engineering is about building set of related software products. From existing assets, products are assembled reducing production costs and time–to–market. Variability Models are used to represent the set of products of an SPL. Typical variability models are feature models [3], orthogonal variability models [21] or decision models [9]. The automated analysis of variability models deals with the automated extraction of information from variability models [3]. In the last years, a large number of techniques haven been proposed in the literature in order to automatically analyse variability models. In [3], Benavides et al. provide a detailed literature review on twenty years of automated analysis of feature models identifying 30 different analysis operations (e.g. Valid Product and Dead Features). These analyses are mainly performed using logic paradigms such propositional logic, constraint programming or description logic. There are also several commercial and open source tools supporting the analysis of variability models such as *AHEAD tool suite* [2], *DecisionKing* [8], *FaMa* [4], *pure::variants* [20] *S.P.L.O.T* [17] and *VarMod (OVM)* [25].

The lack of realistic and large–scale variability models to evaluate and compare the performance of analysis tools is acknowledged to be a challenge in the SPL community [1], [23], [11], [24], [13]. Although there are references to real feature models with hundreds or even thousands of features [1], these are not available for the research community. This has led authors to use trivially small feature models as case studies or to generate the models randomly [23]. Therefore, realistic models are highly desirable in order to assess the scalability of analysis tools.

In order to look for realistic and large–scale variability models, we decided to look into the open source community. We found that Debian based linux distributions have a language to describe dependencies in software packages repositories. They are used in Debian individual installations in order to add, remove or update packages and manage their possible conflicts and dependencies [16]. We found that this language can be interpreted as a variability language similar to those of feature models, OVMs or decision models. Thus, a Debian–based linux distribution could be interpreted as a product line and the individual installations could be therefore considered the products of the product line. In this context, a product is a set of installed packages, i.e. a Debian installation. Similarly, we assume that a set of repositories define an SPL of Debian-based distributions. We may remark that this could also be applied to other dependency languages such as RPM [14], [10] (previusly called Red-Hat Package Manager) but for simplicity they are out of scope of this paper.

The main contribution of this paper is twofold. First, we suggest that Debian packages dependency language can be considered as a variability language. Second, we provide a mapping from this language to propositional formulas enabling their analysis by means of SAT solvers. As a part of our contribution, we present a motivating example showing how our approach could be used to detect uninstallable packages in the repositories, a task that is currently done by hand. Finally, we explain how we plan to integrate our approach in the FaMa Framework, a tool we developed, enabling an efficient and user-friendly extraction of Debian variability models and their analysis .

The rest of the paper is structured as follows. Section II describes the main elements of the Debian package dependency language. Section III proposes a graphical view of Debian variability models. A motivation example is presented in Section IV clarifying how the Debian packaging system describes a SPL and how can the automated analysis help in the detection of inconsistencies and redundancies. We show a mapping from *Debian Variability Modelling Language* (DebianVML) to propositional formula in Section V to enable the automated reasoning of it.

II. Debian package dependency language

This section describes the main elements of the Debian package dependency language. A Debian–based installation is associated with a set of package repositories. A package is a unit of software that can be installed and configured. Each package repository is associated with a configuration file written in the Debian package dependency language including information about the software packages and their relationships/dependencies.

Figure 1 shows a fragment of a repository configuration file. For each package in the file, a set of properties are presented. A property is composed of a name and one or more values associated with it. Properties may be divided into those providing basic information about the package (e.g. name, version) and those describing the relationships with other packages. These relationships can be divided into:

- **Depends.** Package A depends on package B if B must be installed for A to work properly. Sometimes, A depends not only on B, but on a specific version of it. For instance, in Figure 1, line 6, describes that package openoffice.org-gnome depends on package libc6 with version greater or equal to 2.1.3. If the relationship requires package B to be properly configured for the installation of A, the relationship is referred to as **Pre-Depends**. Note that a installed package could not be properly configured.
- **Suggests.** Package A suggests package B if B contains files that are related to (and usually enhance) the functionality of A. In Figure 1, line 8, package openoffice.org-gnome suggests package openoffice.org-evolution. When the suggestion of the package is strong and it is intended to be accepted by most of the users this relationship is usually referred to as **Recommends**. Note that is a human who determines if a the relation should be suggests or recommends.
- **Conflicts.** Package A conflicts with package B when A will not operate if B is installed on the system. In most cases, conflicts are cases where A contains files which are an improvement of those in B. For example, in Figure 1, line 9, package openoffice.org-gnome conflicts with package openoffice.org2-gnome in versions prior to 1:2.4.0-3ubuntu6.
- **Replaces.** Package A replaces package B when files installed by B are removed and in some cases over-written by files in A. In Figure 1, line 4, package openoffice.org-gnome replaces package openoffice.org-common in versions prior to 2.0.4 rc1-0. When this relationship affects only a part of the functionality of the package it is named **Provides**.

III. A Graphical Notation for Debian Dependencies

In order to improve the understandability and readability of the paper we propose a simplified graphical notation [18] for the Debian package dependency language described in the previous section. We will refer to this notation as *Debian*

```
1  Package: openoffice.org-gnome
2  Essential: No
3  Version: 1:2.4.0-3ubuntu6
4  Replaces: openoffice.org-common (<< 2.0.4~
    rc1-0),
5  Provides: openoffice.org-gtk-gnome,
    openoffice.org2-gnome
6  Depends: gconf2, libc6 (>= 2.1.3)
7  Pre-Depends: dpkg (>= 1.14.12ubuntu3)
8  Suggests: openoffice.org-evolution
9  Conflicts: openoffice.org2-gnome (<<
    1:2.4.0-3ubuntu6)
10 Task: ubuntu-desktop, edubuntu-desktop,
    gobuntu-desktop
```

Figure 1. Sample Package Definition

Variability Modelling Language (DebianVML). In this reduced notation we used the a concatenation of the name and version of the packages as the name of the nodes. We omit in this reduced visual notation the relationships of versions. In Figure 2, we present the graphical symbols used to represent each relationship among packages: depends, pre-depends, conflicts, suggests, recommends, replaces and provides. Notice that some of these relationships are represented with the same notation since they have similar meaning as described in previous Section.

Figure 3 shows an example using this notation. As illustrated, the repository is represented as a graph in which nodes represent packages and edges the relationships among packages. Each package is represented as a rectangle labelled with the name and version of the package. In this particular example, we also include a capital letter to simplify later references to the packages in the paper.

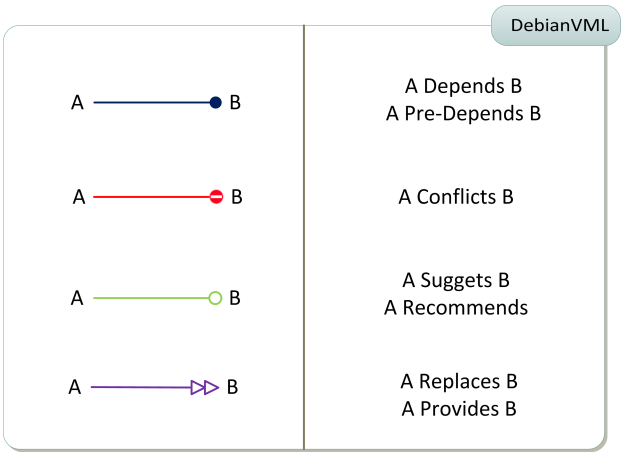

Figure 2. DebianVML graphical view

IV. A motivating example

Consider the example in Figure 3, it represents a valid DebianVML instance composed by four packages and with three different types of relationships: Depends, Conflicts, Replaces.

The model in Figure 3 represents the following products :

1) { }
2) {D}.
3) {C}.
4) {B, C}.
5) {B, D}
6) {A, B, D}

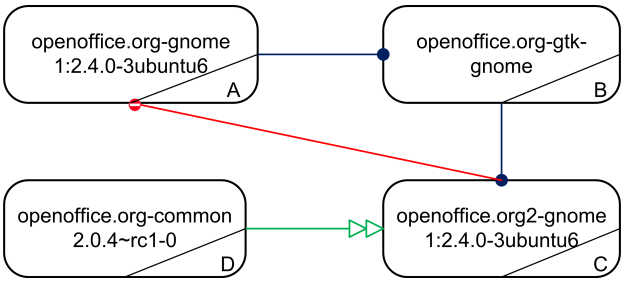

Figure 3. DebianVML example

Notice that the empty product is allowed in this variability model as it represents a Linux distribution without installed packages. Although this could be controversial, a distribution completely void, even without the Linux Kernel, could be considered valid as even the kernel can be installed or uninstalled.

There are also other potential products, that are invalid, for example the product $\{A\}$ is not valid because the **depends** relationship with B package is not satisfied.

An example of how the automated analysis could help in this circumstance is shown if we apply the *valid product* operation [28]. This analysis operation takes a feature model and a product as inputs and determines whether the product is valid for the model. Applying the same idea to Debian configurations, we could determine that $\{A, B, C\}$ (Figure 4) is not a valid instantiation, because the inconsistency marked by the **conflicts** relationship between packages *A* and *C*. In contrast, the product $\{A, B, D\}$ (Figure 5) is a valid product since it does not violate any constraint.

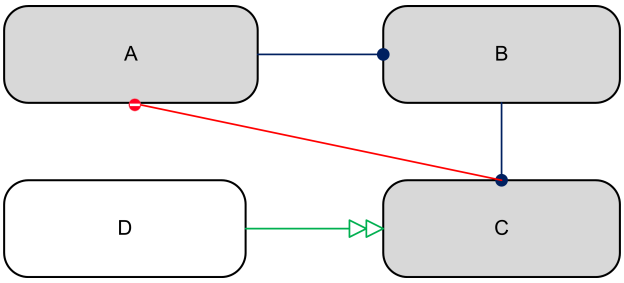

Figure 4. Not Valid Product

Another analysis operation of feature models that can be used in the Debian community is the so–called *dead features*. A feature is dead when the feature does not appear in any valid configuration. In the context of Debian, we can define a *dead package* as the one that is present in a repository but can never be installed because of the violation of dependencies.

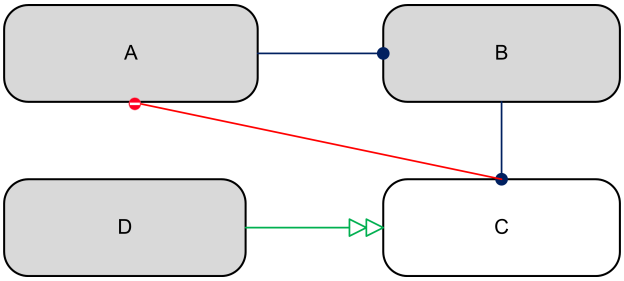

Figure 5. Valid Product

Figure 6 shows an example where A is a dead package. Detecting dead packages in repositories increase the quality of the user experience, preventing the user for an unsupported and frustrating operation. Actually this operation is done manually by the members of the different Debian based distributions. Using the mechanism provided by automated analysis in Debian variability models this task can be done automatically saving time into the development of those distributions.

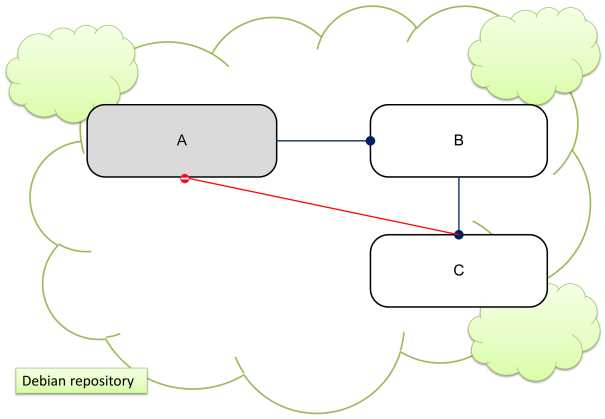

Figure 6. DebianVML with dead package

In our preliminary tests using Ubuntu 8.04 [27] we detected some of these kind of repository inconsistencies. Note that Ubuntu distribution uses the Debian packaging system Figure 7 shows a real example of how a package, wysihtml-le, that is in the *universe* repository can never be installed. Dead Packages can be detected using similar techniques as we do with other kinds of variability models so the knowledge can be reused. The complexity of these variability models are usually bigger than others used in the software product line community having up to 24780 packages and 159812 packages dependencies in Ubuntu 8.04 with *main*, *multiverse*, *restricted* and *universe* repositories enabled.

V. Mapping DebianVML to Propositional Formula

To enable the automated analysis of DebianVML we must define the mapping from DebianVML to constraints into an specific off–the–shelf solver such as Sat4j [5], Choco [6] or

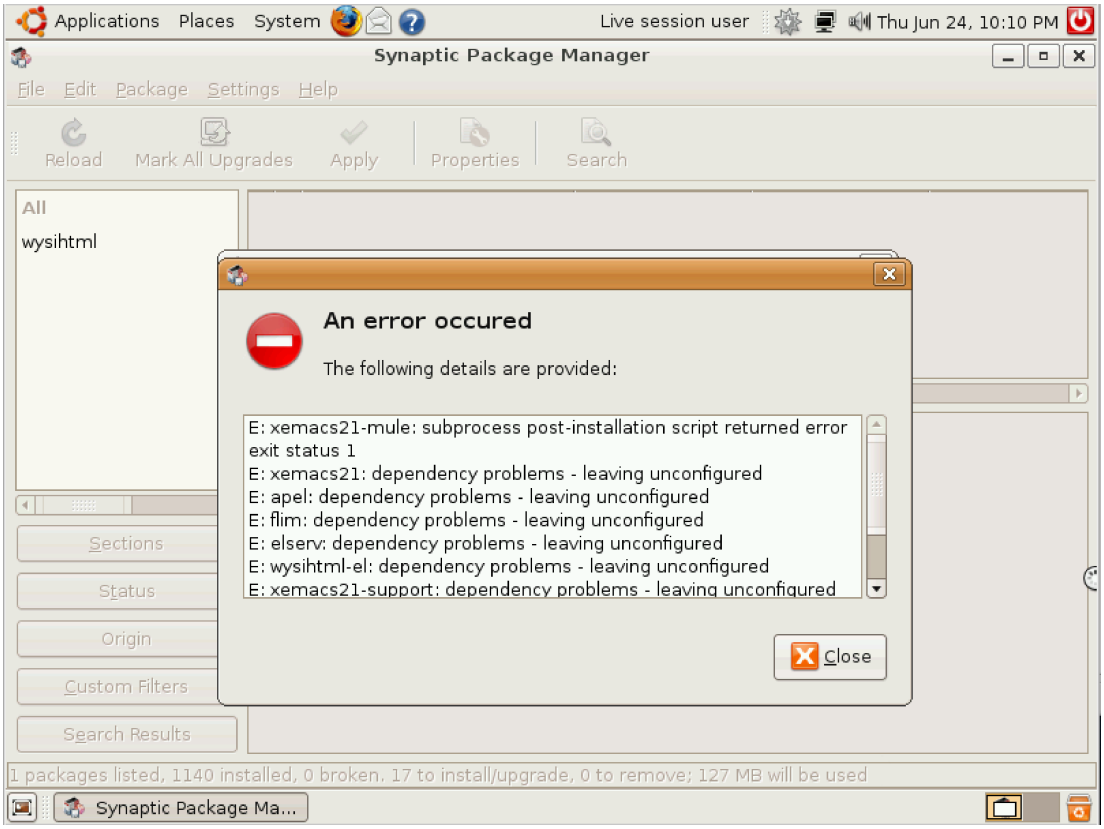

Figure 7. Real demo of a dead package in Ubuntu 8.04

JaCoP [12]. Note that these solvers are widely used for the automated analysis of feature models [3].

In this Section we do not refer to all the relationships as some of them have a very similar meaning and are treated in the same way as the relationships we describe here. Also there are some relationships that do no affect to the variability in the model and does not need to be added to the solver as constraints more exactly suggests and recommends relationships.

The algorithm we followed to translate DebianVML to constraints has the following general form: Each package name make up the set of variables then, each relationship is processed to make up the set of constraints (or propositional clauses) of the problem.

The first relationship to be processed is **Replaces**. This is a special case of relationship as it is needed to generate a set of tuples containing the replaced packages and their replacements to be ready to process the **depends** relationships. This is needed because, after performing a general mapping from Debian dependencies to propositional clauses, we have to change the output clauses according to the replacements that are stored in the tuples. For instance, in the example of Figure 3, we have the tuple (C,D) meaning that D is a replacement of C. Of course, in a real big example, we can have more than a tuple referring to the same package (e.g. (C,X1), (C,X2),... ,(C,Xn) means that C is replaced by X1, X2, ..., Xn).

After processing the **Replaces** relationship we can process the other relationships following the general rules presented in TableI.

Debian Variability Model	Propositional formula
A **Depends** B	$A \Rightarrow (B)$
A **Depends** B (being B replaced by C,D,...)	$A \Rightarrow (B \vee C \vee D \vee ...)$
A **Conflicts** B	$A \Rightarrow (\neg B)$
A **Replaces** B	Used to generate the tuples

Table I
DEBIAN RELATIONS TO PROPOSITIONAL FORMULA

VI. RELATED WORK

There are other contributions interested in solving the existing lack of realistic variability models in the SPL context. Czarnecki et al.[13] present how to extract feature models from the Linux kernel using the LKC (Linux Kernel Configurator) language that describes the relationships in a very similar way as feature models do obtaining feature models with about 6319 features. Our approach focuses on other variability dependency language existing in open source community increasing even more the availability of realistic variability models up to 20000 packages to the SPL community. Although our first attempts tried to map DebianVML to feature models, we finally realised that DebianVML can be considered a variability language by it self.

Recently, Di Cosmo et al. [7] presented a way to describe feature models using Debian packages dependencies and they propose using Debian dependencies management tools to analyse feature models. Our contribution here is the other way around: we propose to use feature model analysis tools and operations to analyse Debian packages repositories. We think that Di Cosmo et al. contribution and ours are good complements to investigate in this direction.

A similar contribution to the problem of the analysis of Debian repositories is presented by some of the results of the EDOS project [19], [15]. Our contribution can benefit their results and we have to study in depth at what extend they have performed analysis operations. We think that we can complement their results providing our know–how in the analysis of feature models and software product lines [3].

VII. Conclusions, future work and discussions

In this paper, we propose using Debian repositories as a good source of realistic and complex variability models. These can be used as motivations inputs problems for those tools dealing with the automation of variability models. At the same time, we consider the Debian community could benefit of this automation to perform several analysis operations like the detection of inconsistencies in repositories, a work that is currently done by hand (e.g. When an error is detected, it is correctly manually).

In our ongoing research, we are working to enable the analysis of DebianVML in FaMa, making FaMa, our tool to analyse feature model, capable to perform this analysis and helping the community of Debian based distributions, applying the knowledge about feature models under Debian VM.

FaMa [26], [4] is a framework to build variability model analysis tools, that was designed with extensibility in mind. It offers a platform to implement other kind of variability models. It has already been used to analyse feature models [26] and also OVMs [22]. Because the analysis of variability models can be very expensive in terms of memory and time, FaMa also offers a set of tools for benchmarking [23] over models. These features, coupled to our familiarity with the tool as members of the FaMa project, made FaMa as a good candidate to be extended in order to support the analysis of Debian variability models. This is part of our future work.

These are the main directions of our future work:

- Address new operations. Using our experience on feature models (e.g. Development of analysis tools, benchmarking, detection of errors ...) we need to be able to detect anomalies in Debian models such as conditionally dead packages or redundancies, probably the operations with more than one variability model can be very useful to Debian community improving the usability of packaging systems. Also the explanations of the errors to help Debian community on its daily work could help to improve the end user experience.
- Benchmarking and optimization. In our preliminary tests we have detected that the Debian based models automated analysis are computationally very expensive, more than 6 hours in a laptop machine with *main*, *multiverse*, *restricted* and *universe* repositories of Ubuntu 8.04 enabled (24780 packages). We will work on the of the codding to improve those results.

We consider that this work could foster discussion about the following topics:

- Can open source models be considered as SPL models?
- Can the OS community benefit from the knowledge acquired by the SPL analysis community in the last 20 years [3]?
- How advanced are Open Source configuration languages to abstract end–users from technical details?

Acknowledments

We would like to thank the friends and co-worker's that encouraged us in this research and the anonymous reviewers for their helpful comments.

This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT projects SETI (TIN2009-07366) and by the Andalusian Government under ISABEL project (TIC-2533).

References

[1] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, December:45–47, 2006.

[2] Don Batory. Feature-oriented programming and the ahead tool suite. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 702–703, Washington, DC, USA, 2004. IEEE Computer Society.

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

[4] D. Benavides, P. Trinidad, S. Segura, and A. Ruiz-Cortés. Fama framework. http://www.isa.us.es/fama/. accessed May 2010.

[5] D. Le Berre. Sat4j. http://www.sat4j.org/. accessed May 2010.

[6] CHOCO solver, http://choco.emn.fr/. accessed January 2010.

[7] Roberto Di Cosmo and Stefano Zacchiroli. Feature diagrams as package dependencies. In *14th International Software Product Line Conference (SPLC'10)*, 2010.

[8] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. Decisionking: A flexible and extensible tool for integrated variability modeling. In *1st International Workshop on Variability Modelling of Software-intensive Systems*, pages 119–128, 2007.

[9] Deepak Dhungana, Rick Rabiser, and Paul Grunbacher. Decision-oriented modeling of product line architectures. *Software Architecture, Working IEEE/IFIP Conference on*, 0:22, 2007.

[10] FOSDEM'08. http://en.opensuse.org/Package_management/Sat_solver.

[11] Arnaud Hubaux, Andreas Classen, Marcílio Mendonça, and Patrick Heymans. A preliminary review on the application of feature diagrams in practice. In *VaMoS*, pages 53–59, 2010.

[12] K. Kuchcinski and R. Szymanek. Jacop. http://jacop.osolpro.com/. accessed May 2010.

[13] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. Evolution of the linux kernel variability model. In *14th International Software Product Line Conference (SPLC'10)*, 2010.

[14] RPM Package Manager. http://www.rpm.org.

[15] Fabio Mancinelli, Jaap Boender, Roberto di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 199–208, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Debian Policy Manual. http://www.debian.org/doc/debian-policy/ch-relationships.html.

[17] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.p.l.o.t.: software product lines online tools. *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762, 2009.

[18] D. Moody. The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *Software Engineering, IEEE Transactions on*, 35(6):756 –779, nov.-dec. 2009.

[19] EDOS project. http://www.edos-project.org.

[20] Pure-systems. pure::variants. http://www.pure-systems.com/. accessed May 2010.

[21] Fabricia Roos-Frantz. A preliminary comparison of formal properties on orthogonal variability model and feature models. In *VaMoS*, pages 121–126, 2009.

[22] Fabricia Roos-Frantz and Sergio Segura. Automated analysis of orthogonal variability models. a first step. In Steffen Thiel and Klaus Pohl, editors, *First Workshop on Analyses of Software Product Lines (ASPL 2008). SPLC'08*, pages 243–248, Limerick, Ireland, September 2008.

[23] 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.

[24] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof 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.

[25] VarMod-PRIME Tool-Environment. http://www.sse.uni-essen.de/wms/en/index.php.

[26] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A.Jimenez. Fama framework. In *12th Software Product Lines Conference (SPLC)*, 2008.

[27] Ubuntu. http://www.ubuntu.com.

[28] J. White, B. Doughtery, D. Schmidt, and D. Benavides. Automated reasoning for multi-step software product-line configuration problems. In *Proceedings of the Sofware Product Line Conference*, pages 11–20, 2009.