Automated Metamorphic Testing on the Analysis of Software Variability

Sergio Segura, Amador Durán, Ana B. Sánchez, Daniel Le Berre, Emmanuel Lonca and Antonio Ruiz-Cortés

sergiosegura@us.es

Applied Software Engineering Research Group
University of Seville, Spain
December 2013

Technical Report ISA-2013-TR-03

This report was prepared by the

Applied Software Engineering Research Group (ISA)
Department of computer languages and systems
Av/ Reina Mercedes S/N, 41012 Seville, Spain
http://www.isa.us.es/

Support: This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT projects SETI (TIN2009-07366) and TAPAS (TIN2012-32273) and the Andalusian Government projects THEOS (TIC-5906) and COPAS (TIC-1867).

Automated Metamorphic Testing of Variability Analysis Tools

Sergio Segura[1 1], Amador Durán[1], Ana B. Sánchez[1], Daniel Le Berre[2], Emmanuel Lonca[2] and Antonio Ruiz-Cortés[1]

[1]ISA research group, Universidad de Sevilla, Spain
[2]Faculté des sciences Jean Perrin, Université d'Artois, Lens, France

Abstract Variability determines the ability of software applications to be configured and customized. A common need during the development of variability–intensive systems is the automated analysis of their underlying variability models, e.g. detecting contradictory configuration options. The analysis operations that are performed on variability models are often very complex, which hinders the testing of the corresponding analysis tools and makes difficult, often infeasible, to determine the correctness of their outputs, i.e. the well–known *oracle problem* in software testing. In this technical report, we present a generic approach for the automated detection of faults in variability analysis tools overcoming the oracle problem. Our work enables the generation of random variability models together with the exact set of valid configurations represented by these models. These test data are generated from scratch using step–wise transformations and assuring that certain constraints (a.k.a. *metamorphic relations*) hold at each step. To show the feasibility and generalizability of our approach, we used to automatically test several analysis tools in three variability domains: feature models, CUDF documents and Boolean formulas. Among other results, we detected 19 real bugs in seven out of the 15 tools under test.

Key Words: Software testing, metamorphic testing, automated testing, software variability

1 Introduction

Modern software applications are increasingly configurable driven by customer demands, competitiveness and continuous changing business conditions. This leads to software systems which expose a high degree of variability. *Software variability* refers to the ability of a software system to be extended, changed, customized or configured to be used in a particular context [1]. Operating systems as Linux or eCos, for instance, can be configured by installing set of packages, e.g. Debian Wheezy offers more than 37,000 packages [2]. Modern ecosystems and browsers are configured in terms of plug-ins or extensions, e.g. the Eclipse Marketplace currently provides about 1,650 Eclipse plug-ins [3]. Also, cloud applications are increasingly flexible, e.g. the Amazon elastic compute cloud service has 1,758 different possible configurations [4].

Software variability is documented by using variability models. A *variability model* describes all the possible configurations of a system in terms of composable units (a.k.a. variants) and constrains defining the way in which they can be combined. Variability can be modelled either at the problem or at the solution level. At the problem level, variability is managed in terms of features or requirements using variability models such as feature models [5], orthogonal variability models [6] or decision models [7]. At the solution level, variability is modelled using domain-specific languages such as Kconfig in Linux [8], p2 in Eclipse [9] or WS-Agreement in web services [10] (sec. II.B).

The number of configurations and dependencies in variability models is potentially huge. For instance, according to [8], the Linux kernel has 6,320 packages and 86% of them are connected by constraints that restrict their interactions, this is colloquially known as the "*dependency hell*" in the operating system domain [11]. To manage this complexity automated support is primordial. The automated analysis of variability models deals with the computer-aided extraction of information from variability models. These analyses can be catalogued in terms of analysis operations. For instance, given a variability model, typical operations allow us to know whether the model is consistent (i.e. it represents at least a valid configuration), whether a given configuration fulfils the constraints of the model or whether the model contains any errors, e.g. contradictory configuration options. Typical approaches for the analysis of variability models are those based on propositional logic, constraint satisfaction problems or description logic, among others. Tools supporting the analysis of variability models can be found in most of the domains where variability exists. Some examples are the FaMa Framework [12] and the SPLAR tool [13, 14] in the context of feature models, the CDL [15] and APT [16] configurators in the context of operating systems or the dependency analysis tool integrated into Eclipse [9].

Variability analysis tools commonly deal with complex data structures and algorithms, e.g. the FaMa framework has more than 20,000 lines of code. This makes analyses far from trivial and easily leads to errors increasing development time and reducing the reliability of analysis solutions. Testing of variability analysis tool aim at detecting faults that produce wrong analysis results. A test case in the domain of variability analysis is composed of an input (i.e. variability model) plus the expected output of the analysis operation under test. As an example, the feature model in Fig. 3 represents 10 different product configurations which is the expected output of the analysis operation *NumberOfProducts* [17].

Current testing methods on the analysis of variability are either manual or based on redundant testing. Manual methods rely on the ability of the tester to decide whether the output of an analysis is correct. However, this is time-consuming,

error-prone and in most cases infeasible due to the combinatorial complexity of the analyses, this is known as the *oracle problem* [18] i.e. impossibility to determine the correctness of a test output. Redundant testing is based on the use of alternative implementations of the same analysis operation to check the correctness of an output. Although feasible, this is a limited solution since it cannot be guaranteed that such alternative tool exists and that it is error-free.

Metamorphic testing [19, 18] was proposed as a way to address the oracle problem. The idea behind this technique is to generate new test cases based on existing test data. The expected output of the new test cases can be checked by using known relations (so–called *metamorphic relations*) among two or more input data and their expected outputs. Key benefits of this technique are that it overcomes the oracle problem and it can be highly automated. Metamorphic testing has shown to be effective in a number of testing domains including numerical programs [20], graph theory [21] or service–oriented applications [22].

Problem description. In previous works [23, 24], we presented a metamorphic testing approach for the automated detection of faults in feature model analysis tools. Feature models are the de-facto standard for variability modelling in software product lines [5]. For the evaluation of our work, we introduced hundreds of artificial faults (i.e. mutants) into several subject programs and checked how many of them were detected by our test data generator. The percentage of detected faults ranged between 98.7% and 100% which supported the feasibility of our contribution. However, despite the promising results obtained, two research questions remain open, namely:

- *RQ1. Can metamorphic testing be used as a generic approach for test data generation on the analysis of variability?* It is unclear whether our approach could be used to automate the generation of test data in other variability domains beyond feature models. Generalizing our previous work in that direction would be a major step forward in supporting automated testing and overcoming the oracle problem in a number of variability analysis domains, e.g. dependencies in open-source distributions.

- *RQ2. Is metamorphic testing effective in detecting real bugs in variability analysis tools?* Despite the mutation testing results obtained in our previous works, the ability of our approach to detect real bugs is still to be demonstrated. Answering this question is especially challenging since the number of available tools for testing is usually limited and it requires a deep knowledge of the tools under test.

Contribution. In this technical report, we extend and generalize our previous work into a metamorphic testing approach for the automated detection of faults in variability analysis tools. Our approach enables the generation of variability models (i.e. inputs) plus the exact set of valid configurations represented by the models (i.e. expected output). Both, the models and their configurations are generated from scratch using step-wise transformations and making sure that certain constraints (i.e. metamorphic relations) hold at each step. Complex variability models representing thousands of configurations can be efficiently generated by applying this process iteratively. Once generated, the configurations of each model are automatically inspected to get the expected output of a number of analyses over the models. Our approach is fully automated and highly generic being applicable to any domain with common variability constraints. Also, our work follows a black-box approach and therefore it is independent of the internal aspects of the tools under test, e.g. it can be used to test tools written in different programming languages. In order to answer RQ1 and RQ2, we present an extensive empirical evaluation of the ability of our approach to automatically detect faults in three different software variability domains, namely:

- *Feature models*. These are hierarchical variability models used to describe the products of a software product line in terms of features and relations among them [5]. We propose five metamorphic relations for feature models and present a test data generator relying on them. For its evaluation, we automatically tested 19 different analysis operations in three feature model reasoners. We detected twelve faults.

- *CUDF documents*. These are variability documents used to describe variability in package-based Free and Open Source Software distribution [25, 26]. We present four metamorphic relations for CUDF documents and an associated test data generator. For its evaluation, we automatically tested two analysis operations, including an upgradeability optimization operation, in three CUDF reasoners. We detected two faults.

- *CNF formulas*. Among its applications, CNF (Boolean) formulas are extensively used to representation and analyse variability at a low level of abstraction. Many variability models such feature models or decision models can be automatically analysed by translating them into CNF formulas and solving the boolean satisfiability problem (SAT) [17, 7]. Also, SAT technology is used to deal with variability management in software ecosystems such as Eclipse or Linux [9, 27]. We present five metamorphic relations for CNF formulas and a test data generator relying on them. For its evaluation, we automatically tested the satisfiability operation in nine SAT solvers. We detected five faults.

The rest of the report is structured as follows: Section 2 introduces the variability languages used to illustrate our approach as well as a brief introduction to metamorphic testing. Section 3 presents the proposed metamorphic relations for the variability languages under study. Section 4 introduces our approach for the automated generation of test data using metamorphic relations. In Section 5, we evaluate our approach checking the ability of our test data generators to detect faults in a number of variability analysis tools. Section 6 presents the threats to validity of our work. The related works are presented and discussed in Section 7. Finally, we summarize our conclusions in Section 8.

2 Preliminaries

Variability languages are used to describe all the possible configurations of a family of systems in terms of composable units (a.k.a. variants) and constraints restricting the way in which they can be combined. There exists a variety of variability languages spread across multiple software domains. In the following sections, the three variability languages used to illustrate and evaluate our approach are presented, followed by a brief introduction to metamorphic testing.

2.1 Feature models

Feature Models (FMs) are commonly used as a compact representation of all the products in a *Software Product Line* (SPL) [5]. A FM is visually represented as a tree-like structure in which nodes represent features and connections illustrate the relationships between them. These relationships constrain the way in which features can be combined to form valid configurations, i.e. products. For example, the FM in Fig. 1 illustrates how features are used to specify and build software for Global Position System (GPS) devices. The software loaded in the GPS is determined by the features that it supports. The root feature (i.e. 'GPS') identifies the SPL. The different types of relationships that constrain how features can be combined in a product are the following:

- **Mandatory**. If a feature has a mandatory relationship with its parent feature, it must be included in all the products in which its parent feature appears. In Fig. 1, all GPS products must provide support for *Routing*.
- **Optional**. If a feature has an optional relationship with its parent feature, it can be optionally included in all the products including its parent feature. For instance, *Keyboard* is defined as an optional feature of the user *Interface* of GPS products.
- **Set relationship.** A set relationship relates a parent feature with a set of child features using *group cardinalities*, i.e intervals such as $\langle n..m \rangle$ limiting the number of different child features that can be present in a product in which their parent feature appears. In Fig. 1, software for GPS devices can provide support for *3D map* viewing, *Auto–rerouting* or both of them in the same product.

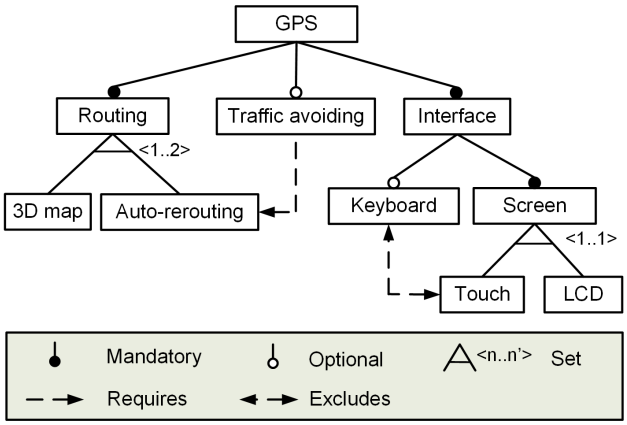

Figure 1: A sample feature model

In addition to hierarchical relationships, FMs can also contain *cross–tree constraints* between features. These are typically of the form "Feature A **requires** feature B" or "Feature A **excludes** feature B". For example in Fig. 1, GPS devices with *Traffic avoiding* require the *Auto–rerouting* feature.

The automated analysis of FMs deals with the computer–aided extraction of information from FMs. Catalogs with up to 30 analysis operations on FMs have been published [17]. Typical analysis operations allow us to know whether a FM is consistent (i.e. it represents at least one product), what is the number of products represented by a FM or whether a FM contains any errors. Common techniques to perform these operations are those based on propositional logic [28], constraint programming [29] or description logic [30]. Also, these analysis capabilities can be found in a number of commercial and open source tools including the *FaMa* framework [12], the *FLAME* framework [31] and SPLAR [13, 14].

2.2 CUDF documents

The *Common Upgradeability Description Format* (CUDF) is a format for describing variability in package–based Free and Open Source Software (FOSS) distributions [25, 26]. This format is one of the outcomes of the *Mancoosi* European research project [32], intended to build better and generic tools for package–based system administration. CUDF combines features of the RPM and the Debian packaging systems, and also allows to encode other formats such as metadata of Eclipse plugins [9]. A key benefit of CUDF is that it permits to describe variability in a distribution and package–manager–independent manner. Also, the syntax and semantics of CUDF documents are well documented, something that facilitates the development of independent analysis tools.

Fig. 2 depicts a sample CUDF document. As illustrated, it is a text file composed by several paragraphs (so–called *stanzas*) separated by an empty line. Each stanza is composed of set of properties, i.e. key/value pairs. The document starts with a so–called *preamble stanza* with meta–information about the document followed by several consecutive *package stanzas*. A package stanza describes a single package known to the package manager and may include, among others, the following properties:

- **Package**. Name of the package, e.g. php5-mysql.
- **Version**. Version of the package as a positive integer. Version strings like "2.3.1a" are not accepted since they have no clear cross–distribution semantics. It is assumed that if each set of versions in a given distribution has a total order then they could be easily mapped to positive integers.
- **Depends**. Set of dependencies indicating the packages that should be installed for this package to work. Version constraints can be included using the operators =, !=, >, <, >= and <=. Also, complex dependencies are supported by the use of conjunctions (denoted by ",") and disjunctions (denoted "|"). As an example, package arduino in Fig. 2 should be installed together with a version of libantlr-java greater than 4 and either any version of openjdk-jdk or sun-java-jdk version 6 or greater.
- **Conflicts**. Comma–separated list of packages that are incompatible with the current package, i.e. they cannot be installed at the same time. Package–specific version constraints are also allowed. In the example, package php5-mysql is in conflict with mysqli.
- **Installed**. Boolean value indicating whether the package is currently installed in the system or not. The default value is false. In Fig. 2, package arduino is installed while the package php5-mysql is not.

```
preamble:
…

package: arduino
version: 6
depends: libantlr-java>4 , openjdk-jdk | sun-java-jdk>=6
installed: true

package: php5-mysql
version: 5
depends: libc, libmysqlclient >= 5
conflicts: mysqli
…

request:
install: apt , apmd , kpdf = 6
remove: php5-mysql
```

Figure 2: A sample CUDF document

The CUDF document concludes with a so–called *request stanza* which describes the user request, i.e. the changes the user wants to perform on the set of installed packages. The request stanza may include three properties: a list of packages to be installed, a list of packages to be removed and a list of packages to be upgraded. Version constraints are allowed in all cases. In the example, the user wishes to install the packages apt, apmd and kpdf version 6 and remove the package php5-mysql.

The automated analysis of CUDF documents is mainly intended to solve the so–called *upgradeability problem* [25]. Given a CUDF document, this problem consists in finding a valid configuration, i.e. a set of packages that fulfills all the constraints of the package stanzas and fulfils all the requirements expressed in the user request. This problem is often turned into an optimization problem by searching not only a valid solution but a good solution according to an input

optimization criterion. For instance, the user may wish to perform the request minimizing the number of changes (i.e. set of installed and removed packages) or minimizing the number of outdated packages in the solution.

The analysis of CUDF documents is supported by several tools that meet annually in the MISC performance competition arranged by the Mancoosi project. In the competition, CUDF reasoners must analyse a number of CUDF documents using a set of given optimization functions. CUDF documents are either random or generated from the information obtained in open source repositories. CUDF reasoners rely on techniques such as answer set programming [33] and pseudo boolean optimization [25].

2.3 CNF formulas

A *Boolean formula* consists of a set of propositional variables and a set of logical connectives constraining the values of the variables, e.g. $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$. Boolean Satisfiability (SAT) is the problem of determining if a given Boolean formula is satisfiable, i.e. if there exists a variable assignment that makes the formula evaluate to true. Among its many applications, Boolean formulas can be regarded as the canonical representation of variability. Many variability models such feature models or decision models can be automatically analysed by translating them into Boolean formulas and solving the SAT problem [17, 7]. Not only that, SAT technology is used to deal with dependency management in software ecosystems such as Eclipse or Linux [9, 27].

A SAT solver is a software package that takes as input a CNF formula and determines if the formula is satisfiable. CNF is a standard form to represent propositional formulas where only three connectives are allowed: $\neg$, $\wedge$, $\vee$. CNF formulas consists of the conjunction of a number of *clauses*, where a clause is a disjunction of *literals*, and a literal is a propositional variable or its negation. As an example, consider the following propositional formula in CNF form: $(a \vee \neg b) \wedge (\neg a \vee b \vee c)$. The formula is composed of two clauses ($(a \vee \neg b)$, $(\neg a \vee b \vee c)$) and three literals (a, b and c). A possible solution for this formula is a=1, b=0, c=1, i.e. the formula is satisfiable.

There exists a vast array of available SAT solvers as well as SAT benchmarks to measure their performance. Every two years a competition is held to rank the performance of the participant's tools. In the last edition in 2013, 93 solvers took part in the SAT competition[2].

2.4 Metamorphic testing

An *oracle* in software testing is a procedure by which testers can decide whether the output of a program is correct [18]. In some situations, the oracle is not available or is too difficult to apply. This limitation is referred in the testing literature as the *oracle problem* [34]. Consider, as an example, checking the results of complicated numerical computations (e.g. Fourier transform) or processing non–trivial outputs like the code generated by a compiler. Furthermore, even when the oracle is available, the manual prediction and comparison of the results are in most cases time–consuming and error–prone.

Metamorphic testing [19, 18] was proposed as a way to address the oracle problem. The idea behind this technique is to generate new tests from previous successful test cases. The expected output of the new test cases can be checked by using so–called *metamorphic relations*, that is, known relations among two or more input data and their expected outputs. As a result, the oracle problem is alleviated and the test data generation process can be highly automated.

Consider, as an example, a program that compute the sine function ($\sin x$). Suppose the program produces the output 0.207 when run with input $x = 12$. A mathematical property of the sine function states that $\sin(x) = \sin(x + 360)$. Using this property as a metamorphic relation, we could design a new test case with $x = 12 + 360 = 372$. Assume the output of the program for this input is 0.375. When comparing both outputs, we could easily conclude the program is faulty.

Metamorphic testing has been successfully applied to a number of testing domains including numerical programs [20], graph theory [21] or service–oriented applications [22].

3 Metamorphic relations on variability models

In this section, a set of metamorphic relations between models of the variability languages presented in Section 2 and their corresponding set of valid configurations is presented. These relations are based on the fact that when a variability model M is modified, depending on the kind of modification, the set of valid configurations of the resulting *neighbour* model M' can be derived from the original one and therefore new test cases can be automatically derived.

[2] `http://www.satcompetition.org`

3.1 Metamorphic relations on feature models

The identified metamorphic relations between neighbour FMs are defined as follows.

MR$_1$: Mandatory. Consider the neighbour FMs and their associated product sets in Figure 3, where M' is derived from M by adding a mandatory feature D as a child of feature A. According to the semantics described in section 2.1, the set of products of M' can be derived by adding the new mandatory feature D in all the products of M where its parent feature A appears.

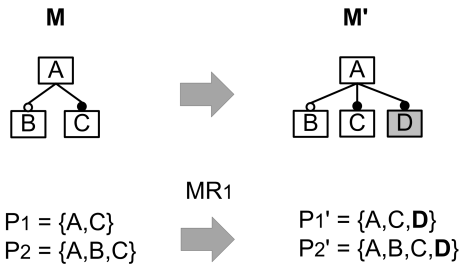

Figure 3: Neighbour models after mandatory feature is added

Formally, let f_m be the mandatory feature added to M, f_p its parent feature, $\Pi(M)$ the function returning the set of products of a FM, and $\#$ the cardinality function on sets. Then, MR$_1$ can be defined as follows:

$$\begin{aligned} &\#\Pi(M') \;=\; \#\Pi(M) \;\wedge \\ &\forall p \in \Pi(M) \;\bullet\; f_p \notin p \;\Rightarrow\; p \in \Pi(M') \wedge \\ &\qquad\qquad f_p \in p \;\Rightarrow\; (\; p \cup \{f_m\}\;) \in \Pi(M') \end{aligned} \tag{MR$_1$}$$

MR$_2$: Optional. When an optional feature is added to a FM, the derived set of products is formed by the original set and the new products created by adding the new optional feature to all the products including its parent feature (see Figure 4). Formally, let f_o be the optional feature and f_p its parent feature. Consider the product selection function $\Pi_\sigma(M, S, E)$ that returns the set of products of M including all the selected features in S and excluding all the features in E. Then, MR$_2$ can be defined as follows:

$$\begin{aligned} &\#\Pi(M') \;=\; \#\Pi(M) \;+\; \#\Pi_\sigma(M, \{f_p\}, \varnothing) \;\wedge \\ &\forall p \in \Pi(M) \;\bullet\; p \in \Pi(M') \wedge \\ &\qquad\qquad f_p \in p \;\Rightarrow\; (\; p \cup \{f_o\}\;) \in \Pi(M') \end{aligned} \tag{MR$_2$}$$

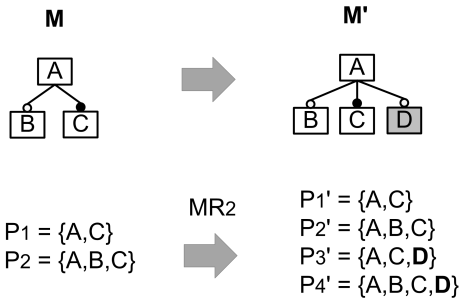

Figure 4: Neighbour models after optional feature is added

MR$_3$: Set relationship. When a new set relationship with a $\langle n, m\rangle$ cardinality is added to a FM, the derived set of products is formed by all the original products not containing the parent feature of the set relationship and the new products created by adding all the possible combinations of size $n..m$ of the child features to all the products including the parent feature (see Figure 5). Formally, let F_s be the set of features added to the model by means of a set relationship

with a $\langle n, m \rangle$ cardinality and a parent feature f_p. Let also be $\wp_n^m F_s = \{S \in \wp F_s \mid n \leq \#S \leq m\}$ the set of all possible subsets of F_s with cardinality in the $\langle n, m \rangle$ interval. Then, assuming that $1 \leq n \leq \#F_s$ and $n \leq m \leq \#F_s$, MR_3 can be defined as follows:

$$\begin{aligned} \#\Pi(M') \;=\; & \#\Pi_\sigma(M, \varnothing, \{f_p\}) \\ & + \#\wp_n^m F_s \cdot \#\Pi_\sigma(M, \{f_p\}, \varnothing) \;\wedge \\ \forall p \in \Pi(M) \;\bullet\; & f_p \notin p \;\Rightarrow\; p \in \Pi(M') \wedge \\ & f_p \in p \;\Rightarrow\; \forall S \in \wp_n^m F_s \;\bullet\; (\, p \cup S \,) \in \Pi(M') \end{aligned} \tag{MR$_3$}$$

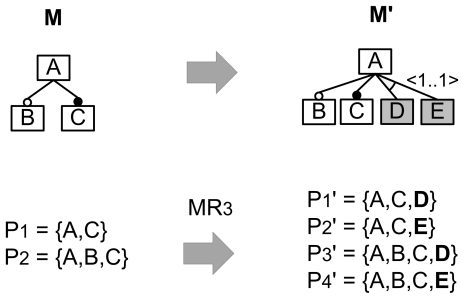

Figure 5: Neighbour models after set relationship with cardinality 1..1 is added

MR_4: Requires. When a new f_1 **requires** f_2 constraint is added to a FM, the derived set of products is the original set except those products containing f_1 but not f_2 (see Figure 6). Formally, MR_4 can be defined as follows using the product selection function Π_σ:

$$\Pi(M') \;=\; \Pi(M) \setminus \Pi_\sigma(M, \{f_1\}, \{f_2\}) \tag{MR$_4$}$$

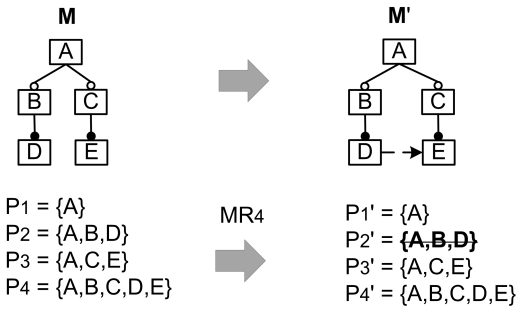

Figure 6: Neighbour models after requires constraint is added

MR_5: Excludes. When a new f_1 **excludes** f_2 constraint is added to a FM, the derived set of products is the original set except those products containing both f_1 and f_2 (see Figure 7). Formally, MR_5 can be defined as follows:

$$\Pi(M') \;=\; \Pi_\sigma(M, \varnothing, \{f_1, f_2\}) \tag{MR$_5$}$$

3.2 Metamorphic relations on CUDF documents

From a variability management point of view, CUDF variants correspond to pairs (p, v), where p is a package identifier and v is a version number. A valid configuration is considered as a set of package pairs $\{\ (p_i, v_i)\ \}$ which can be installed simultaneously satisfying all their dependencies without conflicts.

With respect to CUDF documents, two assumptions have been made in order to keep our metamorphic relations simple. The first one is that, although the CUDF specification [26] allows multiple versions of the same package in the same document and therefore in a configuration, we restrict this to at most one version of the same package. The second

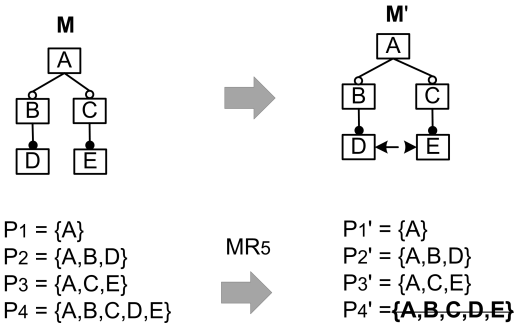

Figure 7: Neighbour models after excludes constraint is added

one is that all CUDF documents must be self–contained, i.e. that all dependencies and conflicts of a given package reference only other packages already present in the same CUDF document. Considering these two assumptions, it is possible to define the following metamorphic relations between the valid configurations of neighbour CUDF documents.

MR$_6$: New package. When a new package is added to a CUDF document, the derived set of valid configurations is formed by the original set, a configuration containing the new package only, and all the original configurations with the new package added (see Figure 8). Formally, let D' be the CUDF document created by adding a package (p, v) to another document D, and $\Psi(D)$ the function returning all the valid configurations of a CUDF document. Then MR$_6$ can be defined as follows:

$$\begin{aligned} \#\Psi(D') \;&=\; 2 \cdot \#\Psi(D) + 1 \wedge \\ \forall c \in \Psi(D) \;&\bullet\; c \in \Psi(D') \wedge \\ &\quad \{\ (p, v)\ \} \in \Psi(D') \wedge \\ &\quad c \cup \{\ (p, v)\ \} \in \Psi(D') \end{aligned} \tag{MR$_6$}$$

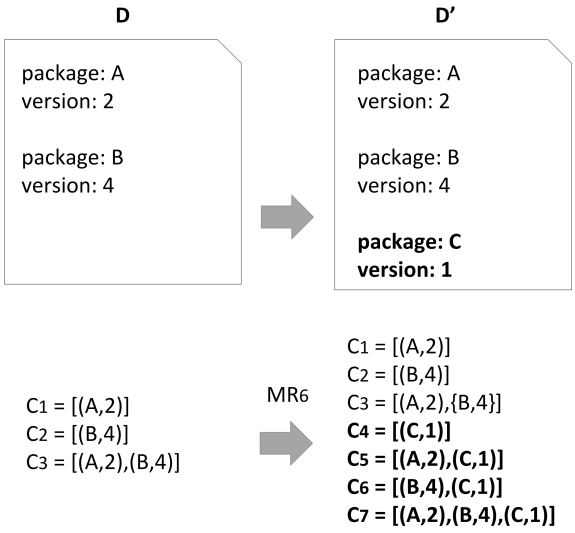

Figure 8: CUDF document after a new package is added

MR$_7$: Disjunctive dependency set. When a new set of disjunctive dependencies is added to a given package (p, v) in a CUDF document, the derived set of valid configurations is formed by all the original configurations satisfying at least one of the added disjunctive dependencies (see Figure 9). Formally, package dependencies in CUDF documents can be represented as 5–tuples (p, v, q, k, θ), where p and q are the identifiers of the depender and dependee packages respectively, v and k are literal version values and θ is a comparison operator. For example, $(\text{arduino}, 2, \text{JDK}, 6, \geq)$ indicates that version 2 of the arduino package depends on the JDK package version 6 or higher.

Let Δ be the set of package dependencies $\{ \delta_i \}$ of the (p, v) package added to a CUDF document, and $\psi(c, \delta)$ a predicate that holds if configuration c satisfies dependency δ. Then MR_7 can be defined as follows:

$$\Psi(D') = \{ c \in \Psi(D) \mid \exists \delta \in \Delta \bullet \psi(c, \delta) \} \quad (\text{MR}_7)$$

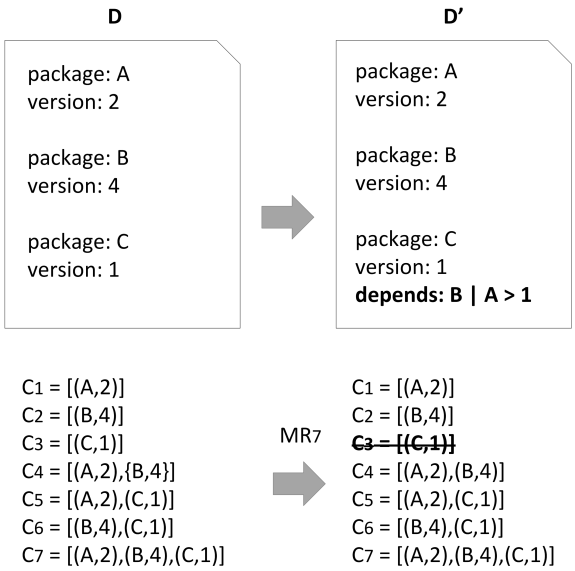

Figure 9: CUDF document after a new set of disjunctive dependencies is added

MR_8: Conjunctive dependency. When a new conjunctive dependency is added to a given package (p, v) in a CUDF document, the derived set of valid configurations is formed by all the original configurations satisfying the added conjunctive dependency (see Figure 10). Formally, let δ be the conjunctive dependency added to the p package in a CUDF document. Then MR_8 can be defined as follows:

$$\Psi(D') = \{ c \in \Psi(D) \mid \psi(c, \delta) \} \quad (\text{MR}_8)$$

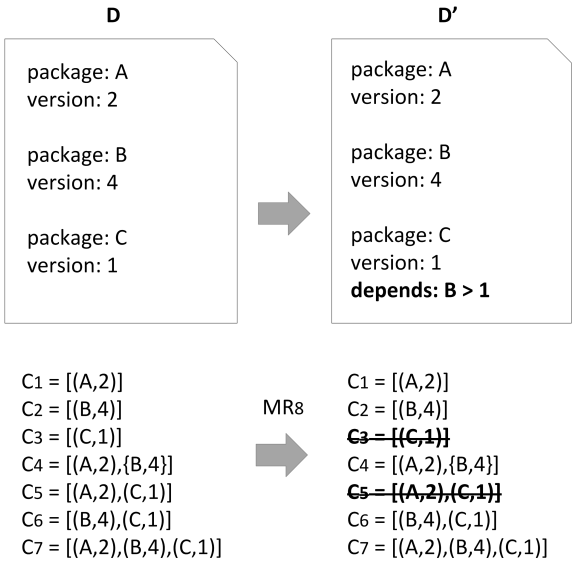

Figure 10: CUDF document after a new conjunctive dependency is added

MR$_9$: Conflict. When a new conflict is added to a given package (p, v) in a CUDF document, the derived set of valid configurations is formed by all the original configurations not affected by the new conflict (see Figure 11). Formally, a conflict can be represented as a dependency that must not hold in a valid configuration. Let κ be the conflict added to the (p, v) package in a CUDF document. Then MR$_9$ can be defined as follows:

$$\Psi(D') \;=\; \{\; c \in \Psi(D) \;\mid\; \neg\psi(\; c, \kappa \;)\;\} \tag{MR$_9$}$$

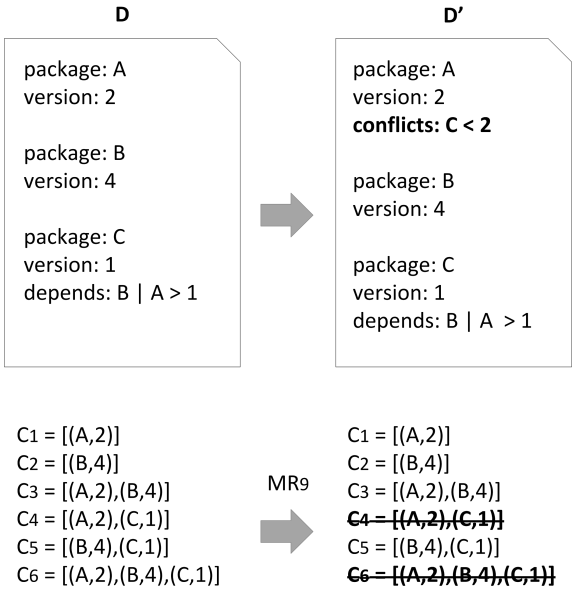

Figure 11: CUDF document after a conflict is added to a package

3.3 Metamorphic relations on CNF formulas

Considering CNF formulas as a way of expressing variability, variants correspond to variables and valid configurations correspond to pairs (V_t, V_f), where $V_t = \{v_i\}$ is the subset of variables set to *true* and $V_f = \{v_j\}$ are the subset of variables set to *false* for a given satisfiable assignment. The following metamorphic relations between the solutions of a Boolean formula in CNF form and the ones of its neighbours have been identified.

MR$_{10}$: Disjunction with a new variable. When a new variable is added to a CNF formula with a single clause, the derived set of solutions is formed by the original set of solutions duplicated by adding the new variable to the *true* and *false* sets of each solution, and a new solution where the new variable is set to *true* and all the others are set to *false* (see Figure 12). Formally, let F' be the CNF formula created by adding a disjunction with a new variable v to a one–clause–only CNF formula F, and SAT the function returning all the solutions of a CNF formula. Then MR$_{10}$ can be defined as:

$$\begin{aligned} \#\text{SAT}(F') \;&=\; 2 \cdot \#\text{SAT}(F) + 1 \;\wedge \\ \forall\; (V_t, V_f) \in \text{SAT}(F) \bullet\; &(\; V_t \cup \{v\},\; V_f \;) \in \text{SAT}(F') \;\wedge \\ &(\; V_t,\; V_f \cup \{v\} \;) \in \text{SAT}(F') \;\wedge \\ &(\; \{v\},\; V_t \cup V_f \;) \in \text{SAT}(F') \end{aligned} \tag{MR$_{10}$}$$

MR$_{11}$: Disjunction with a new negated variable. This metamorphic relation is identical to the previous one except that in the neighbour formula solutions, the new variable is set to *false* and all the others are set to *true* (see Figure 13). Formally, MR$_{11}$ can be defined as follows:

$$\begin{aligned} \#\text{SAT}(F') \;&=\; 2 \cdot \#\text{SAT}(F) + 1 \;\wedge \\ \forall\; (V_t, V_f) \in \text{SAT}(F) \bullet\; &(\; V_t \cup \{v\},\; V_f \;) \in \text{SAT}(F') \;\wedge \\ &(\; V_t,\; V_f \cup \{v\} \;) \in \text{SAT}(F') \;\wedge \\ &(\; V_t \cup V_f, \{v\} \;) \in \text{SAT}(F') \end{aligned} \tag{MR$_{11}$}$$

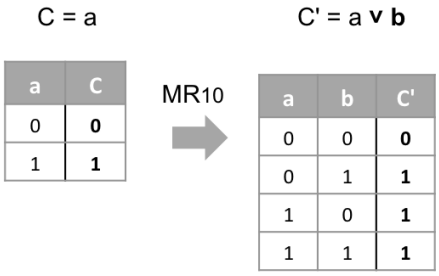

Figure 12: CNF clause after a new variable is added as a disjunction

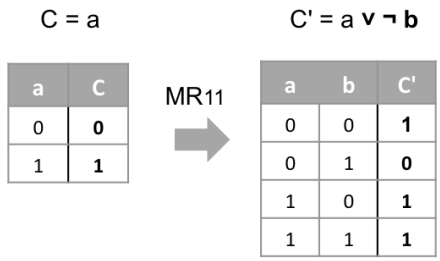

Figure 13: CNF clause after a new negated variable is added as a disjunction

MR$_{12}$: Disjunction with an existing variable. When an existing variable is added to a CNF formula with a single clause (e.g. $F = a \vee b$ and $F' = a \vee b \vee a$), the derived set of solutions is the same as the original one (see Figure 14). Formally, MR$_{12}$ can be defined as follows:

$$\text{SAT}(F') = \text{SAT}(F) \tag{MR$_{12}$}$$

MR$_{13}$: Disjunction with an existing inverted variable. When an existing inverted variable is added to a CNF formula with a single clause (e.g. $F = a \vee b$ and $F' = a \vee b \vee \neg a$), the clause becomes a *tautology*, so any variable assignment becomes a solution (see Figure 15). Formally, let VAR be the function returning all the variables in a CNF formula. Then MR$_{13}$ can be defined as follows, where the new solution set if formed by all the pairs of the cartesian product of the powerset of the variables with itself that form a *partition* over the variable set:

$$\begin{aligned}\text{SAT}(F') = \{\ (V_t, V_f) \in \wp\text{VAR}(F) \times \wp\text{VAR}(F)\ \mid \\ V_t \cup V_f = \text{VAR}(F) \wedge V_t \cap V_f = \varnothing\ \}\end{aligned} \tag{MR$_{13}$}$$

MR$_{14}$: Conjunction with a new clause.

When a new clause is added as a conjunction to a CNF formula with a single clause (e.g. $F = C_1$ and $F' = C_1 \wedge C_2$), the derived set of solutions is formed by those combinations of the sets of solutions of both clauses with no contradictions, i.e. without a given variable set to *true* and *false* simultaneously(see figure **??**)

$$\text{SAT}(F') = \text{SAT}(F) \cap \text{SAT}(F_2) \tag{MR$_{14}$}$$

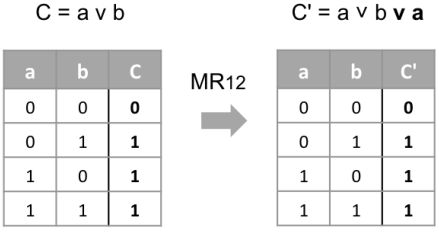

Figure 14: CNF clause after an existing variable is added as a disjunction

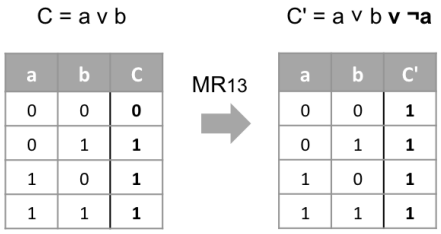

Figure 15: CNF clause after an existing inverted variable is added as a disjunction

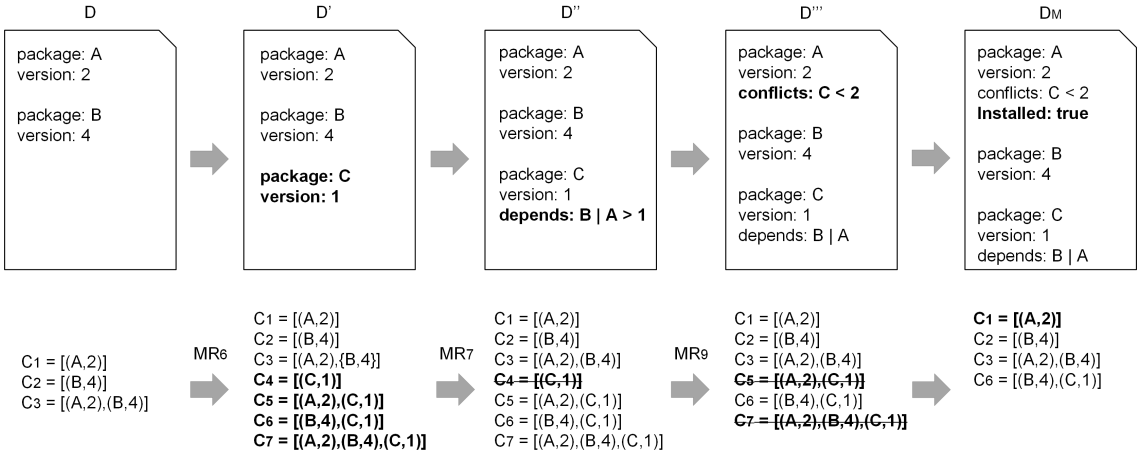

Figure 16: Random generation of a CUDF document and its set of configurations using metamorphic relations

Formally, if C_1 and C_2 are the two CNF clauses to be conjuncted, then MR_{14} can be defined as follows:

$$\begin{aligned} &\forall V_{t_1}, V_{f_1}, V_{t_2}, V_{f_2} \bullet (\ (V_{t_1} \cup V_{t_2}),\ (V_{f_1} \cup V_{f_2})\ ) \in \text{SAT}(C_1 \wedge C_2) \Leftrightarrow \\ &(\ (V_{t_1}, V_{f_1}),\ (V_{t_2}, V_{f_2})\ ) \in \text{SAT}(C_1) \times \text{SAT}(C_2) \wedge \\ &(\ (V_{t_1} \cup V_{t_2}) \cap (V_{f_1} \cup V_{f_2})\ ) = \varnothing \end{aligned} \qquad (\text{MR}_{14})$$

4 Automated test data generation

The semantics of a variability model is defined by the set of configurations that it represents. Most analysis operations on variability models can be answered by inspecting this set adequately. Based on this idea, we propose a two-step process to automatically generate test data for the analyses of variability models as follows:

Variability model generation. We propose using metamorphic relations together with model transformations to generate variability models and their respective set of configurations. Note that this is a singular application of metamorphic testing. Instead of using metamorphic relations to check the output of different computations, we use them to actually compute the output of follow–up test cases. Fig. 16 illustrates an example of our approach. The process starts with an input variability model whose set of configurations is known, i.e. a seed. This seed can be trivially generated from scratch (as in our approach) or taken from an existing test case [24]. A number of step–wise transformations are then applied to the model. Each transformation produces a neighbour model as well as its corresponding set of configurations according to the metamorphic relations. In the example, *D'* is generated by adding a new package (C) to *D*. The set of configurations of *D"* is then easily calculated by making sure that the metamorphic relation MR_6, between the set of configurations of *D'* and the one of *D"*, holds. Transformations can be applied either randomly or using deterministic heuristics. This process is repeated until a variability model (and corresponding set of configurations) with the desired properties is generated. In the example, configuration C1 (i.e. package A) is marked as installed at the end of the process to simulate the current status of the system. Note that this implies no changes in the set of valid configurations.

Test data extraction. Once a variability model with the desired properties is generated, it is used as non-trivial input for the analysis. Similarly, its set of configurations is automatically inspected to get the output of a number of analysis operations i.e. any operation that extracts information from the set of configurations of the model. As an example, consider the CUDF document D_M and its set of configurations generated in Fig. 16. We can obtain the expected output of a number of analyses on the document by inspecting the set of configurations as follows:

- *Is D_M consistent?* Yes, it represents at least a valid configuration.
- *How many different configurations represent D_M?* 4 different configurations.
- *Is $C = [(A, 2), (B, 4)]$ a valid configuration of D_M?* Yes. It is included in its set of configurations, i.e. C_3.
- *Does D_M contain any dead package (i.e. a package that cannot be installed [35])?* No, all packages are included in the set of configurations.

Consider now we include a request stanza in D_M with *Install: B*, i.e. the user wishes to update the current installation by installing the package B. It is easy to observe by checking the set of configurations that the expected configurations fulfilling the user request are C_2, C_3 and C_6 since all of them include package B. More importantly, we can inspect the set of configurations to find out the expected output of certain optimization operations. For instance, the so-called paranoid optimization criterion [25, 32] is used to search for a configuration that fulfils the request and minimizes the number of changes in the system (i.e. number of installed and removed packages). In our example, upgrading the system according to C_2 implies two changes (installing B and uninstalling A), C_3 implies one change (installing B) and C_6 requires three changes (uninstalling A and installing B and C). Therefore, the expected output for the upgradeability problem using the paranoid optimization criterion is the configuration C_3. This expected output can be easily obtained by iterating over the set of configurations and selecting those that: i) satisfy the user request, ii) have a maximum number of preinstalled packages, i.e. those with installed: true, and iii) have a minimum number of packages. Note that upgradeability problems may have more than one possible solution.

Another example is shown in Fig. 17. The figure depicts how our approach is used for the generation of a sample feature model and its set of products. The generation starts with a trivial feature model and its corresponding set of products created from scratch. Then, new features and relationships are added to the model in a step-by-step process. The set of products is updated at each step assuring that the metamorphic relations defined in Section 3.1 hold. For instance, *FM"'* is generated from *FM"* by adding a cross-tree constraint of the form feature G *requires* feature F. According to MR_4, the new set of products must be the one of *FM"* excluding those products containing G but not F. Consider now the feature model FM_M obtained as a result of the process. We can easily find the expected output of most of the analysis operations over the model defined in the literature [17] by simply checking its set of products, for instance:

- *Is FM_M consistent?* Yes, its set of products is not empty.
- *How many different products represent FM_M?* 6 different products.
- *Is P={A,B,F} a valid product of FM_M?* No. It is not included in its set of products.
- Which are the core features of FM_M (i.e. those included in all products)? Features {A,C}.
- *What is the commonality of feature B?* Feature B is included in 5 out of the 6 products of the set. Therefore its commonality is $5/6 = 0.83(83.3\%)$
- *Does FM_M contain any dead feature?* Yes. Feature G is dead since it is not included in any of the products represented by the model.

Finally, Fig. 18 illustrates how metamorphic relations can be used to generate CNF formulas (input) and their respective solutions (output). First, a clause with a single variable and its corresponding set solutions is created ($C_1 = a$). Then, the clause is extended in a set of steps creating successive neighbours. On each step, a new disjunction is added to the clause and the set of solutions is updated using the metamorphic relations $M_{10} - M_{13}$ defined in Section 3.3. This process is repeated until obtaining a set of random clauses $(C_1, C_2...C_n)$ and their respective solutions $(S(C_1), S(C_2)...S(C_n))$. Then, the final formula is created as a conjunction of the clauses previously created, $F = C_1 \wedge C_2... \wedge C_n$. The final set of solutions is calculated using the metamorphic relation M_{14}, which obtains the intersection of the set of solutions of the clauses in the formula, i.e. $S(F) = S(C_1) \cap S(C_2)... \cap S(C_n)$. In the example, F is composed of two clauses (C_1'', C_2''), three variables (a, b, c) and five solutions, i.e. those variable assignments that make the formula evaluate to true.

5 Evaluation

In this section, we evaluate whether our metamorphic testing approach is able to automate the generation of test cases in multiple variability analysis domains ($RQ1$). Also, and more importantly, we explore whether the generated test cases are actually effective in detecting real bugs in variability analysis tools ($RQ2$). For the evaluation, we developed three test data generators based on the metamorphic relations previously defined. Then, we evaluated their ability to automatically detect faults within a number of analysis tools in the tree domains under study: FMs, CUDF documents and CNF formulas. The results are reported in the following sections.

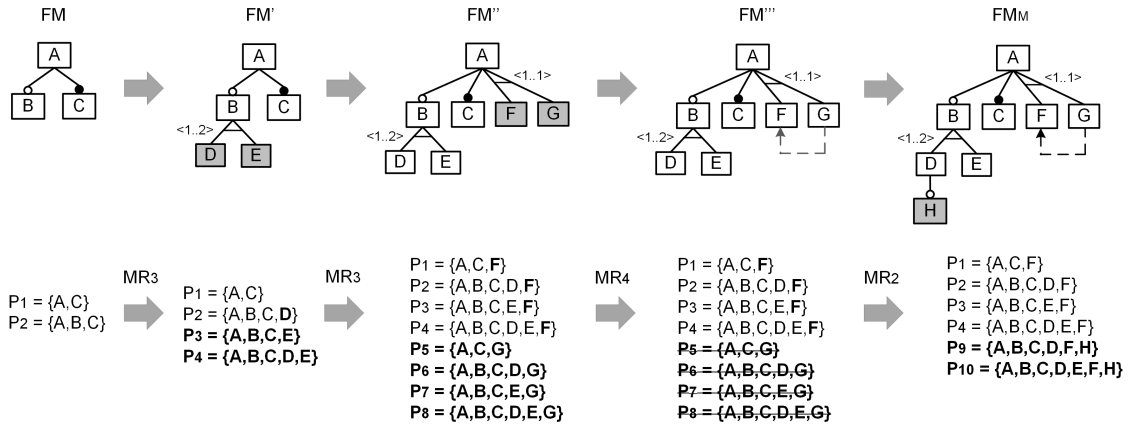

Figure 17: Random generation of a feature model and its set of products using metamorphic relations

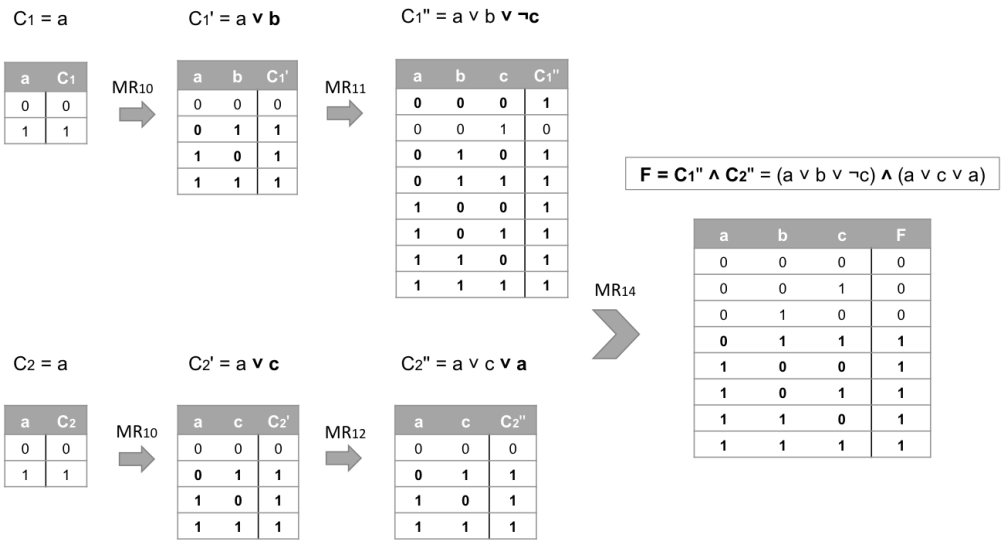

Figure 18: Random generation of a CNF formula and its set of solutions using metamorphic relations

The experiments were performed by two teams in different execution environments for compatibility with the tools under test. For each solver, the specific execution setting is presented in Table 1. The execution environments used were [1] Linux CentOS 6.3 in an Intel Xeon X5560@2.8GHz with 8GB RAM, [2] Windows 8 on a laptop equipped with an Intel Core i5-3317U@1,70Ghz with 6GB RAM, [3] Linux CentOS 6,0 in an Intel Xeon X5550 @ 2,66GHz with 32GB RAM and [4] Linux Debian 7,2 in an Intel Xeon E5 @ 3GHZ with 16GB RAM. Also, the test data for all reasoners were generated in the execution setting [1].

5.1 Detecting faults in FM reasoners

As a part of our work, we developed a test data generator for the analysis of FMs based on the metamorphic relations presented in Section 3.1. The tool generates FMs of a predefined size plus the exact set of configurations that they represent following the procedure presented in Section 4. This test data generator is stable and available as a part of the BeTTy framework [36]. In this experiment, we evaluated the fault detection capability of our metamorphic test data generator by testing the latest release of three FM reasoners in which twelve faults were found.

Experimental setup. We evaluated the effectiveness of our test data generator in detecting faults in three FM reasoners:

Solver	Execution environment
FaMa 1.1.2	[1]
FLAME 1.0	[2]
Splar 05/04/2013	[1]
p2cudf 1.14	[1]
aspcudf 1.7	[3]
cudf-check 0.6.2-1	[4]
sat4j 2.3.1	[1]
Lingeling ala-b02	[3]
Minitsat 2.2	[3]
Clasp 2.1.3	[3]
Picosat 535	[3]
RSAT 2.0	[3]
March_ks 2007	[3]
March_rw 2011	[3]
Kcnfs 1.2	[3]

Table 1: Execution environments used for the experiments

FaMa Framework 1.1.2, SPLAR[3] and FLAME 1.0. FaMa [12] and SPLAR [13, 14] are two open source Java tools for the automated analysis of FMs. FLAME is a prolog-based reasoner developed by some of the authors as a reference implementation to validate a formal specification for the analysis of FMs [31]. FaMa was tested with its default configuration. SPLAR is actually composed of two reasoners using SAT-based and BDD-based analysis, we tested both of them. Tests with FLAME were performed as part of a previous contribution and reproduced for this evaluation [31]. In total, we tested 19 operations in FaMa, 18 en FLAME and 9 in SPLAR. Table 2 provides a detailed description of the operations tested on each reasoner. For each reasoner, the operations passed, failed and not available are specified. The operations marked with an asterisk '(*)' are not in the FaMa 1.1.2 release. They were implemented by FaMa developers under request for this work. The name and formal semantics of the analysis operations mentioned in this paper are based on the work presented in [31].

Operation	FaMa 1.1.2	SPLAR (SAT)	SPLAR (BDD)	FLAME 1.0
Void	Pass	N/A	Fail	Pass
Valid product	Pass	Fail	N/A	Fail
Valid configuration	Fail	Fail	N/A	Pass
Products	Pass	Fail	N/A	Pass
#Products	Pass	Fail	Fail	Pass
Core features	Fail	Fail	Fail	Pass
Unique features	Pass (*)	N/A	N/A	Pass
Variant features	Fail	Fail	Fail	Fail
Dead features	Pass	Fail	Fail	Pass
False optional	Pass	N/A	N/A	N/A
Atomic sets	Fail	N/A	N/A	Pass
Filter	Pass	Fail	N/A	Pass
Commonality	Pass	Fail	N/A	Fail
Variability	Pass	N/A	N/A	Pass
Refactoring	Pass (*)	N/A	N/A	Fail
Generalization	Pass (*)	N/A	N/A	Pass
Specialization	Pass (*)	N/A	N/A	Pass
Arbitrary edit	Pass (*)	N/A	N/A	Pass
Homogeneity	Pass (*)	N/A	N/A	Fail

Table 2: Analysis operations tested in the FM reasoners

[3] SPLAR does not use a version naming system. We tested the tool as it was in April 2013.

The evaluation was performed in two steps. First, we used our metamorphic test data generator to generate 1,000 random FMs and their corresponding set of products. Table 3 summarizes the FM test data parameters used to generate the FMs. The size of the models was between 10 and 20 features and 0% and 20% of cross-tree-constraints (with respect to the number of features). Cardinalities were restricted to $\langle 1..1 \rangle$ and $\langle 1..n \rangle$ (being n the number of subfeatures) for compatibility with the tools under test. The maximum branching factor considered was 10. The generated models represented between 0 and 5,800 products. Then, we proceeded with test execution. For each test case, a FM and its corresponding set of products were loaded, the expected output derived from the set of products and the test run. We ran 1,000 test cases for each analysis operation and reasoner using this procedure. In order to test FLAME test cases were written in an intermediate text file ready to be processed by prolog. In the cases of operations receiving additional inputs apart from the FM those inputs were selected using a basic partition equivalence strategy making sure that the most significant values were tested. We may remark that some of the analysis operations receive two input FMs and return an output indicating how they are related, e.g. refactoring. For those specific operations, an extra suite was generated composed of 1,000 pairs of models and their corresponding set of configurations. The generation of the test cases took less than one minute. The total execution time was 55 minutes, with an average time of 51 seconds per operation under test.

Parameter	Value
Min features	10
Max features	20
Min % CTC	0
Max % CTC	20
Max branching factor	10
Prob mandatory	Random
Prob. optional	Random
Prob or (<1..n>)	Random
Alternative (<1..1>)	Random

Table 3: FM test data parameters

Analysis of results. Table 4 presents the faults detected in the three FM reasoners. For each fault, an identifier, the operations revealing it, a description of the failure and the number of failed tests (out of 1,000) are presented. As illustrated, we detected 4 faults in FaMa, 5 faults in FLAME and 3 faults in SPLAR. In total, we detected 12 faults in 11 different analysis operations. Faults in FaMa and FLAME affected to single operations. In SPLAR, however, failures were identically reproduced in several operations. Due to space limitations, we indicate the number of operations revealing the fault in SPLAR, not their names.

Faults F1, F4 and F7 were revealed when testing the operations with inconsistent models, i.e. a model that represents no products. In FaMa and FLAME, for instance, we found that all features were marked as variants (i.e. selectable) when the model is inconsistent which is a contradiction. Fault F2 revealed a mismatch between the informal definition of the atomic sets operation given in [17] and the formal semantics described in [31]. Fault F3 made some non-valid feature combinations to be wrongly recognized as a valid product. Faults F5 and F6 raised zero division exceptions. Faults F8 and F9 made the order of features in products matter, e.g. [A,B,C] and [A,C,B] were erroneously considered as different products. Fault F10 raised an exception (*org.sat4j.specs.ContradictionException*) when dealing with either inconsistent model or invalid products. The fault was revealed in the initialization of the SPLAR SAT reasoner and therefore affected all operations. Fault F12 made the SPLAT BDD reasoner to fail when processing group cardinalities of the form $\langle 1..n \rangle$. Instead, only group cardinalities of the form $\langle 1..* \rangle$ were supported with identical meaning. Faults F10 and F12 were patched by the authors for further testing of the SPLAR reasoner. Finally, fault F11 was revealed in five operation when receiving exactly the same input inconsistent FMs. We found that several consecutive call to these operations with the same models produced different outputs, i.e. the analysis operations were not idempotent as expected.

The number of failed tests gives an idea of how difficult was to detect each fault. Faults F2, F4, F7 and F12, for instance, were easily detected by a large number of test cases, between 208 and 790 test cases (out of 1,000). Faults F8 and F11, however, were detected by 10 and 5 test cases respectively which shows that some faults are extremely hard to detect. Finally, fault F10 was revealed by a different number of test cases on each operation ranging from 21 test cases (fairly hard to detect) to 759 test cases (very simple to uncover). This supports the need for automated testing mechanisms able to exercise programs with multiple input values and input combinations.

5.2 Detecting faults in CUDF reasoners

For this experiment, we developed a test data generator for the analysis of CUDF documents based on the metamorphic relations defined in Section 3.2. The tool generates CUDF documents of a predefined size plus the exact set of valid

Fault	Operation	Description	Failures
FaMa 1.1.2			
F1	Core features	Wrong output	21
F2	Atomic sets	Wrong output	208
F3	Valid conf.	Wrong output	153
F4	Variant features	Wrong output	219
FLAME			
F5	Homogeneity	Exception	124
F6	Commonality	Exception	37
F7	Variant	Wrong output	273
F8	Refactoring	Wrong output	10
F9	Valid product	Wrong output	121
SPLAR (SAT)			
F10	8 operations	Exception	21-759
F11	5 operations	Wrong output	5
SPLAR (BDD)			
F12	6 operations	Exception	790

Table 4: Faults detected in FM reasoners

configuration represented by the document. In this experiment, we evaluated the ability of the test data generator to detect faults in several CUDF reasoners in which two faults were found.

Experimental setup. We evaluated the effectiveness of our test data generator in detecting faults in three CUDF reasoners: p2cudf 1.14, aspcudf 1.7 and cudf-check 0.6.2-1. p2cudf [25, 37] is a Java tool that reuses the Eclipse dependency management technology (p2) to solve upgradeability problems in CUDF, it internally relies on the Pseudo-Boolean solver Sat4j[38]. Aspcudf [39, 33] uses several C++ tools for Answer Set Programming (ASP), a declarative language. Cudf-check is a command line CUDF reasoner provided by as a part of the Debian cudf-tools package [40]. This tool is mainly used to check the validity CUDF documents and their configurations, i.e. it does not support optimization. In this experiment, we tested two different analysis operations. In the cudf-checker tool, we tested the operation that checks whether a given configuration is valid with respect to a given CUDF document and a given request. In p2cudf and aspcudf, we tested the upgradeability problem using the paranoid optimization criterion [25, 32]. As shown in Section 4, this criterion searches for a configuration that fulfils the user request and minimize the number of changes in the system, i.e. number of installed and removed packages. We selected this optimization operation because it is used in the annual Mancoosi competition and it is supported by most CUDF reasoners.

The evaluation was performed in two steps. First, we used our metamorphic test data generator to generate 1,000 random CUDF documents (with no request) and their corresponding set of configurations. We parametrically controlled the generation assuring that the documents had a fair proportion of all types of properties. Table 5 specifies the parameters and values used for the generation. The generated documents had between 5 and 20 packages and 50% and 120% of constraints, i.e. depends and conflicts. Also, version constraints (e.g. A >= 2) were added with certain probability. Each document represented up to 168,000 different configurations. Once a CUDF document and its configurations were generated, the packages of a random configuration were marked as installed (installed: true) to simulate the current status of the system. Also, a random request was added to each document making no changes in the set of configurations. The request included a list of packages to be installed and a list of packages to be removed. The number of packages in the request was proportional to the number of packages of the document ranging from 1 to 9. Then, we proceeded with test execution. For each test case, a CUDF document and its corresponding set of configuration were loaded, the expected output calculated as described in Section 4 and the test run. We ran 1,000 test cases for each analysis operation and reasoner using this procedure.

Analysis of results. The results revealed 2 faults in the p2cudf reasoner, shown in Table 6. For each fault, an identifier, the operation revealing it, a description of the failure and the number of failed tests (out of 1,000) are presented. The two faults detected, F13 and F14, were uncovered when processing non-equal version constraints in depends disjunctions, e.g. depends: A | B ! = 2. Fault F13 raised an unexpected exception (*org.eclipse.equinox.p2.cudf.metadata.ORRequirement*). The fault was caused by a Java type safety issue in arrays which raised the *ArrayStoreException*. We patched this bug for further testing of the tool. Once fixed, F14 arose due to a wrong handling of the non-equal operator within a disjunction during the encoding step in p2cudf, which makes the tool returns a wrong output. We may mention that this is not a trivial bug because it is caused by a lack of support for nested disjunctions in p2cudf, which occurs scarcely in practice (never in the Mancoosi competition benchmarks). It is noteworthy that fault F14 was detected by only 4 out of our 1,000 test cases, which show the difficulty to reveal certain faults. Again, this motivates the need for automated approaches, as our, able to generate a variety of different inputs that lead to the execution of different paths in the tool under test.

Parameter	Value
Min num packages	5
Max num packages	20
Min % constraints	50
Max % constraints	120
Max dependencies in disjunction	5
Max version	5
Probability new package	0.6
Probability dependency (conjunction)	0.2
Probability dependency (disjunction)	0.1
Probability conflict	0.1
Probability version constraint	0.05

Table 5: CUDF test data parameters

Fault	Operation	Description	Failures
F13	Paranoid	Exception	43
F14	Paranoid	Wrong output	4

Table 6: Faults detected in the CUDF reasoner p2cudf

5.3 Detecting faults in SAT reasoners

For this experiment, we developed a test data generator for the analysis of Boolean formulas based on the metamorphic relations defined in Section 3.3. The tool generates boolean formulas in CNF form plus the exact set of solutions of the formula. For its evaluation, we automatically tested nine SAT solvers in which one bug was revealed.

Experimental setup. We automatically tested nine SAT reasoners written in different languages. The binaries of unversioned reasoners were taken from the SAT competition in which they participated, indicated in parenthesis, namely: Sat4j 2.3.1 [38], Lingeling ala-b02 [41], Minisat 2.2 [42], Clasp 2.1.3 [43], Picosat 535 [44], Rsat 2.0 [45], March_ks (2007) [46], March_rw (2011) [46] and Kcnfs 1.2 [47]. In a related work [48], Brummayer et al. automatically detected faults in the exact same versions of the reasoners Picosat, RSAT and March_ks. We included these three reasoners in our experiments to compare our results with theirs. For each input CNF formula, we enumerated the solutions provided by each reasoner checking that the set of solutions was the expected one. Most of the reasoners do not support enumeration of solutions, they just returns the first solution found if the formula is satisfiable (SAT) or none if it is unsatisfiable (UNSAT). To enable enumeration, we added a new constraint in the input formula after each solution found in order to prevent the same solution to be found in successive calls to the solver, until no more solutions were found.

For the evaluation, we used the same number of test cases as in [48] to make our results comparable. In particular, we first used our metamorphic test data generator to generate 10,000 random CNF formulas in DIMACS format and their corresponding set of solutions. The generated formulas had between 4 and 12 variables and between 5 and 25 clauses. Each clause had between 2 and 5 variables. Most of the generated formulas (94.3%) were satisfiable representing up to 3,480 different solutions. Most reasoners assume that input clauses have no duplicated variables ($a \vee a$) or tautologies ($a \vee \neg a$) since this is not allowed in the input format of the SAT competition, in which most of them participate. Thus, we disabled metamorphic relations MR_{12} and MR_{13} to make the test inputs compatible with most of the tools under test. After the generation we proceeded with test execution. For each test case, a CNF formula and its corresponding set of solutions were loaded and the test run. On each test, we checked that the solutions returned by the reasoner matched the solutions generated by our test data generator. We ran 10,000 test cases on each SAT reasoner using this procedure. Since each test case exercises the SAT solver once per solution found and a last time to check that no more solution exists, each reasoner was expected to answer SAT 1,817,142 times (i.e. total number of solutions in the suite) and UNSAT 10,000 times in total. The generation of the test data took 3 hours. The execution time ranged between 9 minutes in Sat4j and almost 10 days in Kcnfs (due to timeouts, see comments below). Note that Sat4j does support solution enumeration natively, so it did not require to read each time a new problem with a new blocking clause. Thus, no filesystem I/O operations incurred in that case and, more importantly, the solver can take advantage of an incremental setting. In contrast, solvers such as Minisat or Clasp had to run during 6 hours due mainly to the creation of intermediate CNF input files.

Analysis of results. Table 7 summarizes the faults detected in the SAT reasoners. Note that no faults were detected in the reasoners Sat4j, Minisat, Lingeling, and Clasp. This was expected since these are widely used SAT reasoners highly tested and validated by their community of users. However, we detected various defects on more prototypical reasoners, such as Kcnfs or March reasoners. In particular, we automatically detected 3 faults in March_ks, 1 fault in March_rw and 1 fault in Kcnfs, 5 faults in total.

Fault	Operation	Description	Failures
March_ks			
F15	Satisfiability	UNSAT instead of SAT	1
F16	Satisfiability	SAT instead of UNSAT	38
F17	Satisfiability	Cannot decide	21
March_rw			
F18	Satisfiability	Cannot decide	6
Kcnfs			
F19	Satisfiability	Timeout exceeded	952

Table 7: Faults detected in SAT reasoners

Two of the faults made March_ks to answer incorrectly UNSAT instead of SAT (F15) or SAT instead of UNSAT (F16). Faults F17 and F18 made the reasoners unable to decide the satisfiability of the formula, i.e. they return UNKNOWN instead of SAT or UNSAT. According to March developers, this was due to a "*problem with the solution reconstruction after the removal of XOR constraints*". Regarding fault F19, Kcnfs seemed to enter into an infinite loop after iterating over a few solutions. To complete the tests, we used a timeout of 15 minutes before considering the program faulty, it was reached 952 times.

In [48], Brummayer et al. compared the effectiveness of three test data generators for SAT: 3SATGen, CNFuzz and FuzzSAT. Each generator was used to generate 10,000 test cases, 30,000 in total. When comparing our results to theirs, the findings are heterogeneous. On the one hand, they found 86 errors in Rsat (i.e. unexpected termination without providing a result) and 2 failures in Picosat producing a wrong answer. We could not reproduce any of these defects in our work. On the other hand, we detected 39 failures producing a wrong answer in March_ks while they revealed only 4. This is mainly due to the enumeration of all solutions in our approach, i.e. most faults would not have been detected using a single call to the SAT solver as in [48]. In fact, only 11 out of 39 failures in March_ks were revealed with the first call to the SAT solver. The remaining 28 failures were detected while iterating over all the solutions of the input formula. This suggests that our metamorphic test data generator could be complementary to the existing testing tools for SAT helping them to reveal more faults.

As previously mentioned, we disabled metamorphic relations MR_{12} and MR_{13} to avoid generating clauses with duplicated variables or tautologies, since these are not supported by most reasoners. As a sanity check, we enabled these relations and generated and executed another 10,000 test cases. We found that some reasoners, as Sat4j, lingeling, Minisat manage tautologies and duplicate variables effectively while other such as march or kcnfs crashes or simply return a wrong answer. This suggests that our test data generator would also be effective in detecting faults related to a wrong handling of duplicated variables and tautologies in production reasoners.

The number of failures revealed by faults F16-F18 was significantly low ranging from 1 to 38, out of 10,000. This again demonstrates how hard is to detect certain bugs and motivate the need for automated testing techniques. This also suggests that using a larger test suite could have revealed more bugs.

6 Threats to validity

The factors that could have influenced our results are summarized in the following internal and external validity threats.

External validity. This can be mainly divided into limitations of the approach and generalizability of the conclusions. Regarding the limitations, the number of configurations generated by our test data generator increases exponentially with the size of the variability models. As a result, our approach is unable to generate huge variability models hard to analyse. We remark, however, that computationally-hard inputs are not appealing from a functional testing point of view, e.g. executing a test case per hour is unlikely to provide successful results. Instead, as in our work, test data generators should be able to generate multiple inputs of various complexity, most of them easy to process, in order to exercise as many execution paths as possible in the tools under test. This is supported by our previous works with FMs and mutation in which we found that most faults were detected by small inputs [23, 24]. Having said this, we emphasize that our test data generators can efficiently generate variability models representing hundreds of thousands of configurations which goes well beyond the scope of manual testing.

Regarding the generalization of the conclusions, we evaluated our approach with three different variability languages which could seem not to be sufficient to generalize the conclusions of our study. We remark, however, that these languages are used in completely different domains and have particularities that make them sufficiently heterogeneous such as hierarchical constraints in FMs, version and installation constraints in CUDF documents or negated variables in Boolean formulas. Beside these particularities, the three languages have variability constraints with similar semantics, e.g. excludes in FMs $\approx$ conflicts in CUDF. These constraints are very common in variability modelling which suggests that our approach could be easily applicable to other variability languages such as orthogonal variability models [6] or decision models [7].

We made same assumptions in CUDF to keep our metamorphic relations simple, namely: i) only one version of the same package can be included in a CUDF document, ii) CUDF documents are self-contained, i.e. all constraints reference packages defined in the same document, and iii) the CUDF property "*provides*", used to describe the abstract features provided by packages, was omitted. This means that we worked with a subset of CUDF which may affect the generalizability of our conclusions. We remark, however, that this subset includes most of the features of the language and was sufficient to detect several bugs in the CUDF reasoners under test. In fact, this reflects a positive point in favour of our approach since it can be used to test, at least partially, variability analysis tools even if some features of the input languages are omitted.

Finally, since FMs and CUDF documents can be translated to (pseudo) Boolean formulas, it could be argued that working directly with Boolean formulas is a simpler and more generic approach. We did not follow this direction for two reasons. First, the translation from high-level variability models to Boolean formulas should be bidirectional which is a complex and language-specific task [49]. Second, and more importantly, translating models to formulas, forward and backward, would make test data generators very complex and probably more error-prone than the analyses under test.

Internal validity. This refers to whether there is sufficient evidence to support the conclusions. In order to evaluate our approach, we automatically tested 22 analysis operations in 15 different reasoners written in a variety of programming languages. Among the reasoners, four were developed by some of the authors meanwhile eleven of them were developed by external developers. This clearly shows the black-box nature of work which enables testing analysis tools with no knowledge about their internal details. As a result of the tests, we detected 19 total faults in the three domains under study: analysis of FMs, CUDF document and CNF formulas. Most faults were confirmed by the respective tool's developers, the related literature or fix reports. In a few cases (F11, F14-F19), we could confirm the failures but not the faults causing them. Hence, there is chance that faults F15-F17, detected in March_ks, are actually the same fault revealing a different behaviour. Analogously, since some isolated defects are still being investigated by their respective developers (e.g. F11, F14, F18), it could be the case that certain failures are caused by the interaction of more than one fault. Therefore, we must admit a small margin of error (above or below) in the number of reported faults.

7 Related work

Brummayer et al. [48] presented a fuzz testing approach for the automated detection of faults in SAT reasoners. Fuzzy testing is a black-box technique in which the tools under test are fed with random and syntactically valid inputs in order to find faults. To check the correctness of the outputs, the authors used redundant testing, that is, they compared the results of several reasoners and trusted on the majority. In their paper, the authors mentioned "*If all solvers agreed that the current instance is unsatisfiable, we did not further validate the unsatisfiability status as it is highly unlikely that all solvers are wrong.*". Note that SAT solvers can also be equipped to produce UNSAT proofs to be checked by independent external tools [50]. A similar approach for testing ASP reasoners was presented by Brummayer and Järvisalo in [51]. Artho et al. [52] proposed a model-based testing approach to test sequences of methods calls and configurations in SAT reasoners. This approach is tool-dependent since it requires to model the valid sequences of API calls as well as valid configuration options of the SAT reasoner under test. For its evaluation, they introduced artificial faults in the Lingeling SAT reasoner. In contrast to these works, our contribution is generic being applicable to different variability languages and tools regardless of their implementation details, i.e. black-box approach. Also, our work truly overcomes the oracle problem by generating the exact set of solutions of each SAT formula instead of depending on third-party tools using redundant testing.

In [53], some of the authors presented a test suite for the analysis of FMs. The suite was composed of 192 manually-designed test cases intended to test six different analysis operations. The suite was evaluated using mutation testing in the FM reasoner FaMa in which two real bugs were detected. Although partially effective, we found that the manual design of test cases was extremely time consuming and error-prone. This motivated the need for the proposed approach which clearly outperforms the manual suite in terms of automation, generalizability and effectiveness.

Regarding CUDF, the Mancoosi solver competition 2012 provided a solution checker to assess the correctness of the solutions returned by the participant CUDF reasoners [32], i.e. redundant testing. Other related works have been presented in the context of systems with high variability. Vouillon and Di Cosmo [54] proposed a theoretical framework to detect co-installability conflicts in component–based systems, i.e. components that cannot be installed together. Artho et al. [55] proposed a similar approach to detect and prevent conflicts in package–based distributions. Cohen et al. [56] presented a set of combinatorial testing algorithms for sample generation in highly–configurable systems. Most related works propose novel analysis operations on variability models but no specific means to test the own analysis tools themselves. In this sense, our work complements previous approaches laying the foundations for the automated detection of faults in variability analysis tools as those found in open–source package configurators, web configuration tools (e.g. car's features selector) or plugin–based IDEs.

In the context of performance testing, some authors have presented algorithms for the automated generation of computationally-hard SAT problems [57] and FMs [36]. Interestingly, some of the algorithms for SAT can be configured to generate satisfiable or unsatisfiable instances only. This is usually done by starting from a known formula and adding constraints to it assuring at each step that the formula is still (un)satisfiable. This procedure could also be used for the automated detection of functional faults. Our work, however, goes a step further since it enables not only knowing whether the input model is satisfiable or not but also the exact set of configurations of it. This enables testing not only

the satisfiability operation, but any analysis operation that extract information from the set of configurations of the model, 22 in our work.

Regarding metamorphic testing, Kuo et al. [58] presented an approach for the automated detection of faults in decision support systems. In particular, the authors focused in so-called Multi-Criteria Group Decision Making (MCGDM) where decision problems are modelled as a matrix with several dimensions: alternatives, criteria and experts. The authors introduced eleven metamorphic relations in natural language and evaluated their approach using artificial faults in the research tool Decider. This work has certain commonalities with our contribution since variability models could be used as decision models during software configuration. Also, as in our work, Kuo et al. used metamorphic relations to actually construct the output of follow up test cases (i.e. follow up matrices) instead of just checking the output of the tests. However, our contribution is applied to a different domain, analysis of software variability, in which three different variability languages were used to illustrate our approach. Also, we formally defined our metamorphic relations and, more importantly, we evaluated our test data generators with numerous reasoners in which 19 real bugs were detected.

The automated generation of test cases is a hot research topic that involves numerous techniques [59]. Adaptive random testing [60] proposes using random inputs spread across the input domain of the system under test. Combinatorial interaction testing [61] systematically select inputs that may reveal failures caused by the interaction between two or more input values. Model-based testing [62] use models of systems (e.g. finite state machines) to derive test suites using a test criterion based on a test hypothesis that justify the adequateness of the selection. Other techniques such as those based on symbolic execution, mutation testing and most variants of search-based testing work at the code level (i.e. white-box) and are therefore out of the scope of our approach. Most previous work concentrates on the problem of generating good test inputs, but they do not address the equally relevant challenge of assessing the correctness of the outputs produced by the generated inputs, i.e. oracle problem. In contrast, our approach overcomes both problems, automated generation of inputs and expected outputs, providing a fully automated fault detection mechanism. This is especially valuable considering the black-box nature of our approach, which makes it suitable to detect faults in numerous reasoners and analysis operations regardless of their implementation details.

Finally, we may mention that our approach is independent of the heuristic used to generate the input variability models. Thus, our approach could be complementary to most related works on test data generation provided that the construction of the inputs is performed using incremental step–wise transformations and assuring that metamorphic relations hold at each step.

8 Conclusions

In this technical report, we presented a metamorphic testing approach for the automated detection of faults in variability analysis tools. This method enables the generation of non-trivial variability models and their corresponding configurations, from which the expected output of a number of analyses over the model can be derived overcoming the oracle problem. A key benefit of the approach is its applicability to any variability language with common variability constraints where metamorphic relations can be identified. To show the feasibility and generalizability of our work, we automatically tested the implementation of 22 analysis operations in 15 reasoners written in different languages in the domains of feature models, CUDF documents and CNF formulas. In total, we automatically detected 19 real bugs in seven of the tools under test. Most faults were directly acknowledged by the tools' developers from whom we received comments as "*You hammered it right on the nail!*" or "*the bugs found by your tests are non trivial issues*". This supports our conclusions and reinforce the potential of metamorphic testing as an automated testing technique.

Acknowledgments

We appreciate the help of Dr Martin Monperrus whose comments and suggestions helped us to improve the technical report substantially. We would also like to thank Dr. Marcílio Mendonça, Dr. Marijn J. H. Heule and José A. Galindo for confirming the bugs found in their respective tools.

References

1. Svahnberg M, van Gurp L, Bosch J. A taxonomy of variability realization techniques: Research Articles. Software Practice and Experience. 2005;35(8):705–754. Available from: `http://dx.doi.org/10.1002/spe.v35:8`.

2. Debian 7.0 Wheezy released; 2013. Accessed November 2013.

3. Eclipse Marketplace `http://marketplace.eclipse.org/`;. Accessed November 2013.

4. García-Galán J, Rana OF, Trinidad P, Ruiz-Cortés A. Migrating to the Cloud: a Software Product Line based analysis. In: 3rd International Conference on Cloud Computing and Services Science (CLOSER'13); 2013. .

5. Kang K, Cohen S, Hess J, Novak W, Peterson S. Feature–Oriented Domain Analysis (FODA) Feasibility Study. SEI; 1990. CMU/SEI-90-TR-21.

6. Pohl K, Bückle G, , van der Linden F. Software Product Line Engineering: Foundations, Principles, and Techniques. Springer–Verlag; 2005.

7. Stoiber R, Glinz M. Supporting Stepwise, Incremental Product Derivation in Product Line Requirements Engineering. In: International Workshop on Variability Modelling of Software-intensive Systems.. vol. 37; 2010. p. 77–84. Available from: http://dblp.uni-trier.de/db/conf/vamos/vamos2010.html#StoiberG10.

8. Berger T, She S, Lotufo R, Wasowski A, Czarnecki K. Variability modeling in the real: a perspective from the operating systems domain. In: International Conference on Automated Software Engineering (ASE'10); 2010. p. 73–82.

9. Berre DL, Rapicault P. Dependency management for the eclipse ecosystem: eclipse p2, metadata and resolution. In: Proceedings of the 1st international Workshop on Open Component Ecosystems. IWOCE '09. New York, NY, USA: ACM; 2009. p. 21–30. Available from: http://doi.acm.org/10.1145/1595800.1595805.

10. Müller C, Resinas M, Ruiz-Cortés A. Automated Analysis of Conflicts in WS–Agreement Documents. IEEE Transactions on Services Computing. 2013;Available from: http://dx.doi.org/10.1109/TSC.2013.9.

11. Jang M. Linux Annoyances for Geeks. O'Reilly; 2006.

12. FaMa Tool Suite. http://www.isa.us.es/fama/; Accessed November 2013.

13. Mendonca M, Branco M, Cowan D. S.P.L.O.T.: Software Product Lines Online Tools. In: Companion to the 24th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). Orlando, Florida, USA: ACM; 2009. p. 761–762.

14. Software Product Lines Automated Reasoning library (SPLAR) http://code.google.com/p/splar/;. Accessed November 2013.

15. Veer B, Dallaway J. The eCos Component Writer's Guide;. Accessed November 2013. ecos.sourceware.org/ecos/docs-latest/cdl-guide/cdl-guide.html.

16. Debian Reference Guide. http://www.debian.org/doc/manuals/debian-reference/;. Accessed November 2013.

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

18. Weyuker EJ. On Testing Non-Testable Programs. The Computer Journal. 1982;25(4):465–470.

19. Chen TY, Cheung SC, Yiu SM. Metamorphic testing: a new approach for generating next test cases. Hong Kong: University of Science and Technology; 1998. HKUST-CS98-01.

20. Chen TY, Feng J, Tse TH. Metamorphic testing of programs on partial differential equations: a case study. In: Proceedings of the 26th International Computer Software and Applications Conference; 2002. p. 327–333.

21. Chen TY, Huang DH, Tse TH, Zhou ZQ. Case studies on the selection of useful relations in metamorphic testing. In: Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC); 2004. p. 569–583.

22. Chan W, Cheung S, Leung K. A metamorphic testing approach for online testing of service-oriented software applications. International Journal of Web Services Research. 2007;4(2):61–81.

23. Segura S, Hierons RM, Benavides D, Ruiz-Cortés A. Automated Test Data Generation on the Analyses of Feature Models: A Metamorphic Testing Approach. In: International Conference on Software Testing, Verification and Validation. Paris, France: IEEE press; 2010. p. 35–44.

24. Segura S, Hierons RM, Benavides D, Ruiz-Cortés A. Automated Metamorphic Testing on the Analyses of Feature Models. Information and Software Technology. 2011;53:245–258.

25. Argelich L, Berree DL, Lynce I, Silva JP, Rapicault P. Solving Linux Upgradeability Problems Using Boolean Optimization. In: Lynce I, Treinen R, editors. Workshop on Logics for Component Configuration. vol. 29 of EPTCS; 2010. p. 11–22.

26. Treinen R, Zacchirol S. Common Upgradeability Description Format (CUDF) 2.0. The Mancoosi project (FP7); 2009. 003.

27. Berre DL, Parrain A. On SAT Technologies for Dependency Management and Beyond. In: First workshop on Analyses of Software Product Lines. vol. 2; 2008. p. 197–200.

28. Mendonca M, Wasowski A, Czarnecki K. SAT–based analysis of feature models is easy. In: Proceedings of the International Sofware Product Line Conference (SPLC); 2009. .

29. Benavides D, Ruiz-Cortés A, Trinidad P. Automated Reasoning on Feature Models. In: 17th International Conference on Advanced Information Systems Engineering (CAiSE). vol. 3520 of Lecture Notes in Computer Sciences. Springer–Verlag; 2005. p. 491–503.

30. Wang HH, Li YF, Sun J, Zhang H, Pan J. Verifying Feature Models using OWL. Journal of Web Semantics. 2007 June;5:117–129.

31. Durán A, Benavides D, Segura S, Trinidad P, Ruiz-Cortés A. FLAME: FAMA Formal Framework (v 1.0). Seville, Spain; 2012. ISA–12–TR–02.

32. Mancoosi European research project. `http://www.mancoosi.org/`;. Accessed November 2013.

33. Gebser M, Kaminski R, Schaub T. aspcud: A Linux Package Configuration Tool Based on Answer Set Programming. In: Drescher C, Lynce I, Treinen R, editors. Workshop on Logics for Component Configuration. vol. 65 of EPTCS; 2011. p. 12–25.

34. Zhou ZQ, Huang D, Tse T, Yang Z, Huang H, Chen T. Metamorphic testing and its applications. In: Proceedings of the 8th International Symposium on Future Software Technology; 2004. p. 346–351.

35. Galindo JA, Benavides D, Segura S. Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis. In: Proceedings of the 1st International Workshop on Automated Configuration and Tailoring of Applications (ACoTA). Antwerp, Belgium; 2010. .

36. Segura S, Galindo JA, Benavides D, Parejo JA, Ruiz-Cortś A. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In: Eisenecker UW, Apel S, Gnesi S, editors. Sixth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12). Leipzig, Germany: ACM; 2012. p. 63–71.

37. p2cudf `http://wiki.eclipse.org/Equinox/p2/CUDFResolver`;. Accessed November 2013.

38. Berre DL, Parrain A. The Sat4j library, release 2.2. Journal on Satisfiability, Boolean Modeling and Computation. 2010;7:59–64. System description.

39. aspcud. `http://www.cs.uni-potsdam.de/wv/aspcud`;. Accessed November 2013.

40. Cudf-tools Debian package `http://packages.debian.org/wheezy/cudf-tools`;. Accessed November 2013.

41. Lingeling SAT solver. `http://fmv.jku.at/lingeling/`;. Accessed November 2013.

42. Eén N, Sörensson N. An Extensible SAT-solver. In: Giunchiglia E, Tacchella A, editors. SAT. vol. 2919 of Lecture Notes in Computer Science. Springer; 2003. p. 502–518.

43. Gebser M, Kaufmann B, Neumann A, Schaub T. *clasp* : A Conflict-Driven Answer Set Solver. In: Baral C, Brewka G, Schlipf JS, editors. LPNMR. vol. 4483 of Lecture Notes in Computer Science. Springer; 2007. p. 260–265.

44. Biere A. PicoSAT Essentials. JSAT. 2008;4(2-4):75–97.

45. Pipatsrisawat K, Darwiche A. A Lightweight Component Caching Scheme for Satisfiability Solvers. In: Marques-Silva J, Sakallah KA, editors. SAT. vol. 4501 of Lecture Notes in Computer Science. Springer; 2007. p. 294–299.

46. Heule M. SmArT Solving: Tools and Techniques for Satisfiability Solvers. TU Delft; 2008.

47. Dubois O, Dequen G. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In: Nebel B, editor. IJCAI. Morgan Kaufmann; 2001. p. 248–253.

48. Brummayer R, Lonsing F, Biere A. Automated testing and debugging of SAT and QBF solvers. In: Proceedings of the 13th international conference on Theory and Applications of Satisfiability Testing. SAT'10. Berlin, Heidelberg: Springer-Verlag; 2010. p. 44–57. Available from: `http://dx.doi.org/10.1007/978-3-642-14186-7_6`.

49. Czarnecki K, Wasowski A. Feature Diagrams and Logics: There and Back Again. In: 11th International Software Product Line Conference (SPLC). Los Alamitos, CA, USA: IEEE Computer Society; 2007. p. 23–34.

50. Van Gelder A. Producing and verifying extremely large propositional refutations - Have your cake and eat it too. Ann Math Artif Intell. 2012;65(4):329–372.

51. Brummayer R, Järvisalo M. Testing and debugging techniques for answer set solver development. Journal of Theory and Practice of Logic Programming. 2010 Jul;10(4-6):741–758. Available from: `http://dx.doi.org/10.1017/S1471068410000396`.

52. Artho C, Biere A, Seidl M. Model-Based Testing for Verification Back-ends. In: 7th International Conference on Tests & Proofs. Budapest, Hungary: Springer; 2013. .

53. Segura S, Benavides D, Ruiz-Cortés A. Functional testing of feature model analysis tools: a test suite. Software, IET. 2011;5(1):70–82.

54. DiCosmo R, Vouillon J. On software component co-installability. In: 13th European conference on Foundations of Software Engineering. ESEC/FSE '11. New York, NY, USA: ACM; 2011. p. 256–266. Available from: http://doi.acm.org/10.1145/2025113.2025149.

55. Artho C, Suzaki K, DiCosmo R, Treinen R, Zacchiroli S. Why do software packages conflict? In: 9th IEEE Working Conference of Mining Software Repositories; 2012. p. 141–150.

56. Cohen MB, Dwyer MB, Jiangfan S. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. Software Engineering, IEEE Transactions on. 2008;34(5):633–650.

57. Cook SA, Mitchell DG. Finding Hard Instances of the Satisfiability Problem: A Survey. In: Satisfiability Problem: Theory and Applications. vol. 35 of Dimacs Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society; 1997. p. 1–17.

58. Kuo FC, Zhou ZQ, Ma J, Zhang G. Metamorphic testing of decision support systems: a case study. Software, IET. 2010;4(4):294–301.

59. Anand S, Burke E, Chen TY, Clark J, Cohen MB, Grieskamp W, et al. An Orchestrated Survey on Automated Software Test Case Generation. Journal of Systems and Software. 2013 August;86(8):1978–2001.

60. Chen TY, Kuo FC, Merkel RG, Tse TH. Adaptive Random Testing: The ART of test case diversity. Journal of Systems and Software. 2010 Jan;83(1):60–66. Available from: http://dx.doi.org/10.1016/j.jss.2009.02.022.

61. Nie C, Leung H. A survey of combinatorial testing. ACM Computing Surveys. 2011 Feb;43(2):11:1–11:29. Available from: http://doi.acm.org/10.1145/1883612.1883618.

62. Utting M, Pretschner A, Legeard B. A taxonomy of model-based testing approaches. Software Testing Verification and Reliability. 2012 Aug;22(5):297–312. Available from: http://dx.doi.org/10.1002/stvr.456.