The Drupal Framework: A Case Study to Evaluate Variability Testing Techniques

Ana B. Sánchez, Sergio Segura, and Antonio Ruiz-Cortés
Department of Computer Languages and Systems
University of Seville, Spain
{anabsanchez,sergiosegura,aruiz}@us.es

ABSTRACT

Variability testing techniques search for effective but manageable test suites that lead to the rapid detection of faults in systems with high variability. Evaluating the effectiveness of these techniques in real settings is a must but challenging due to the lack of variability-intensive systems with available code, automated tests and fault reports. In this paper, we propose using the Drupal framework as a case study to evaluate variability testing techniques. First, we represent the framework variability as a feature model. Then, we report on extensive data extracted from the Drupal git repository and the Drupal issue tracking system. Among other results, we identified 378 faults in single features and 11 faults triggered by the interaction between two of the features of Drupal v7.23, reported during a one-year period. These data may give a new insight into the distribution of faults in variability-intensive systems and the fault propensity of features. To show the feasibility of our work, we used the case study to evaluate the effectiveness of a history-based test case prioritization criterion. Results suggest that this technique could contribute to accelerate the detection of faults of test suites based on combinatorial testing.

Keywords

Variability, testing, feature model, automated testing, test case selection, test case prioritization.

1. INTRODUCTION

Variability determines the ability of software applications to be configured and customized. One of the most prominent examples of variability-intensive systems are Software Product Lines (SPLs). An SPL is a family of related software products. Each product represents a specific combination of features of the SPL. Institutions such as General Motors, NASA and Boeing are using SPL technology to decrease time to market and improve software quality [1]. Typically, SPLs are modelled by using variability models such as feature models [3]. A feature model represents the set of products of an SPL in terms of features and relationships among them (see Fig. 3).

The number of configurations and dependencies in variability models is potentially huge. For instance, the Billing feature model available in the SPLOT repository has 88 features and 66% of them are connected by constraints, representing more than 1 billion of potential products [15]. This explosion of combinations makes testing variability-intensive systems a major challenge. To address this problem, researchers have proposed various techniques to reduce the cost of testing in the presence of variability, including test case selection and test case prioritization techniques. Test case selection approaches [8, 17, 24] select an appropriate subset of the existing test suite according to some coverage criteria. Test case prioritization approaches [9, 19, 22, 24] schedule test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goal, e.g. accelerate the detection of faults [19].

The number of works on variability testing is growing rapidly and thus the number of experimental evaluations [6, 9, 16]. However, it is hard to find real variability-intensive systems with available code, test cases, detailed fault reports and good documentation that enable reproducible experiments [20, 21]. As a result, authors often evaluate their testing approaches using artificial variability models [8, 9] and simulated faults [6] which introduce threats to validity and weaken their conclusions. A related problem is the lack of information about the distribution of faults in variability-intensive systems, e.g. number and types of faults, fault severity, etc. This may be an obstacle for the design of new testing techniques since researchers are not fully aware of the type of faults they are looking for.

In the search for real variability systems some authors have explored the domain of open source operating systems [7, 13, 21]. However, these works mainly focus on the variability modelling perspective and thus ignore relevant data for testers such as the number of test cases or the distribution of faults, i.e. how faults are distributed over the different features. Also, SPL2GO[1] is a catalog of SPLs for which source code and variability model (e.g. feature model) are publicly available. However, these also lack of detail data about the faults found in the programs and it is up to the user to inspect the code searching for test cases.

In order to look for a real variability-intensive system with available code, we followed the steps of previous authors and looked into the open source community. In particular, we found the open source Drupal framework [4] to be a

[1] http://spl2go.cs.ovgu.de/

VaMoS '14 Nice, France.

http://dx.doi.org/10.1145/2556624.2556638.

motivating variability-intensive system. Drupal is a modular web content management framework written in PHP [4, 23]. Drupal provides detailed fault reports including fault description, fault severity, type, status and so on. Also, most of the modules of the framework include a number of automated test cases. The high number of the Drupal community members together with its extensive documentation have also been strengths to choose this framework. Drupal is maintained and developed by a community of more than 630,000 users and developers.

In this paper, we propose using the Drupal framework as a motivating case study to evaluate variability testing techniques. We propose mapping some of the main Drupal modules to features and represent the framework variability using a feature model. The resulting model has 28 features, 27 cross-tree constraints and represents 96,768 different Drupal configurations. Also, we report on extensive data extracted from the Drupal git repository and the Drupal issue tracking system referred to a period of one year. For each feature under study, we report its size, number of changes, number of test cases and a full fault report. Faults are classified according to their severity and the feature(s) that trigger it. The study of faults was replicated in two consecutive Drupal versions, v7.22 and v7.23, to study the evolution in the distribution of faults. Among other results, we identified 390 faults in total, 11 faults triggered by the interaction between two features and a single fault caused by the interaction among three features in Drupal v7.23. For the evaluation of our work, we used the case study to evaluate the effectiveness of test case prioritization based on historical of faults. More specifically, we used the information about the faults collected in Drupal v7.22 to effectively accelerate the detection of faults of a pairwise suite in Drupal v7.23. In summary, this work contributes to give a new insight into the distribution of faults in a real variability-intensive system and the fault propensity of its features. Also, this may be a valuable input to drive the design and evaluation of variability testing approaches.

The rest of the paper is structured as follows: Section 2 introduces the Drupal framework. Section 3 presents the Drupal feature model. The information about Drupal features (size, changes and tests) is presented in Section 4. The description about faults detected in Drupal v7.22 and v7.23 is presented in Section 5. Section 6 summarizes the main conclusions of our study. Section 7 shows a preliminary evaluation of our case study. Finally, we summarize our conclusions in Section 8.

2. THE DRUPAL FRAMEWORK

Drupal is a highly modular open source web content management framework implemented in PHP [4, 23]. This tool can be used to build a variety of web sites including internet portals, e-commerce applications and online newspapers [23]. Drupal is composed of a set of modules. A *module* is a collection of functions that provide certain functionality to the system. Installed modules in Drupal can be enabled or disabled. An enabled module is activated to be used by the Drupal system. A disabled module is deactivated and adds no functionality to the framework. The modules can be classified into core modules and additional modules [4, 23]. The *core modules* are approved by the core developers of the Drupal community and they are included by default in the basic installation of Drupal framework. However, some of them can be enabled or disabled, while others must always be enabled. The core modules are responsible for providing the basic functionality that is used to support other parts of the system. The Drupal core includes code that allows the system to bootstrap when it receives a request, a library of common functions frequently used with Drupal, and modules that provide basic functionality like user management and templating. *Additional modules* can be divided into contributed modules and custom modules. *Contributed modules* are written by the Drupal community and shared under the same GNU Public License (GPL) as Drupal. *Custom modules* are those created by the developer of websites (third parties). Fig. 1 depicts some popular core and additional Drupal modules.

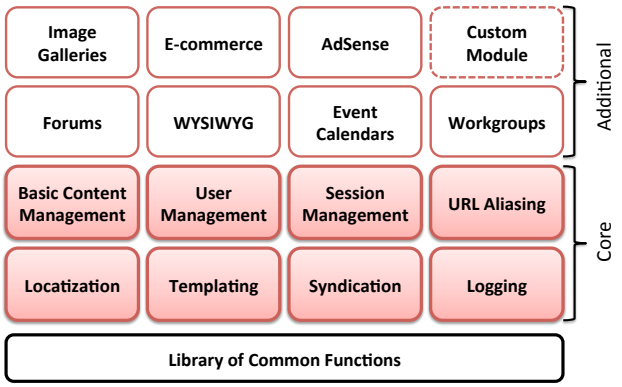


Figure 1: Drupal core and additional modules

2.1 Module structure

At the code level, every Drupal module is mapped to a directory including the source files of the module. These files may include PHP files, CSS stylesheets, JavaScript code, test cases and help documents. Also, every Drupal module must include a .module file and a .info file with meta information about the module. Besides this, a module can optionally include the directories and files of other modules, i.e. submodules. A submodule increases the functionality of the module containing it.

A Drupal .info file is a plain text file that describes the basic information required for Drupal to recognize the module. The name of this file must match the name of the module. This file contains a set of so-called directives. A *directive* is a property $name = value$. Some directives can use an array-like syntax to declare multiple values properties, $name[] = value$. Any line that begins with a semicolon (";") is treated as a comment. For instance, Listing 1 describes a fragment of the views.info file included in the *Views* module of Drupal v7.23:

Listing 1: Fragment of the file views.info

```
name = Views
description = Create customized lists and queries from your db.
package = Views
core = 7.x
php = 5.2

stylesheets[all][] = css/views.css

dependencies[] = ctools
; Handlers
files[] = handlers/views_handler_area.inc
files[] = handlers/views_handler_area_result.inc
files[] = handlers/views_handler_area_text.inc
```

```
... more
; Information added by drupal.org on 2013-04-09
version = "7.x-3.7"
core = "7.x"
project = "views"
```

The structure of .info files is standard across all Drupal 7 modules. The *name* and *description* directives specify the name and description of the module that will be displayed in the Drupal configuration page. The *package* directive defines which package or group of packages the module is associated with. On the modules configuration page, modules are grouped and displayed by package. The *core* directive defines the version of Drupal for which the module was written. The *php* property defines the version of PHP required by the module. The *files* directive is an array with the names of the files to be loaded by Drupal. Furthermore, the .info file can optionally include the *dependencies* that the module has with other modules, i.e. modules that must be installed and enabled for this module to work properly. In the example, the module *Views* depends on the module *Ctools*. There exist some core modules that always have to be installed and enabled, i.e. those including the directive $required = TRUE$.

2.2 Module tests

Drupal modules can optionally include a test directory with the test cases associated to the module. Drupal defines a test case as a class composed of functions (i.e. tests). These tests are performed through assertions, a group of methods that check for a condition and return a Boolean. If it is TRUE, the test passes, if FALSE, the test fails. There exist three types of tests in Drupal, unit, integration and upgrade test cases. *Unit tests* are methods that test an isolated piece of functionality of a module, such as functions or methods. *Integration tests* test how different components (i.e. functionality) work together. These tests may involve any module of the Drupal framework. Integration tests usually simulate user interactions with the graphical user interface through HTTP messages. According to the Drupal documentation[2], these tests are the most common tests in Drupal. *Upgrade tests* are used to detect faults caused by the upgrade to a newer version of the framework, e.g. from Drupal v6.1 to v7.1. In order to work with tests in Drupal it is necessary to enable the SimpleTest module. This module is a testing framework moved into core in Drupal v7. SimpleTest automatically runs the test cases of all the installed modules. Fig. 2 shows a snapshot of SimpleTest while running the tests (i.e. 328 test cases and 20,230 assertions) of Drupal modules.

3. THE DRUPAL FEATURE MODEL

In this section, we describe the process followed to model Drupal variability using a feature model, depicted in Fig. 3.

3.1 Feature tree

According to the Drupal documentation, each module that is installed and enabled adds a new *feature* to the framework [23]. Thus, we propose modelling Drupal modules as features of the feature model. Also, when a module is installed, new subfeatures can be enabled adding extra functionality to the module. These features are considered as children

[2]https://drupal.org/simpletest

Figure 2: Running Drupal tests

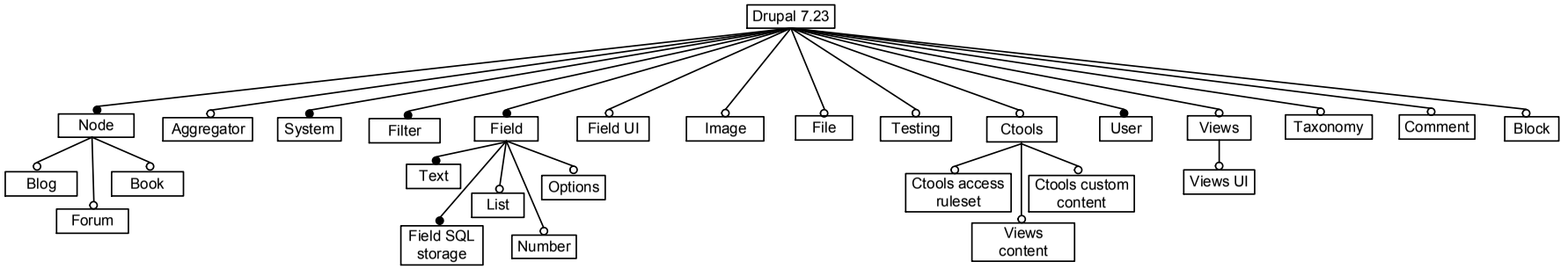
features of the module that contains them. Fig. 3 shows the Drupal features that were considered in our study, 28 in total including the feature root. In particular, we first selected those Drupal core modules that must be always enabled, 7 in total, e.g. *Node*. In Fig. 3, these features appear with a mandatory relation with the features root and *Field*. These features are included in all Drupal configurations. A Drupal configuration is a valid combination of features installed and enabled. Then, we randomly selected 20 modules within the rest of the Drupal core modules (e.g. *Image*) and additional modules (e.g. *Views*). All these modules can be optionally installed and enabled and thus were modelled as optional features in the feature model.

Subfeatures were modelled considering as children of other module those submodules that appear inside of its module directory. These submodules provide extra functionality to its parent module and they have no meaning without it. As an example, the feature *Ctools* presents several subfeatures such as *Ctools access ruleset*, *Ctools custom content* and *Views content*. Exceptionally, the submodules of *Node*, *Blog*, *Book* and *Forum*, appear in separate module folders, however, the description of the modules in the Drupal documentation indicates that these modules are specializations/types of *Node*. With respect to the relationships *or* and *alternative* none of them were identified among the features considered in Fig. 3.

3.2 Cross-tree constraints

We define the dependencies among modules as cross-tree constraints in the feature model. Constraints in feature models are typically of the form *requires* or *excludes*. If a feature A requires a feature B, the inclusion of A in a configuration implies the inclusion of B in such configuration. On the other hand, if a feature A excludes a feature B, both features cannot be part of the same configuration.

Cross-tree constraints were identified by manually inspecting the dependencies directive in the .info file of each module. For each dependency, we created a requires constraint in the feature model, 27 in total. For instance, consider the views.info file depicted in Listing 1. The file indicates that *Views* depends on the *Ctools* module, i.e. dependencies[]=ctools. Thus, we established a requires constraint between modules *Views* and *Ctools*. No exclude constraints were identified among the modules. Interestingly, we found



Forum Requires **Taxonomy**
Forum Requires **Options**
Forum Requires **Field**
Forum Requires **Field SQL storage**
Forum Requires **Comment**
Forum Requires **Text**
Text Requires **Field SQL storage**
List Requires **Field SQL storage**
List Requires **Options**
Number Requires **Field SQL storage**

Options Requires **Field SQL storage**
Field UI Requires **Field**
Field UI Requires **Field SQL storage**
Image Requires **File**
Image Requires **Field**
Image Requires **Field SQL storage**
File Requires **Field**
File Requires **Field SQL storage**
Views content Requires **Views**
Views Requires **Ctools**

Views UI Requires **Ctools**
Taxonomy Requires **Options**
Taxonomy Requires **Field**
Taxonomy Requires **Field SQL storage**
Comment Requires **Text**
Comment Requires **Field**
Comment Requires **Field SQL storage**

Figure 3: Drupal feature model

that all modules in the same version of Drupal are expected to work fine together. If a Drupal module presents incompatibilities with others, this is reported as a bug that must be fixed. As a sanity check, we confirmed the constraints identified using the JIT Drupal module (Javascript InfoVis Toolkit) which shows a graphical representation of the modules and their relationships.

The ratio of features involved in cross-tree constraints to the total number of features (CTCR) is 57.1%. The model represents 96,768 valid Drupal configurations.

4. DRUPAL FEATURES DATA

In this section, we report on the size, recent changes and test cases of the Drupal features shown in Fig. 3. These data are often used as good indicators of the error-proneness of a software application. Additionally, this may provide researchers and practitioners with helpful information about the characteristics of features in a real variability-intensive application. In particular, for each Drupal feature, we collected the following data:

Feature size. This provides a rough idea of the complexity of each feature and its error proneness. The size of a feature was calculated in terms of its number of Lines of Code (LoC). LoC were counted using the `wc` Linux command on each one of the source files included in the module directory associated to each feature. The command used is shown below. Blank lines and comments were included in the counting for simplicity. Test files were excluded from the counting. Table 1 depicts the number of LoC of each feature. The sizes ranged between 326 LoC (feature *Ctools custom content*) and 70,372 LoC (feature *Views*). It is noteworthy that subfeatures are significantly smaller than their respective parent features.

```
wc -l ‘find . -iname "module*" -o -iname "otherfile*"‘
```

Number of changes. Changes in the code are likely to introduce faults [24]. Thus, the number of changes in a feature may be a good indicator of its error proneness and could help us to predict faults in the future. To obtain the number of changes made in each feature, we collected the commits from the git repository of Drupal[3]. The search was narrowed by focusing on the changes performed during a period of one year, from September 30^{th} 2012 to September 29^{th} 2013. First, we cloned the entire Drupal v7.x repository. Then, we applied the console command showed below to get the number of commits by module, version and date. As illustrated in Table 1, the number of changes ranged between 0 (feature *Blog*) and 43 (feature *Views*). The modules with the highest number of commits are *Views* (43) and *Ctools* (24). It is noteworthy that these are the only additional modules considered in Fig. 3, i.e. not included in Drupal core.

```
git log --pretty=oneline --after={2012-09-30}
--before={2013-09-29} 7.22..7.23 name_module | wc -l
```

Number of tests. Table 1 shows the total number of test cases and assertions of each feature, obtained from the output of the *SimpleTest* module. In total, Drupal features include 328 test cases and 20,230 assertions. In some cases such as *Ctools*, the number of tests (7 test cases and 121 assertions) seems low considering that the size of the feature is over 9,000 LoC. It is also noteworthy that some of the subfeatures include no test cases.

Feature	LoC	Changes	Tests	
			TCs	Asserts.
Aggregator	4,384	1	12	985
Block	3,806	3	9	675
Blog	647	-	1	244
Book	2,734	1	1	531
Comment	6,547	1	14	3,287
Ctools	9,267	24	7	121
Ctools access ruleset	385	1	-	-
Ctools custom content	326	-	-	-
Field	9,561	9	9	870
Field SQL storage	1,428	2	1	94
Field UI	3,895	3	3	287
File	2,428	1	39	2,293
Filter	5,206	3	9	958
Forum	3,262	2	2	677
Image	6,570	3	13	811
List	638	1	3	232
Node	11,397	4	32	1,391
Number	632	1	1	87
Options	1,057	1	2	227
System	23,478	16	58	2,138
Taxonomy	6,625	4	14	677
Testing	3,002	-	12	219
Text	1,286	1	3	444
User	10,117	12	23	1,355
Views	70,372	43	51	1,089
Views content	3,468	1	-	-
Views UI	2,462	2	9	538
Total	**194,980**	**140**	**328**	**20,230**

Table 1: Feature data

5. FAULTS IN DRUPAL

In this section, we report the faults detected in the Drupal modules shown in Fig. 3. The collection of faults was carried out on the Drupal issue tracking system[4]. To get the information about faults, Drupal offers a web-based search tool to filter the faults by severity, status, module and Drupal version. The searches on Drupal issue tracking were performed in the date range from September 30^{th} 2012 to September 29^{th} 2013. We collected the faults of two consecutive Drupal versions, v7.22 and v7.23. We considered both versions to achieve a better understanding of the evolution of a real system and due to the existing interest in obtaining historical of faults for testing [5, 10].

The process of collecting faults was manual. First, we filtered the faults by the name of the module (i.e. label name_module.module in the issue tracking system). Second, we refined the search by the Drupal versions v7.23 and v7.22 and the dates previously mentioned. A total of 453 faults matched the initial search for Drupal v7.23 and a total of 481 faults for Drupal v7.22. In order to identify the faults

[3]http://drupalcode.org/project/drupal.git

[4]https://drupal.org/project/issues

Module	Drupal v7.22					Drupal v7.23				
	Severity				Total	Severity				Total
	Minor	Normal	Major	Critical		Minor	Normal	Major	Critical	
Aggregator	-	6	1	-	7	-	7	-	-	7
Block	2	12	2	1	17	2	9	2	1	14
Blog	-	2	1	-	3	-	1	1	-	2
Book	1	5	-	-	6	1	4	-	-	5
Comment	2	16	2	1	21	2	13	2	2	19
Ctools	7	130	20	8	165	7	130	20	8	165
Field UI	3	7	-	-	10	3	4	-	-	7
File	1	14	3	-	18	1	15	3	-	19
Filter	-	10	1	-	11	-	9	1	-	10
Forum	-	3	1	-	4	-	3	-	-	3
Image	-	9	1	1	11	-	9	1	1	11
Node	2	17	-	2	21	2	16	-	2	20
Number	-	1	-	-	1	-	-	-	-	-
System	4	23	4	2	33	4	20	4	2	30
Taxonomy	-	20	4	1	25	-	23	4	1	28
Testing	-	4	2	1	7	-	4	2	1	7
Text	-	5	-	-	5	-	4	-	-	4
User	4	15	-	-	19	4	13	-	-	17
Views	-	2	-	-	2	-	1	-	-	1
Views Content	1	18	2	-	21	1	18	2	-	21
Total	**27**	**319**	**44**	**17**	**407**	**27**	**303**	**42**	**18**	**390**

Table 2: Faults in Drupal v7.22 and v7.23

caused by the interaction of several modules among all the faults found, we first tried a similar approach to the one presented by Artho et al. [2] in the context of operating systems. This is, for each module, we searched for faults descriptions containing the keywords "break", "conflict" or "overwrite". However, the search did not return any result related with interaction among modules. Thus, we decided to follow a different approach. For each module, we searched for faults descriptions and tags including the name (without distinguishing between lower and upper case) of any of the other modules in Fig. 3. For instance, if a reported fault in the module *Ctools* included the name of the module *Taxonomy* in its description or tags, we selected the fault as a candidate integration fault between *Ctools* and *Taxonomy*. Then, we manually checked which of the faults found were actually integration faults. Several of the candidate integration faults were discarded after checking them. A total of 18 faults matched in the search of both versions: 17 out of 18 were caused by the interaction of two modules and one of them by the interaction of three modules. It is possible that some reporters could have not included the involved module names in some interaction faults. As a result, some of these faults could have not been identified.

Then, the search was refined to eliminate those faults that were not accepted by the Drupal community, namely: *duplicated bugs*, *non reproducible bugs* and *bugs working as designed*. The latter are issues that have been considered not to be a bug because the reported behaviour was either an intentional part of the project, or the issue was caused by customizations manually applied by the user. This left 390 valid faults in Drupal v7.23 and 407 in Drupal v7.22.

Table 2 summarizes the faults found in both version of Drupal modules. For each module, the total number of faults (single and interaction faults) is presented classifying them according to their severity level. Interaction faults are counted in the modules in which they were detected. Only the modules in which we found faults are shown. We may remark that there exist modules such as *Ctools* that do not discriminate among versions of Drupal 7, i.e. they only specify v7.x. This explains the same number and severity of faults that appear in Table 2 for *Ctools*. On the other hand, there exist some faults that appear in Drupal v7.22 and continue in v7.23. For example, the same minor fault of the module *Book* appears in both versions. This is not the case of the module *Aggregator* that presents the same number of faults in v7.23 and v7.22 but the faults are different.

Among the valid faults found, we identified 12 integration faults. The type of interaction and the name of the modules causing the fault are presented in Table 3. Among the integration faults detected, 11 were caused by the interaction between two modules and just one was caused by the interaction among three modules. All of these faults were identified in both Drupal versions. It is noteworthy that 8 out of 12 faults were caused by the interaction of *Ctools* with other module. *Ctools* provides a set of APIs and tools for modules to improve the developer experience. Also, 6 out of 12 faults were triggered by the interaction of the module *User* with other module. The module *User* allows users to register, log in, and log out and supports user roles and permissions. Finally, since the process of fault collection was manual, there exists the possibility that some faults could have not been correctly identified.

6. DISCUSSION

Next, we present some of the conclusions of our study:

Relation between faults and feature size. We found

that 7 out of the 10 largest features were also among the 10 features with a higher number of faults. Similarly, 8 out of 10 smallest features were among the 10 features with less faults. This suggests that there exists a certain correlation between the size of a feature and the probability of faults on it. However, we found some exceptions. For instance, the largest feature in Drupal, *Views*, has a single reported fault. Conversely, the feature with a highest number of faults, *Ctools*, is the sixth largest feature in Drupal. Thus, we conclude that the size of a feature could be interpreted as a rough estimation of its error-proneness.

Relation between faults and changes. We found that 6 out of the 10 features with a higher number of changes were also among the 10 features with a higher number of faults. In fact, the two features with the highest number of faults, *Ctools* and *System*, were the second and third features with more changes during the last 12 months respectively. We also observed that the features with few or no changes at all were among those with a lower number of faults. This also suggests a correlation between the number of changes in a feature and the probability of faults on it.

Relation between faults and type of features. We identified 81 faults in the core features of the Drupal feature model ($81/7 = 11.5$ faults per feature) and 309 faults in the optional features ($309/20 = 15.4$ faults per feature). Core features have a lower ratio of faults per feature. We presume that this is due to the fact that core features are included in all Drupal configurations and thus they are better supported and more stable. Regarding fault severity, around 78% of the faults were classified as normal, 11% as major, 7% as minor and 4% as critical. We found no correlation between fault severity and the types of features.

Relation between faults and CTC (Cross-Tree Constraints). In [6], Bagheri et al. studied several feature model metrics and suggested that those features involved in a higher number of CTCs are more error-prone. To explore this fact, we analyzed the features involved in constraints in order to study the relation between them and their fault propensity. In particular, we found that only 4 out of the 10 features involved in a higher number of CTCs were among the 10 features with a higher number of faults. In fact, the three features involved in a higher number of constraints in Drupal, *Field SQL storage* (10), *Field* (6) and *Forum* (6) had just three faults in total. Therefore, we conclude that the correlation between feature involvement in CTCs and fault propensity is not confirmed in our study.

Test cases. We found a clear relation between the size of the features and the number of test cases and assertions included as a part of their source code. Hence, for instance, the two largest Drupal features, *Systems* and *View*, were those including a larger number of test cases, 58 and 51 respectively. This seems reasonable since larger program require a higher number of tests. Regarding the distribution of assertions in the test cases results were disparate. As an example, feature *System* includes 2,138 assertions distributed along 58 test cases while feature *Comment* includes 3,287 assertions distributed along 14 test cases. We presume this is simply due to different developer's practices.

n-wise	Modules involved
2	Blog + User
2	Comment + User
2	Ctools + User
2	Ctools + Text
2	Ctools + User
3	Ctools + Views content + Filter
2	Ctools + Text
2	Ctools + Views
2	Ctools + Taxonomy
2	Ctools + Taxonomy
2	System + User
2	User + Image

Table 3: Interation faults in Drupal

7. EVALUATION

Previous studies show that the detection of faults in an application can be accelerated by testing first those components that showed to be more error-prone in previous versions of the software. This is referred to as history-based test case prioritization [5, 10]. In this section, we use the Drupal case study to evaluate whether this prioritization technique could contribute to accelerate the detection of faults of a pairwise test suite. Pairwise is a well-known test case selection approach that generates all possible combinations of pairs of features based on the observation that most faults originate from a single feature or by the interaction of two features [16]. In this scenario, we define a test case as a configuration of the Drupal framework to be tested, i.e. a set of features.

The evaluation was performed in several steps. First, we translated the Drupal feature model to SPLX format to make it machine-processable [14]. Then, we seeded the model with the faults detected in Drupal v7.23, 390 in total. To this purpose, we created a list of faulty feature sets. Each set simulated faults triggered by n features ($n \in [1, 3]$). For instance, the list {{Node}{Ctools,User}} simulates a fault in the feature *Node* and another fault caused by the interaction between the *Ctools* and *User* features. Second, we used the SPLCAT tool [11] to generate a pairwise suite for the Drupal feature model. As a result, we obtained a set of 10 test cases (out of 96,768) that covered all the possible pairs of feature combinations. Then, we checked whether the pairwise suite detected the seeded faults. We considered that a test case detects a fault if the test case includes the feature(s) that trigger the fault. The pairwise test suite detected all the seeded faults.

Next, we measured how fast the faults were detected by the pairwise suite calculating the APFD (Average Percentage of Faults Detected) metric [18]. The APFD metric measures the weighted average of the percentage of faults detected during the execution of the test suite [18]. To formally illustrate APFD, let T be a test suite which contains n test cases, and let F be a set of m faults revealed by T. Let TFi be the position of the first test case in ordering T' of T which reveals the fault i. The APFD metric for the test suite T' is given by the equation: $APFD = 1 - \frac{TF_1+TF_2+...+TF_n}{n \times m} + \frac{1}{2n}$. APFD value ranges from 0 to 1.

The resulting APFD value for the pairwise suite was 86.2%. Then, we prioritized the test suite placing first those test cases that included the features with more faults in the pre-

vious version of Drupal. More specifically, each test case was given a priority value equal to the sum of the faults detected in its features in Drupal v7.22. Then, the list was ordered scheduling first for testing those test cases with a higher priority value. The APFD metric for the prioritized suite was 94.2%, improving up to 8 points the APFD of the pairwise suite (86.2%). This suggests that using fault history information contributes to accelerate the detection of faults of test suites based on combinatorial testing.

8. CONCLUSIONS

In this paper, we presented the Drupal framework as a motivating real variability-intensive system. In contrast to the related work on variability modelling in open-source systems, we focused on the testing perspective providing detailed data about the fault distribution and faults propensity of its features. Also, we reported on how test cases are distributed and arranged in the framework providing helpful feedback to researchers and practitioners in the field of variability. All these data may be a valuable asset to evaluate variability testing techniques in a real setup rather than using random variability models and simulated faults. To show the feasibility of our work, we used the case study to evaluate the effectiveness of a history-based test case prioritization criterion. Results suggest that this approach in combination with pairwise could contribute to accelerate the detection of faults in variability-intensive systems.

9. ACKNOWLEDGMENTS

This work was partially supported by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programmes (grants TIN2009-07366 (SETI), TIN2012-32273 (TAPAS), TIC-5906 (THEOS), TIC-1867 (COPAS)).

10. REFERENCES

[1] S. Apel, A. von Rhein, P. Wendler, A. GröBlinger, and D. Beyer. Strategies for product-line verification: case studies and experiments. In *International Conference on Software Engineering*, 2013.

[2] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Why do software packages conflict? In *Conference on Mining Software Repositories*, 2012.

[3] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analyses of feature models 20 years later: A literature review. *Information Systems*, 2010.

[4] D. Buytaert. Drupal Framework. http://www.drupal.org, accessed in October 2013.

[5] E. C. P. Cristian Simons. Regression test cases prioritization using failure pursuit sampling. In *ISDA*, 2010.

[6] F. Ensan, E. Bagheri, and D. Gasevic. Evolutionary search-based test generation for software product line feature models. In *Conference on Advanced Information Systems Engineering (CAiSE'12)*, 2012.

[7] J. A. Galindo, D. Benavides, and S. Segura. Debian packages repositories as software product line models. towards automated analysis. In *ACOTA*, 2010.

[8] J. Guo, J. White, G. Wang, J. Li, and Y. Wang. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software*, 2011.

[9] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test suites for large software product lines. Technical report, 2012.

[10] Y.-C. Huang, C.-Y. Huang, J.-R. Chang, and T.-Y. Chen. Design and analysis of cost-cognizant test case prioritization using genetic algorithm with test history. In *Computer Software and Applications Conference*, 2010.

[11] M. F. Johansen, O. Haugen, and F. Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In *MODELS*, 2011.

[12] M. F. Johansen, O. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen. Generating better partial covering arrays by modeling weights on sub-product lines. In *International Conference MODELS*, 2012.

[13] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the linux kernel variability model. In *SPLC*, 2010.

[14] M. Mendonca. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, 2009.

[15] M. Mendonca, M. Branco, and D. Cowan. S.p.l.o.t. - software product lines online tools. In *OOPSLA*, 2009.

[16] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Budry, and Y. le Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 2011.

[17] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Conference Software Testing, Verification and Validation*, 2010.

[18] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. Software Eng*, 27:929–948, 2001.

[19] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. A comparison of test case prioritization criteria for software product lines. Technical report, University of Seville, Spain., 2013.

[20] S. Segura and A. Ruiz-Cortés. Benchmarking on the automated analyses of feature models: A preliminary roadmap. In *VaMoS*, 2009.

[21] S. She, R. Lotufo, T. Berger, A. Wasowski, and krzysztof Czarnecki. The variability model of the linux kernel. In *International Workshop on Variability Modelling of Software-intensive Systems*, 2010.

[22] H. Srikanth, M. B. Cohen, and X. Qu. Reducing field failures in system configurable software: Cost-based prioritization. 2009.

[23] T. Tomlinson and J. K. VanDyk. *Pro Drupal 7 development: third edition*. 2010.

[24] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. In *Software Testing, Verification and Reliability*, 2012.