View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by idUS. Depósito de Investigación Universidad de Sevilla

Automated Metamorphic Testing of Variability Analysis Tools

Sergio Segura[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*

SUMMARY

Variability determines the capability 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 article, 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, it has been 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 7 out of the 15 tools under test.

KEY WORDS: Metamorphic testing, automated testing, software testing, software variability

1. INTRODUCTION

Modern software applications are increasingly configurable, driven by customer demands and dynamic business conditions. This leads to software systems that need a high degree of *variability*, i.e. the capability to be extended, changed, customized, or configured to be used in a particular context [60]. For example, operating systems such as *Linux* or *eCos* can be configured by installing packages, e.g. *Debian Wheezy*, a well–known Linux distribution, offers more than 37,000 available packages [22]. Modern software ecosystems and browsers, configurable in terms of *plugins* or extensions, are another example of software variability, e.g. the *Eclipse Marketplace* currently provides about 1,650 Eclipse plugins [26]. Recently, cloud applications are becoming increasingly flexible, e.g. the *Amazon Elastic Compute Cloud* (EC2) service offers 1,758 different possible configurations [30].

Software variability is usually documented by *variability models* (VMs), which describe all the possible configurations of a software system in terms of composable units or *variants*, and *constraints* indicating how those variants can be properly 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 VMs such as *feature models* [35], *orthogonal variability models* [51] or *decision models* [59]. At the solution level, variability is modelled using domain–specific languages such as *Kconfig* in Linux [8], *p2* in Eclipse [39] or *WS–Agreement* in web services [47].

Regardless of the abstraction level of VMs, the number of constraints in VMs is potentially huge. For instance, the Linux kernel has 6,320 packages and 86% of them are connected by constraints that restrict their interactions [8], something colloquially known as the "*dependency hell*" in the operating system domain [34]. To manage this complexity, automated support is primordial. The automated analysis of VMs deals with the computer–aided extraction of information from VMs by means of the so–called *analysis operations*. For a given VM, typical analysis operations would allow us to know whether the VM is *consistent*, i.e. whether it represents at least a valid configuration; or whether the VM contains any errors, e.g. contradictory configuration options. Tools supporting the analysis of VMs can be found in most of the software domains where variability management is considered as a relevant problem. Some examples are the *FaMa* framework [28] and the *SPLAR* tool [45, 58] in the context of feature models; the *CDL* [64] and *APT* [21] package configurators in the context of operating systems; or the dependency analysis tool integrated into Eclipse [39].

Variability analysis tools are complex software systems that have to deal with also complex data structures and algorithms, e.g. the FaMa framework has more than 20,000 lines of code. Analysis operations are usually far from trivial and their development is error–prone, increasing development time and reducing the reliability of analysis tools. In this context, the testing of variability analysis tools aim at detecting faults that produce unexpected analysis results. Roughly speaking, a test case in the domain of variability analysis is composed of a VM as the input, and the expected result of the analysis operation under test as the output. As an example, the feature model in Fig. 1 (Section 2.1) represents 10 different product configurations, which is the expected result of the analysis operation *NumberOfProducts* [7].

Current testing methods on variability analysis tools 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 operation is correct or not. However, this is time–consuming, error–prone and, in most cases, infeasible due to the combinatorial complexity of the analysis operations. This is known as the *oracle problem* [66], i.e. the impossibility to determine the correctness of a test output. On the other hand, 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 [13] was proposed as a way to address the oracle problem. The rationale 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 that it can be highly automated. Metamorphic testing has shown to be effective in a number of testing domains, including numerical programs [14], graph theory [15] or service–oriented applications [12].

Problem description. In previous works [55, 56], some of the authors 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 [35]. For the evaluation of our work, we introduced hundreds of artificial faults (i.e. *mutants*) into several subject programs and checked how many 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 languages different from 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.

- *RQ2. Is metamorphic testing effective in detecting real bugs in variability analysis tools?* Despite the mutation testing results obtained in our previous works, the capability of our approach to detect real bugs is still to be assessed. 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 article, 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 VMs (i.e. inputs) and the exact set of valid configurations represented by the models (i.e. expected output). Both, the VMs and their configurations are generated from scratch using step–wise transformations and assuring that certain constraints (i.e. metamorphic relations) hold at each step. Complex VMs representing thousands of configurations can be efficiently generated by applying this process iteratively. Once generated, the configurations of each VM are automatically inspected to derive the expected output of a number of analysis operations performed on the VMs. Our approach is fully automated, highly generic, and 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 capability of our approach to automatically detect faults in three different software variability domains, namely:

- *Feature models*. These hierarchical VMs are used to describe the products of a software product line in terms of features and relations among them [35]. Five metamorphic relations for feature models and their corresponding test data generator are presented on this article. For its evaluation, we automatically tested 19 different analysis operations in 3 feature models reasoners. We detected 12 faults.
- *CUDF documents*. These VMs are textual documents used to describe package–based *Free and Open Source Software* distributions [2, 61]. 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 3 CUDF reasoners. We detected 2 faults.
- *CNF formulas*. Among its applications, Boolean formulas in *Conjunctive Normal Form* (CNF) are extensively used for variability representation and analysis at a low level of abstraction. Many VMs such feature models or decision models can be automatically analysed by translating them into CNF formulas and solving the Boolean *satisfiability* problem (SAT) [7, 59]. Also, SAT technology is used to deal with variability management in software ecosystems such as Eclipse or Linux [37, 39]. Five metamorphic relations for CNF formulas and a test data generator relying on them are presented in this article. For its evaluation, we automatically tested the satisfiability operation in 9 SAT reasoners. We detected 5 faults.

In quantitative terms, this article extends and generalizes our previous works by automating the detection of faults in two new variability domains, CUDF and CNF formulas. Also, we present 10 new metamorphic relations (out of 14) and a thorough empirical evaluation with more analysis operations (from 6 to 22), tools (from 2 to 15) and detected faults (from 4 to 19).

The rest of the article is structured as follows: Section 2 introduces the variability languages used in our empirical evaluation 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 some guidelines for the application of our approach to other variability analysis domains. The limitations of our approach are presented in Section 7. Section 8 presents the threats to validity of our work. The related work is presented and discussed in Section 9. Finally, we summarize our conclusions in Section 10.

2. PRELIMINARIES

Variability languages are used to develop VMs describing all the possible configurations of a family of software systems in terms of variants and constraints restricting how those variants 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) [35]. An 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.

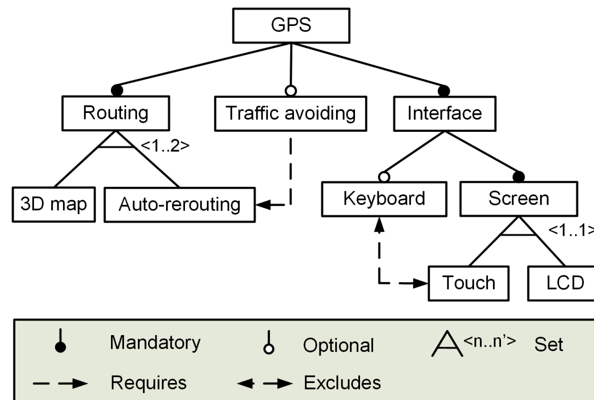

Figure 1. A sample feature model

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, whereas those provided with *Touch screen* exclude the support for a *Keyboard*.

The automated analysis of FMs deals with the computer–aided extraction of information from FMs. Catalogues with up to 30 analysis operations on FMs have been published [7]. Typical analysis operations allow us to know whether an FM is consistent (i.e. it represents at least one product), what is the number of products represented by an FM, or whether an FM contains any

errors. Common techniques to perform these operations are those based on propositional logic [46], constraint programming [6] or description logic [65]. Also, these analysis capabilities can be found in a number of commercial and open source tools including the *FaMa* framework [28], the *FLAME* framework [25] and SPLAR [45, 58].

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 [2, 61]. This format is one of the outcomes of the *Mancoosi* European research project [43], intended to build better and generic tools for package–based system administration. CUDF combines features of the Red Had package manager and the Debian packaging systems, and also allows to encode other formats such as metadata of Eclipse plugins [39]. A key benefit of CUDF is that it allows to describe variability independently of the distribution and the package manager used. Also, the syntax and semantics of CUDF documents are well documented, something that facilitates the development of independent analysis tools.

```
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

Fig. 2 depicts a sample CUDF document. As illustrated, it is a text file composed by several paragraphs, so–called *stanzas*, separated by empty lines. Each stanza is composed of set of properties in the form of *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 by '|'). 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 version 6 or greater of sun-java-jdk.
- **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 any version of 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.

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* [2]. Given a CUDF document, this problem consists in finding a valid configuration, i.e. a set of packages that fulfils 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. the 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 *Mancoosi International Solver Competition* (MISC) arranged by the Mancoosi project. In this 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 [31] and Pseudo–Boolean optimization [2].

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 languages such as feature models or decision models can be automatically analysed by translating them into Boolean formulas and solving the corresponding SAT problem [7, 59]. SAT technology is also used to deal with dependency management in software ecosystems such as Eclipse or Linux [37, 39].

A SAT solver is a software package that takes as input a CNF formula and determines if the formula is satisfiable. The *Conjunctive Normal Form* (CNF) is a standard form to represent propositional formulas where only three connectives are allowed: $\neg$, $\wedge$, and $\vee$. CNF formulas consists of the conjunction of a number of *clauses*; 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, $(a \vee \neg b)$, and $(\neg a \vee b \vee c)$, and three variables, 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[†].

2.4. Metamorphic testing

In software testing, an *oracle* is a procedure by which testers can decide whether the output of a program is correct or not [66]. In some situations, the oracle is not available or it is too difficult to apply. For example, consider testing the results of complicated numerical computations such as the 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. Situations like these are referred to as the *oracle problem* in the testing literature [70].

Metamorphic testing [13] 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*, i.e. known relations among two or more input data and their expected outputs. For instance, consider 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 that the program is faulty.

It has been shown that a small number of diverse metamorphic relations has a similar fault–detection capability to a test oracle, and could therefore help to alleviate the oracle problem [41]. The effectiveness of metamorphic relations has been studied in several guidelines for the selection of "good" metamorphic relations [15, 44]. Metamorphic testing has been successfully applied to a number of testing domains including numerical programs [14], graph theory [15] or service–oriented applications [12].

3. METAMORPHIC RELATIONS ON VARIABILITY MODELS

In this section, a set of metamorphic relations between VMs expressed in 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.

3.1. Metamorphic relations on feature models

In terms of variability management, in FMs variants are represented as features, and valid configurations are those feature combinations (i.e. *products*) satisfying all the constraints expressed in the FM. According to this, 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 B. 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 B appears.

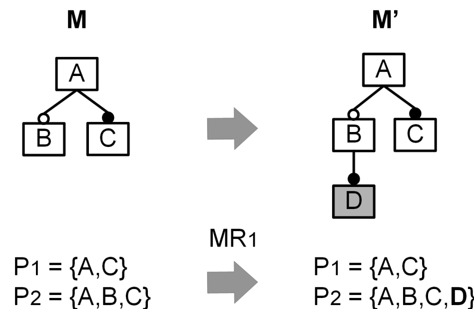

Figure 3. Neighbour models after mandatory feature is added

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 an FM, and $\#$ the cardinality function on sets. Then, MR$_1$ can be

defined as follows:

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

MR$_2$: Optional. When an optional feature is added to an 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. 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:

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

MR$_3$: Set relationship. When a new set relationship with a $\langle n,m\rangle$ cardinality is added to an FM (see Figure 4), the derived set of products is formed by all the original products not containing the parent feature of the set relationship (P_1 in Figure 4), and the new products created by adding all the possible combinations of size $n..m$ of the child features ({D}, {E}, and {D, E} for the FM in Figure 4) to all the products including the parent feature (P_2, P_3, and P_4 in Figure 4).

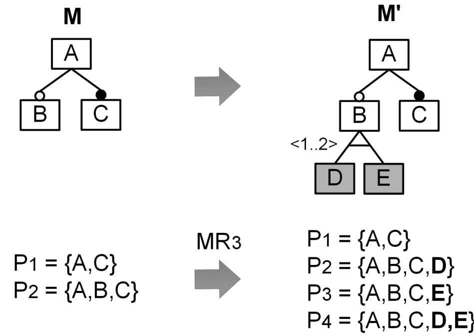

Figure 4. Neighbour feature models after adding a set relationship

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 $\wp^m_n F_s = \{S \in \wp F_s \mid n \le \#S \le m\}$ the set of all possible subsets of F_s with cardinality in the $\langle n,m\rangle$ interval[‡]. Then, assuming that $1 \le n \le m \le \#F_s$, MR$_3$ can be defined as follows:

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

MR$_4$: Requires. When a new f_1 **requires** f_2 constraint is added to an FM, the derived set of products is the original set except those products containing f_1 but not f_2. Formally, MR$_4$ can be defined as follows using the product selection function Π_σ:

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

[‡]$\wp S$ denotes the *powerset* of the set S, containing all possible subsets of S.

MR$_5$: Excludes. When a new f_1 **excludes** f_2 constraint is added to an FM, the derived set of products is the original set except those products containing both f_1 and f_2. 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.

Formally, package dependencies and conflicts 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, a dependency such as $(\text{arduino}, 2, \text{JDK}, 6, \geq)$ indicates that version 2 of the arduino package depends on the JDK package version 6 or higher. In this context, we can also define a *non–constraining* package (p_{nc}, v_{nc}) as a package whose identifier does not appear as dependee in any *constraint*, i.e. dependency or conflict, in a CUDF document. Formally, (p_{nc}, v_{nc}) is non–constraining in the context of a given CUDF document iff:

$$\forall (p, v, q, k, \theta) \in \text{CUDF document} \bullet p_{nc} \neq q$$

Complementarily, a *constraining* package (p_c, v_c) is a package whose identifier *does* appear as dependee in some constraint, i.e.

$$\exists (p, v, q, k, \theta) \in \text{CUDF document} \bullet p_{nc} = q$$

Considering these definitions, it is possible to define the following metamorphic relations between the valid configurations of neighbour CUDF documents.

MR$_6$: New package. When a new non–constraining package is added to a CUDF document, the derived set of valid configurations is formed by the original set, a configuration containing the new non–constraining package only, and all the original configurations with the new non–constraining package added (see Figure 5).

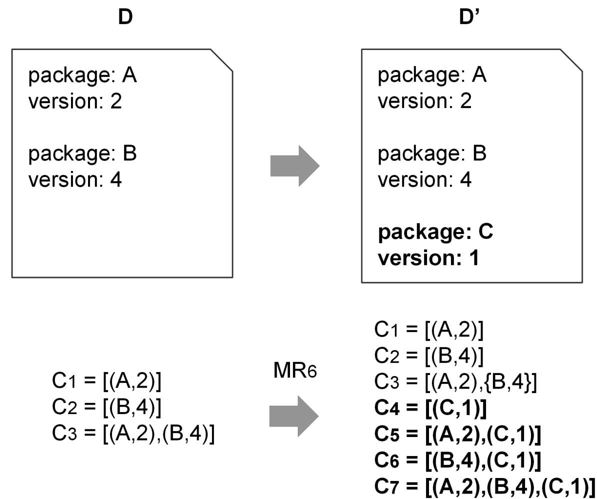

Figure 5. Neighbour CUDF documents after a new non–constraining package is added

Formally, let D' be the CUDF document created by adding a non–constraining package (p_{nc}, v_{nc}) 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:

$$\begin{aligned} \#\Psi(D') &= 2 \cdot \#\Psi(D) + 1 \land \\ \Psi(D) &\subseteq \Psi(D') \land \\ \{ (p_{nc}, v_{nc}) \} &\in \Psi(D') \land \\ \forall c \in \Psi(D) \bullet\ & c \cup \{ (p_{nc}, v_{nc}) \} \in \Psi(D') \end{aligned} \tag{MR$_6$}$$

Notice that constraining packages are excluded from this rule since they would not allow to derive the set of configurations of the new CUDF documents from the previous set[§]. Nevertheless, this exclusion does not affect the diversity of the CUDF documents that can be generated, it only affects the order in which the metamorphic relations should be applied (see Section 4 for details).

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 in which (p, v) does not appear, together with all the original configurations in which (p, v) does appear and at least one of the added disjunctive dependencies is satisfied.

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:

$$\Psi(D') = \{ c \in \Psi(D) \mid (p, v) \in c \Rightarrow \exists\, \delta \in \Delta \bullet \psi(\, c, \delta\,) \} \tag{MR$_7$}$$

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 in which (p, v) does not appear, together with all the original configurations in which (p, v) does appear and the added conjunctive dependency is satisfied. Formally, let δ be the conjunctive dependency added to the (p, v) package in a CUDF document. Then MR_8 can be defined as follows:

$$\Psi(D') = \{ c \in \Psi(D) \mid (p, v) \in c \Rightarrow \psi(\, c, \delta\,) \} \tag{MR$_8$}$$

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 in which (p, v) does not appear, together with all the original configurations in which (p, v) does appear but are not affected by the new conflict. 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:

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

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 neighbour Boolean formulas in CNF 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 6 for an example.

[§]Some configurations, previously discarded for not satisfying a dependency with the new package, would become valid but, since they were not present in the original set, they could not be derived.

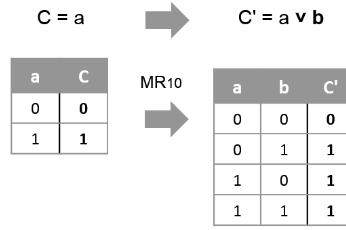

Figure 6. Neighbour CNF formulas after adding a disjunction with a new variable

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*. Formally, MR_{11} can be defined as follows:

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

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. 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. 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:

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

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 9 in Section 4 for an example). Formally, if C_1 and C_2 are the two CNF clauses to be conjuncted, then MR_{14} can be defined as follows:

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

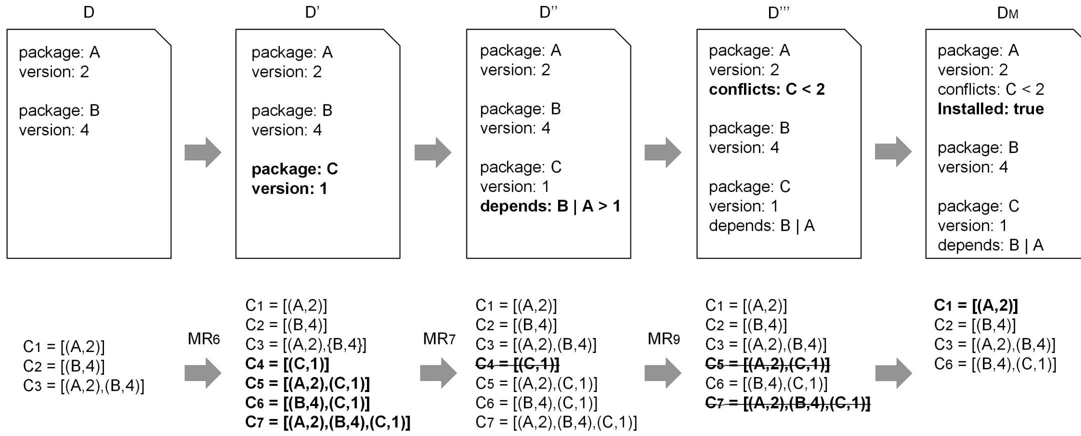

Figure 7. Random generation of a CUDF document and its set of configurations using metamorphic relations

4. AUTOMATED TEST DATA GENERATION

The semantics of a VM is defined by the set of valid configurations that it represents, and most analysis operations on VMs can be performed by inspecting this set adequately. Based on this idea, the two–step process proposed for the automatic generation of test data is presented in this section.

4.1. Variability model generation

The first step of the test data generation is using metamorphic relations, together with model transformations, in order to generate VMs and their sets of valid configurations. Notice that this is a singular application of metamorphic testing, i.e. instead of using metamorphic relations to check the output of different computations, we use them to actually compute the expected output of follow–up test cases. Fig. 7 illustrates an example of our approach.

The process starts with an input VM whose set of valid configurations is known, i.e. a *seed*. This seed can be randomly generated from scratch (as in our approach) or obtained from an existing test case [56]. 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 valid configurations according to the metamorphic relations. In the example, document D' is generated by adding a new package C to document D. The set of configurations of D' is then easily calculated by applying the metamorphic relation MR_6.

Model transformations can be applied either randomly or using deterministic heuristics, although in certain cases, the order in which metamorphic relations are applied matters. In CUDF, for instance, non–constraining packages should be added (MR_6) before adding constraints depending on them (MR_7, MR_8 and MR_9). Similarly, relations MR_{12} and MR_{13} (adding disjunctions with existing variables), cannot be applied until at least one variable is added to the CNF formula. Notice, however, that this does not affect the diversity of the VMs that can be generated. The generation process can be stopped after a given number of transformations, or as soon as a VM and its corresponding set of valid configurations is generated and some desired properties are achieved, e.g. a certain number of variants or configurations is reached. In the example, configuration C_1 (i.e. package A) is marked as installed at the end of the process to simulate the current status of the system. Notice that this implies no changes in the set of valid configurations.

4.2. Test data extraction

Once a VM with the desired properties is generated, it is used as a non–trivial input for the test. Simultaneously, its generated set of valid configurations is automatically inspected to obtain the output of the analysis operations under test. As an example, consider the CUDF document D_M and

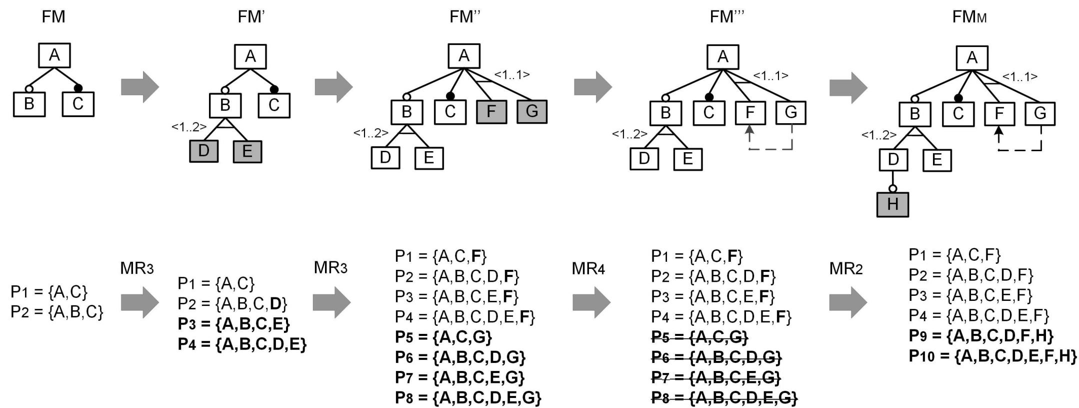

Figure 8. Random generation of a feature model and its set of products using metamorphic relations

its set of valid configurations generated in Fig. 7. The expected output of a number of analysis operations on the document can be obtained by inspecting the set of valid configurations, e.g.:

- *Is D_M consistent?* Yes, it represents at least one valid configuration.
- *How many different valid configurations does D_M represent?* 4 different configurations.
- *Is C = [(A,2),(B,4)] a valid configuration of D_M?* Yes. It is included in its set of valid configurations as C_3.
- *Does D_M contain any* dead *package, i.e. a package that cannot be installed [29]?* No, all packages are included in the set of valid configurations.

Let us consider that a request stanza such as "Install: B" is added to D_M, i.e. the user wishes to upgrade the current system configuration by installing the package B. The valid configurations of D_M fulfilling the user request, those including package B (C_2, C_3, and C_6), can be easily obtained from the set of valid configurations of D_M.

More importantly, we can also inspect the set of valid configurations to compute the expected output of certain optimization operations. For instance, the so–called *paranoid optimization criterion* [2, 43] is used to search for a configuration that fulfils the request while minimizing the number of uninstalled packages and, with less priority, the number of total changes, i.e. the number of installed and uninstalled packages. In our example, upgrading the system according to C_2 implies two changes, installing B and uninstalling A; opting for C_3 implies one change, installing B; and opting for 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 automatically obtained by iterating over the set of configurations and selecting those that: i) satisfy the user request, ii) have a minimum number of uninstalled packages, and iii) have a minimum number of changes. Notice that upgradeability problems may have more than one possible solution.

Another example is shown in Fig. 8, which depicts how our approach is used for the generation of a sample FM and its set of products. The generation starts from scratch with a trivial FM and its corresponding set of products. 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 the cross–tree constraint "G **requires** F". According to MR_4, the new set of products must be the set of FM'' excluding those products containing G but not F. If we consider the model FM_M obtained as a result of the process, we can easily find the expected output of most of the FM analysis operations defined in the literature [7] 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 does FM_M represent?* 6 different products.

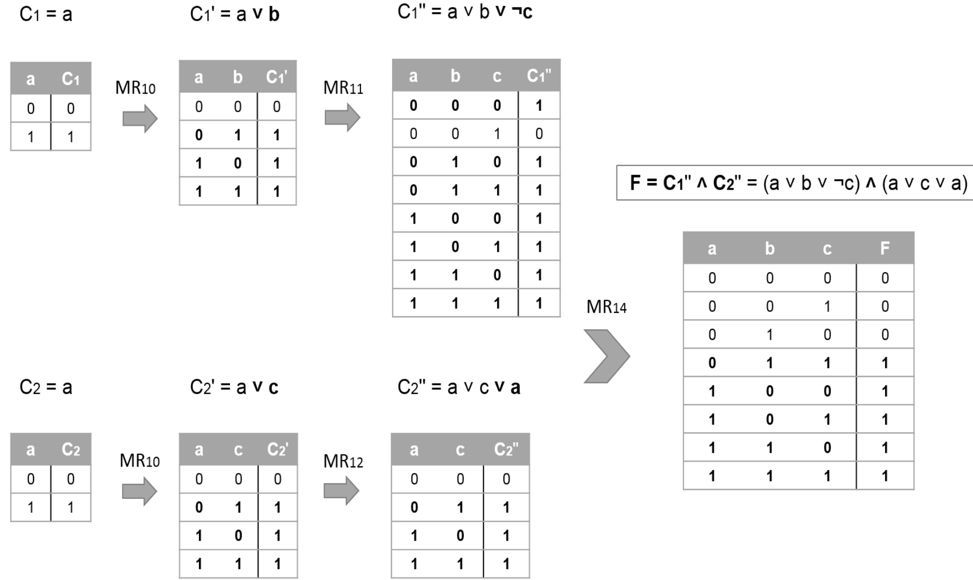

Figure 9. Random generation of a CNF formula and its set of solutions using metamorphic relations

- *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 FM_M.

Finally, Fig. 9 illustrates an example of how metamorphic relations can be used to generate CNF formulas as input test data and their respective solutions as expected output test data. First, a trivial 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 neighbour CNF formulas. On each step, a new disjunction is added to the clause, and the set of solutions is updated applying the metamorphic relations M_{10} to 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 corresponding set of solutions SAT(C_1), SAT(C_2) ... SAT(C_n). Then, the final CNF formula is created as a conjunction of the clauses previously created, i.e. $F = C_1 \wedge C_2 \ldots \wedge C_n$. The final set of solutions is computed using the metamorphic relation M_{14}, which obtains the intersection of the set of solutions of the clauses in the formula, i.e. SAT(F) = SAT(C_1) ∩ SAT(C_2) ... ∩ SAT(C_n). In the example, F is composed of two clauses, C_1'' and C_2''; three variables, a, b, and 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 defined in Section 3. 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.

The experiments were performed by two teams in different execution environments for compatibility with the tools under test. The specific execution settings are described in a separated technical report due to space constraints [53].

Most of the reported faults were confirmed by the respective tool developers, the related literature, or fix reports. For each faulty tool, we contacted their developers (five in total, two of them authors) by e–mail, sending them information about the detected failures and the test case(s) reproducing them. Sometimes, developers were already aware of the bugs meanwhile in other cases they acknowledged them as new defects. Interestingly, some developers were curious about how we have detected the bugs and requested more information.

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 together with the exact set of products 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 [54]. 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 12 faults were found.

Experimental setup. We evaluated the effectiveness of our test data generator in detecting faults in three FM reasoners: FaMa Framework 1.1.2, SPLAR[¶] and FLAME 1.0. FaMa [28] and SPLAR [45, 58] 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 [25]. FaMa was tested with its default configuration. SPLAR is actually composed of two reasoners using SAT–based and BDD–based analysis, which were both tested. Tests with FLAME were performed as part of a previous contribution and reproduced for this evaluation [25]. In total, we tested 19 operations in FaMa, 18 en FLAME and 9 in SPLAR. A detailed description of the operations tested on each reasoner is provided in [53]. The name and formal semantics of the analysis operations mentioned in this article are based on the work presented in [25].

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. 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 generated models represented between 0 and 5,800 products. Then, we proceeded with test execution. For each test case, an 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 the Prolog interpreter. In the cases of operations receiving additional inputs apart from an 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. For those specific operations, an extra suite was generated composed of 1,000 pairs of FMs and their corresponding set of products. 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.

Analysis of results. Table I 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

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

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 [7] and the formal semantics described in [25]. 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 SPLAR 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. We patched faults F10 and F12 for further testing of the SPLAR reasoner. Finally, fault F11 was revealed in five operations 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 indication of the difficulty of each fault detection. 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.

Fault	Operation	Description	Failures
FaMa 1.1.2			
F1	Core features	Wrong output	21
F2	Atomic sets	Wrong output	208
F3	Valid configuration	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 I. Faults detected in FM reasoners

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 and their set of valid configurations. In this experiment, we evaluated the ability of

the test data generator to detect faults in three 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 [2, 49] 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 [5, 31] uses several C++ tools for Answer Set Programming (ASP), a declarative language. Cudf-check is a command line CUDF reasoner provided as part of the Debian cudf–tools package [19]. This tool is mainly used to check the validity of 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 [2, 43]. As described in Section 4, this criterion searches for a configuration that fulfils the user request and minimizes the number of changes in the system. 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 without requests and their corresponding set of valid configurations. We parametrically controlled the generation assuring that the documents had a fair proportion of all types of elements, i.e. dependencies, conflicts, version constraints, etc. We refer the reader to an external technical report for the specific parameters and values used for the generation [53]. 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 197,400 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 configurations 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. Test cases were generated in 8 minutes. The execution of test cases took 1 hour in cudf–check and less than 10 minutes in the rest of solvers.

Analysis of results. The results revealed 2 faults in the p2cudf reasoner, shown in Table II. 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.me-tadata.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 return a wrong output. It is worth mentioning 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 and never in the Mancoosi competition benchmarks. More precisely, the nested disjunctions are simply ignored by the tool, thus the dependencies become much stronger and the number of solutions is reduced. As a result, the tool may provide either a suboptimal solution or consider that there is no solution at all. 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. One of the failure was a suboptimal solution, while the remaining three failures were incorrectly answering that the problem did not have any solution. 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 tools under test.

Fault	Operation	Description	Failures
F13	Paranoid	Exception	42
F14	Paranoid	Wrong output	4

Table II. 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 and the set of solutions of the formula. For its evaluation, we automatically tested nine SAT solvers in which 5 bugs were 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 [40], Minisat 2.2 [27], Clasp 2.1.3 [32], Picosat 535 [9], Rsat 2.0 [50], March_ks (2007) [33], March_rw (2011) [33], and Kcnfs 1.2 [24]. In a related work [11], 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 in order 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 was 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 [11] in order to make our results comparable. In particular, we first used our metamorphic test data generator to generate 10,000 random CNF formulas in the 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 found solution, and a last time to check that no more solution exists, each reasoner was expected to answer SAT 1,817,142 times, the 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 the analysis results in the next paragraph). Notice 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 file system I/O operations were performed in that case and, more importantly, the solver could 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 III 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 that are highly tested and validated by their user community. 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 III. 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. This timeout was reached in 952 out of the 10,000 test cases.

In [11], 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 [11]. 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. To evaluate both relations, we enabled them 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 to F18 was significantly low, ranging from 1 to 38, out of 10,000. This again demonstrates how hard is to detect certain bugs and motivates the need for automated testing techniques. This also suggests that using a larger test suite could have revealed more bugs.

6. APPLICATION TO OTHER VARIABILITY ANALYSIS DOMAINS

Based on our experience, we present some guidelines for the application of our metamorphic testing approach to similar domains. Given a variability modelling language, we propose the following steps to test their analysis tools, namely:

1. *Identify variants*. These are the basic units that can be combined into valid configurations, e.g. features in FMs, packages in CUDF documents or Boolean variables in CNF formulas.

2. *Identify variability constraints*. These are the constraints that restrict how variants can be combined. For example, FMs have five different types of feature constraints: mandatory, optional, set, requires and excludes (see Section 2.1).

3. *Define metamorphic relations*. Define a metamorphic relation for each identified variability constraint. Each relation should relate the set of configurations represented by a VM before (source) and after (follow–up) adding the variability constraint. Ideally, it should be defined a metamorphic relation for each variability constraint of the language to foster diversity during test data generation. This is supported by related works on the effectiveness of metamorphic relations, which suggest that good metamorphic relations should be diverse and semantically rich, i.e. rely on the semantics of the system under test [15, 41, 44].

4. *Generate test data*. Apply the metamorphic relations iteratively to generate increasingly larger and more complex VMs and their corresponding set of valid configurations. In certain domains, it may be required to apply the relations in a certain order. For instance, packages must be added to CUDF documents (MR_6) before adding constraints referencing them (MR_7, MR_8, and MR_9). Generation parameters (e.g. size of the models) should foster diversity assuring a balance among i) number of test cases, i.e. the more the better, ii) size range of the input models, i.e. the wider, the better, and iii) testing time, i.e. time available for the generation and execution of test cases. In general terms, test cases should be simple, numerous and diverse rather than complex, few and homogeneous.

5. *Run test cases*. Run the tools under test using the generated variability models as inputs and their set of configurations as oracles to determine the correctness of the outputs.

7. LIMITATIONS

The number of configurations generated by our test data generators increases exponentially with the size of the corresponding VM. As a result, our approach is unable to generate large, hard–to–analyse VMs. 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 with different complexity degrees, most of them easy to process, in order to exercise as many execution paths as possible. This is supported by our previous works with FMs and mutation, in which we found that most faults were detected by small inputs [55, 56]. Having said that, we emphasize that our test data generators can efficiently generate VMs representing hundreds of thousands of configurations, something that goes well beyond the scope of manual testing.

8. THREATS TO VALIDITY

The main factors that could have influenced our results are summarized in the following *construct*, *internal* and *external* validity threats [67]. Notice that the *conclusion* validity threats are not applicable to our work due to the nature of the experiments.

Construct validity. This validity threat is concerned with the relation between theory and observation [67]. In our case, whether the observed number of failures reflects the real number of faults. As mentioned in Section 5, most faults were confirmed by either the respective tool developers, the related literature, or fix reports. In a few cases (F11, F14 to F19), we could confirm the failures but not the faults causing them. Hence, there is a chance that faults F15 to 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 they are caused by the interaction of more than one fault. A related risk is derived from possible misunderstandings in the interaction with the developers. To minimize this threat, we contacted the tool developers in written form, by e-mail, sending them the failed test cases to reproduce the failures. Despite this, we must admit a small margin of error (above or below) in the number of reported faults.

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, 4 were developed by some of the authors, meanwhile 11 of them were developed by external developers. This clearly shows the black–box nature of the work, testing analysis tools with no prior 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 documents and CNF formulas.

External validity. Regarding the generalization of the conclusions, we evaluated our approach with three different variability languages, a number that could seem insufficient for the generalization of the conclusions of our study. We remark, however, that these languages are used in completely different domains, and have particularities such as hierarchical constraints in FMs, version and installation constraints in CUDF documents, or negated variables in Boolean formulas, that make them sufficiently heterogeneous. Beside these particularities, the three variability languages include constraints with similar semantics, e.g. **excludes** in FMs is very similar to conflicts in CUDF. These constraints are very common in variability modelling, something that suggests that our approach could be easily applicable to other variability languages such as orthogonal variability models [51] or decision models [59].

Since FMs and CUDF documents can be translated into (pseudo) Boolean formulas, it could be argued that working directly with Boolean formulas is a simpler and more generic approach. We did not adopt this approach for two reasons. First, a bidirectional translation from high–level variability models to Boolean formulas is a complex, language–specific task [20]. Second, and more importantly, translating models to formulas, forwards and backwards, would make test data generators very complex and probably more error–prone than the analysis tool under test.

Finally, the power of the presented approach relies on the use of metamorphic relations to construct VMs and their exact set of configurations. The construction of the set of configurations depends on the completeness of the metamorphic relations, i.e. ensuring that all the valid configurations of the input VM are generated. The presented metamorphic relations are a logical consequence of the semantics of the corresponding variability languages, therefore they are as generation-–complete as the original semantics, which are complete by definition. There is a chance, however, that mistakes during the definition of the metamorphic relations could lead to incomplete configuration sets. This threat was discarded in our work, in which thousands of test cases where run in 15 tools in three different domains without finding a single inconsistency in the generated configuration sets. We admit, however, that identifying, implementing, and guaranteeing completeness in more complex variability analysis domains such as attributed variability models [7], could be challenging.

9. RELATED WORK

In the following sections, we present the related works in the areas of testing SAT, FM and CUDF reasoners as well as those in the fields of metamorphic and automated testing.

9.1. Testing SAT reasoners

Brummayer et al. [11] presented a fuzzy 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*". Notice that SAT solvers can also be equipped to produce UNSAT proofs to be checked by independent external tools [63]. A similar approach for testing ASP reasoners was presented by Brummayer and Järvisalo in [10]. Artho et al. [3] proposed a model–based testing approach to test sequences of method 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 and applicable to different variability languages and tools regardless of their implementation details, i.e. we follow a 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 the context of performance testing, some authors have presented algorithms for the automated generation of computationally–hard SAT problems [18]. 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 assuring at each step that the formula is still (un)satisfiable. This procedure can also be used for the automated detection of functional faults. Our work, however, goes a step further since it allows not only knowing whether the input model is satisfiable or not, but also its exact set of solutions. This enables testing not only the satisfiability operation, but any analysis operation that can be expressed as a function on the set of solutions.

9.2. Testing FM reasoners

In [52], 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, generalisability and effectiveness.

In terms of performance testing, some algorithms and tools have been presented for the generation of random [54] and computationally–hard FMs [57]. In contrast to our work, these approaches generate FMs (input) but not their configurations (output), and therefore are not suitable to detect functional faults in FM reasoners.

9.3. Testing CUDF reasoners

The 2012 Mancoosi solver competition provided a solution checker to assess the correctness of the solutions returned by the competitor CUDF reasoners [43], i.e. redundant testing. Other related works have been presented in the context of package–based distributions. Vouillon and Di Cosmo [23] proposed a theoretical framework to detect co–installability conflicts, i.e. packages that cannot be installed together. Artho et al. [4] presented a case study classifying the types of conflicts found in two specific distributions, Debian and Red Hat. In [29], some of the authors proposed using variability analysis techniques for the automated analysis of Debian repositories. Compared to them, our work contributes to detect bugs in package management tools overcoming the oracle problem rather than analysing variability in package repositories.

9.4. *Metamorphic testing*

Kuo et al. [36] presented an approach for the automated detection of faults in decision support systems. In particular, they focused in the so–called *Multi–Criteria Group Decision Making* (MCGDM), in which decision problems are modelled as a matrix with several dimensions: alternatives, criteria and experts. They also 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 VMs could be used as decision models during software configuration. Also, as in our work, Kuo et al. used metamorphic relations to actually construct the expected 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.

Chen et al [15] investigated how to select effective metamorphic relations and concluded that good relations are those that lead to program executions as "different" as possible, e.g. in terms of paths traversed. Mayer et al. [44] presented a set of general rules to assess the suitability of metamorphic relations. Among others, the authors defined as good metamorphic relations those based on the semantics of the system under test. Liu et al. [41] presented an empirical study on the effectiveness of metamorphic testing to alleviate the oracle problem and concluded that diversity is an important aspect to increase the fault–detection capability of metamorphic relations. Our work supports previous results showing the effectiveness of using diverse metamorphic relations derived from the semantics of the variability languages under analysis.

Liu et al. [42] proposed composing metamorphic relations to create new relations from existing ones. Wu [68] proposed applying metamorphic relations iteratively as a way to improve their fault–detection capability. Both approaches were evaluated using case studies and mutation testing. These works are similar to ours in the sense that metamorphic relations are applied iteratively creating increasingly larger and more complex test cases. Compared to them, however, our approach enables the construction of the expected outputs rather than comparing source and follow–up test cases. Also, our work focuses on a specific domain, variability analysis, where 19 real bugs were uncovered.

Regarding the detection of real bugs, Xie et al. integrated program slicing and metamorphic testing, detecting two bugs in the Siemens Suite [69]. Although in a different domain, their work supports our results on the effectiveness of metamorphic testing in detecting real faults.

9.5. *Automated testing*

The automated generation of test cases is a hot research topic that involves numerous techniques [1]. *Adaptive random testing* [16] proposes using random inputs spread across the input domain of the system under test. *Combinatorial interaction testing* [17, 48] systematically select inputs that may reveal failures caused by the interaction between two or more input values. *Model–based testing* [62] use system models like finite state machines to derive test suites using a test criterion based on a test hypothesis justifying 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. the oracle problem. In contrast, our approach overcomes both problems, automated generation of inputs and expected outputs, providing a fully automated fault detection mechanism.

10. CONCLUSIONS

In this article, we have 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 valid configurations, from which the expected output of a number of analysis operations can be derived, thus overcoming the oracle problem. Among others analysis operations, we automatically generated test data for an optimization operation which is a novelty on the application of metamorphic testing to variability analysis tools. A key benefit of this approach is its applicability to any variability language with common variability constraints in which metamorphic relations can be identified. In this sense, we present some guidelines for the application of our metamorphic approach to similar variability analysis domains. 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 FMs, CUDF documents and CNF formulas. In total, we automatically detected 19 real bugs in 7 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 reinforces the potential of metamorphic testing as an automated testing technique.

MATERIAL

The source code of the test data generators as well as the test data and test results (CSV format) of the evaluation are available at `http://www.lsi.us.es/~segura/files/material/STVR14/`.

ACKNOWLEDGMENTS

We appreciate the help of Dr Martin Monperrus whose comments and suggestions helped us to improve the article 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.

This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT projects TAPAS (TIN2012-32273) and SAAS FIREWALL (IPT-2012-0890-390000) and the Andalusian Government projects THEOS (TIC-5906) and COPAS (P12-TIC-1867).

References

1. S. Anand, E. Burke, T. Y. Chen, J. Clark, M. Cohen, W. Grieskamp, M. Harman, M. Harrold, and P. McMinn. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, August 2013.
2. L. Argelich, D. Le Berre, I. Lynce, J. Silva, and P. Rapicault. Solving linux upgradeability problems using boolean optimization. In I. Lynce and R. Treinen, editors, *Workshop on Logics for Component Configuration*, volume 29 of *EPTCS*, pages 11–22, 2010.
3. C. Artho, A. Biere, and M. Seidl. Model-based testing for verification back-ends. In *7th International Conference on Tests & Proofs*, Budapest, Hungary, 2013. Springer.
4. C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Why do software packages conflict? In *9th IEEE Working Conference of Mining Software Repositories*, pages 141–150, 2012.
5. aspcud. `http://www.cs.uni-potsdam.de/wv/aspcud`. Accessed November 2013.
6. D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. In *17th International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 3520 of *Lecture Notes in Computer Sciences*, pages 491–503. Springer–Verlag, 2005.
7. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010.
8. T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In *International Conference on Automated Software Engineering (ASE'10)*, pages 73–82, 2010.
9. A. Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.
10. R. Brummayer and M. Järvisalo. Testing and debugging techniques for answer set solver development. *Journal of Theory and Practice of Logic Programming*, 10(4-6):741–758, July 2010.

11. R. Brummayer, F. Lonsing, and A. Biere. 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, pages 44–57, Berlin, Heidelberg, 2010. Springer-Verlag.
12. W. Chan, S. Cheung, and K. Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4(2):61–81, 2007.
13. T. Chen, S. Cheung, and S. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, University of Science and Technology, Hong Kong, 1998.
14. T. Chen, J. Feng, and T. Tse. Metamorphic testing of programs on partial differential equations: a case study. In *Proceedings of the 26th International Computer Software and Applications Conference*, pages 327–333, 2002.
15. T. Chen, D. Huang, T. Tse, and Z. Zhou. 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)*, pages 569–583, 2004.
16. T. Chen, F. Kuo, R. Merkel, and T. Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, Jan. 2010.
17. M. Cohen, M. Dwyer, and S. Jiangfan. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *Software Engineering, IEEE Transactions on*, 34(5):633–650, 2008.
18. S. Cook and D. Mitchell. Finding hard instances of the satisfiability problem: A survey. In *Satisfiability Problem: Theory and Applications*, volume 35 of *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–17. American Mathematical Society, 1997.
19. Cudf-tools debian package http://packages.debian.org/wheezy/cudf-tools. Accessed November 2013.
20. K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *11th International Software Product Line Conference (SPLC)*, pages 23–34, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
21. Debian reference guide. http://www.debian.org/doc/manuals/debian-reference/. Accessed November 2013.
22. Debian 7.0 wheezy released, May 2013. Accessed November 2013.
23. R. Di Cosmo and J. Vouillon. On software component co-installability. In *13th European conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 256–266, New York, NY, USA, 2011. ACM.
24. O. Dubois and G. Dequen. A backbone-search heuristic for efficient solving of hard 3-sat formulae. In B. Nebel, editor, *IJCAI*, pages 248–253. Morgan Kaufmann, 2001.
25. A. Durán, D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Flame: Fama formal framework (v 1.0). Technical Report ISA–12–TR–02, Seville, Spain, March 2012.
26. Eclipse marketplace http://marketplace.eclipse.org/. Accessed November 2013.
27. N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
28. FaMa Tool Suite. http://www.isa.us.es/fama/, Accessed November 2013.
29. J. Galindo, D. Benavides, and S. Segura. 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.
30. J. García-Galán, O. Rana, P. Trinidad, and A. Ruiz-Cortés. Migrating to the cloud: a software product line based analysis. In *3rd International Conference on Cloud Computing and Services Science (CLOSER'13)*, 2013.
31. M. Gebser, R. Kaminski, and T. Schaub. aspcud: A linux package configuration tool based on answer set programming. In C. Drescher, I. Lynce, and R. Treinen, editors, *Workshop on Logics for Component Configuration*, volume 65 of *EPTCS*, pages 12–25, 2011.
32. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. *clasp* : A conflict-driven answer set solver. In C. Baral, G. Brewka, and J. S. Schlipf, editors, *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.
33. M. Heule. *SmArT Solving: Tools and Techniques for Satisfiability Solvers*. PhD thesis, TU Delft, 2008.
34. M. Jang. *Linux Annoyances for Geeks*. O'Reilly, 2006.
35. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.
36. F. Kuo, Z. Zhou, J. Ma, and G. Zhang. Metamorphic testing of decision support systems: a case study. *Software, IET*, 4(4):294–301, 2010.
37. D. Le Berre and A. Parrain. On sat technologies for dependency management and beyond. In *First workshop on Analyses of Software Product Lines*, volume 2, pages 197–200, 2008.
38. D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010. system description.
39. D. Le Berre and P. Rapicault. Dependency management for the eclipse ecosystem: eclipse p2, metadata and resolution. In *Proceedings of the 1st international Workshop on Open Component Ecosystems*, IWOCE '09, pages 21–30, New York, NY, USA, 2009. ACM.
40. Lingeling sat solver. http://fmv.jku.at/lingeling/. Accessed November 2013.
41. H. Liu, F. Kuo, D. Towey, and T. Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, January 2014.
42. H. Liu, X. Liu, and T. Chen. A new method for constructing metamorphic relations. In *12th International Conference on Quality Software (QSIC)*, pages 59–68, Aug 2012.
43. Mancoosi european research project. http://www.mancoosi.org/. Accessed November 2013.
44. J. Mayer and R. Guderlei. An empirical study on the selection of good metamorphic relations. In *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 01*, pages 475–484, Washington, DC, USA, 2006. IEEE Computer Society.
45. M. Mendonca, M. Branco, and D. Cowan. 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), pages 761–762, Orlando, Florida, USA, October 2009. ACM.
46. M. Mendonca, A. Wasowski, and K. Czarnecki. SAT–based analysis of feature models is easy. In *Proceedings of the International Sofware Product Line Conference (SPLC)*, 2009.
47. C. Müller, M. Resinas, and A. Ruiz-Cortés. Automated Analysis of Conflicts in WS–Agreement Documents. *IEEE Transactions on Services Computing*, 2013.
48. C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11:1–11:29, Feb. 2011.
49. p2cudf http://wiki.eclipse.org/Equinox/p2/CUDFResolver. Accessed November 2013.
50. K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In J. Marques-Silva and K. A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.
51. K. Pohl, G. Bückle, , and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer–Verlag, 2005.
52. S. Segura, D. Benavides, and A. Ruiz-Cortés. Functional testing of feature model analysis tools: a test suite. *Software, IET*, 5(1):70–82, 2011.
53. S. Segura, A. Duran, A. Sánchez, D. Le Berre, E. Lonca, and A. Ruiz-Cortés. Automated metamorphic testing on the analysis of software variability. Technical Report ISA-2013-TR-03, ISA Research Group, Seville, Spain, December 2013. http://www.lsi.us.es/~segura/files/papers/segura13-TR-03.pdf.
54. S. Segura, J. Galindo, D. Benavides, J. Parejo, and A. Ruiz-Corts̈. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In U. Eisenecker, S. Apel, and S. Gnesi, editors, *Sixth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12)*, pages 63–71, Leipzig, Germany, 2012. ACM.
55. S. Segura, R. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated test data generation on the analyses of feature models: A metamorphic testing approach. In *International Conference on Software Testing, Verification and Validation*, pages 35–44, Paris, France, 2010. IEEE press.
56. S. Segura, R. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology*, 53:245–258, 2011.
57. S. Segura, J. Parejo, R. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated generation of computationally hard feature models using evolutionary algorithms. *Expert Systems with Applications*, 41(8):3975 – 3992, 2014.
58. Software Product Lines Automated Reasoning library (SPLAR) http://code.google.com/p/splar/. Accessed November 2013.
59. R. Stoiber and M. Glinz. Supporting stepwise, incremental product derivation in product line requirements engineering. In *International Workshop on Variability Modelling of Software-intensive Systems.*, volume 37, pages 77–84, 2010.
60. M. Svahnberg, L. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. *Software Practice and Experience*, 35(8):705–754, 2005.
61. R. Treinen and S. Zacchirol. Common Upgradeability Description Format (CUDF) 2.0. Technical Report 003, The Mancoosi project (FP7), 2009.
62. M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing Verification and Reliability*, 22(5):297–312, Aug. 2012.
63. A. Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.
64. B. Veer and J. Dallaway. The ecos component writer's guide. http://ecos.sourceware.org/ecos. Accessed November 2013.
65. H. Wang, Y. Li, J. Sun, H. Zhang, and J. Pan. Verifying feature models using OWL. *Journal of Web Semantics*, 5:117–129, June 2007.
66. E. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.
67. C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer, 2012.
68. P. Wu. Iterative metamorphic testing. In *29th Annual International Computer Software and Applications Conference, COMPSAC.*, volume 1, pages 19–24, July 2005.
69. X. Xie, W. Wong, T. Chen, and B. Xu. Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology*, 55(5):866 – 879, 2013.
70. Z. Zhou, D. Huang, T. Tse, Z. Yang, H. Huang, and T. Chen. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology*, pages 346–351, 2004.