Noname manuscript No.
(will be inserted by the editor)

Variability Testing in the Wild: The Drupal Case Study

Ana B. Sánchez · Sergio Segura · José A. Parejo · Antonio Ruiz-Cortés

the date of receipt and acceptance should be inserted later

Abstract Variability testing techniques search for effective and manageable test suites that lead to the rapid detection of faults in systems with high variability. Evaluating the effectiveness of these techniques in realistic settings is a must, but challenging due to the lack of variability intensive systems with available code, automated tests and fault reports. In this article, we propose using the Drupal framework as a case study to evaluate variability testing techniques. First, we represent the framework variability using a feature model. Then, we report on extensive non–functional data extracted from the Drupal Git repository and the Drupal issue tracking system. Among other results, we identified 3,392 faults in single features and 160 faults triggered by the interaction of up to 4 features in Drupal v7.23. We also found positive correlations relating the number of bugs in Drupal features to their size, cyclomatic complexity, number of changes and fault history. To show the feasibility of our work, we evaluated the effectiveness of non–functional data for test case prioritization in Drupal. Results show that non–functional attributes are effective at accelerating the detection of faults, outperforming related prioritization criteria as test case similarity.

1 Introduction

Software variability refers to the ability of a software system to be extended, changed, customized or configured to be used in a particular context [62]. Software applications exposing a high degree of variability are usually referred to as *Variability–Intensive Systems (VISs)*. Operating systems as Linux [41,58], development tools as Eclipse [35] or even cloud applications as the Amazon elastic compute service [25] have been reported as examples of VISs. Another prominent example of software variability is found in Software Product Lines (SPL). SPL engineering focuses on the development of families of related products trough the systematic management of variability. For that purpose, *feature models* are typically used as the facto standard for variability modelling in terms of functional features and constraints among them [8,36]. Features are often enriched with non–functional attributes in so-called *attributed feature models* [8]. A feature attribute is usually represented with a name and a value, e.g. cost = 20.

Testing VISs is extremely challenging due to the potentially huge number of configurations under test. For instance, *Debian Wheezy*, a well–known Linux distribution, provides more than 37,000 packages that can be combined with restrictions leading to billions of potential configurations [13]. 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 [29,31,53,68]. *Test case selection* approaches select an appropriate subset of the existing test suite according to some coverage criteria. *Test case prioritization* approaches 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.

The number of works on variability testing is growing rapidly and thus the number of experimental evaluations. However, it is hard to find real VISs with available code, test cases, detailed fault reports and good documentation that enable reproducible experiments

Department of Computer Languages and Systems.
University of Seville, Spain
E-mail: absanchez@us.es

[56,58]. As a result, authors often evaluate their testing approaches using synthetic feature models, faults and non-functional attributes, which introduces threats to validity and weaken their conclusions. A related problem is the lack of information about the distribution of faults in VISs, 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 that they are looking for.

In the search for real variability systems some authors have explored the domain of open source operating systems [10,24]. 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. Also, repositories such as SPLOT [44,60] and SPL2GO [59] provide catalogues of variability models and source code. However, they do not include information 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 find a real VIS 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 [12] to be a motivating VIS. Drupal is a modular web content management framework written in PHP [12,65]. 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 article, we propose using the Drupal framework as a motivating case study to evaluate variability testing techniques. In particular, the following contributions are presented.

1. We map some of the main Drupal modules to features and represent the framework variability using a feature model. The resulting model has 48 features, 21 cross–tree constraints and represents more than 2,000 millions of different Drupal configurations.
2. We report on extensive non–functional data extracted from the Drupal Git repository. For each feature under study, we report its size, number of changes (during two years), cyclomatic complexity, number of test cases, number of test assertions, number of developers and number of reported installations. To the best of our knowledge, the Drupal feature model together with these non–functional attributes represents the largest attributed feature model published so far.
3. We present the number of faults reported on the Drupal features under study during a period of two years, extracted from the Drupal issue tracking system. Faults are classified according to their severity and the feature(s) that trigger it. Among other results, we identified 3,392 faults in Drupal v7.23, 160 of them caused by the interaction of up to 4 different features.
4. We replicated the study of faults in two consecutive Drupal versions, v7.22 and v7.23, to enable fault–history test validations, i.e. evaluate how the bugs detected in Drupal v7.22 could drive the search for faults in Drupal v7.23.
5. We present a correlation study exploring the relation among the non–functional attributes of Drupal features and their fault propensity. The results revealed statistically significant correlations relating the number of bugs in features to the size and cyclomatic complexity of its code, number of changes and number of faults in previous versions of the framework.
6. We present an experimental evaluation on the use of black–box combinatorial testing and non–functional data for test case prioritization. The results of our evaluation shows that prioritization driven by non–functional attributes effectively accelerate the detection of faults of combinatorial test suites, outperforming related functional prioritization criteria such as variability coverage [21,54] and similarity [2,31,54].

These contributions provide a new insight into the functional and non–functional aspects of a real open–source VIS. This case study is intended to be used as a realistic subject for further and reproducible validation of variability testing techniques. It may also be helpful for those works on the analysis of attributed feature models. Last, but not least, this work supports the use of non–functional attributes as effective drivers for test case prioritization in the presence of variability.

The rest of the article is structured as follows: Section 2 introduces the Drupal framework. Section 3 describes the Drupal feature model. A complete catalogue of non–functional feature attributes is reported in Section 4. The number and types of faults detected in Drupal are presented in Section 5. Section 6 presents a correlation study exploring the relation among the reported non–functional attributes and the fault propensity of features. The results of using non–functional attributes to accelerate the detection of faults in Drupal are presented in Section 7. Section 9 discusses

the threats to validity of our work. The related work is introduced and discussed in Section 10. We summarize our conclusions in Section 11. Finally, in Section 12, we provide the produced material.

2 The Drupal framework

Drupal is a highly modular open source web content management framework implemented in PHP [12,65]. It can be used to build a variety of web sites including internet portals, e-commerce applications and online newspapers [65]. 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 [12,65]. *Core modules* are approved by the core developers of the Drupal community and are included by default in the basic installation of Drupal framework. They 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. In turn, core modules can be divided into core compulsory and core optional modules. *Core compulsory modules* are those that must be always enabled while *core optional modules* are those that can be either enabled or disabled. Additional modules can be classified into contributed modules and custom modules and can be optionally installed and enabled. *Contributed modules* are developed by the Drupal community and shared under the same GNU Public License (GPL) as Drupal. *Custom modules* are those created by external contributors. Figure 1 depicts some popular core and additional Drupal modules.

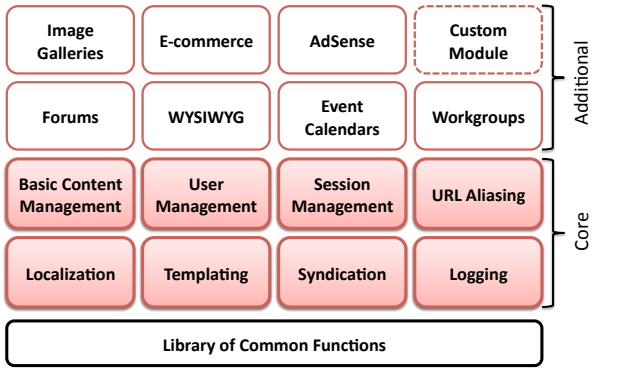


Fig. 1 Several Drupal core and additional modules. Taken from [65]

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 extends 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 2014-05-20
version = "7.x-3.8"
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 requi-

red 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*. The directive $required = TRUE$ is included in the core compulsory modules that must be always enabled.

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 tests. *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[1], these 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. Figure 2 shows a snapshot of SimpleTest while running the tests of Drupal v7.23 modules.

3 The Drupal feature model

In this section, we describe the process followed to model Drupal v7.23 variability using a feature model, depicted in Figure 3. Feature models are the *de–facto* standard for software variability modelling [8,36]. We selected this notation for its simplicity and its broad adoption in the field of variability testing.

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

Fig. 2 Running Drupal tests

3.1 Feature tree

According to the Drupal documentation, each module that is installed and enabled adds a new *feature* to the framework [65] (chapter 1, page 3). Thus, we propose modelling Drupal modules as features of the feature model. Figure 3 shows the Drupal features that were considered in our study, 48 in total including the root feature. In particular, among the 44 core modules of Drupal, we first selected the Drupal core modules that must be always enabled (i.e. core compulsory modules), 7 in total, e.g. *Node*. In Figure 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 selected 40 modules within the most installed Drupal core optional modules (e.g. *Path*) and additional modules (e.g. *Google Analytics*) ensuring that all dependencies were self-contained, i.e. all dependencies points at modules also included in our study. Most of these modules can be optionally installed and enabled and thus were modelled as optional features in the feature model. Exceptionally, the additional module *Date API* has a mandatory relation with its parent feature *Date*.

Submodules were mapped to subfeatures. Drupal submodules are those included in the directory of other modules. They provide extra functionality to its parent module and they have no meaning without it. As an example, the feature *Date* presents several subfeatures such as *Date API*, *Date popup* and *Date views*. Exceptionally, the submodules of *Node*, *Blog* and *Forum*, appear in separate module folders, however, the description of the modules in the Drupal documentation indicates that these modules are specializations of

Fig. 3 Drupal feature model

Node [65]. With respect to the set relationships *or* and *alternative*, typically found in feature models, none of them were identified among the features considered in Figure 3.

The selected modules are depicted in Table 6 in Appendix A. In total, we considered 16 Drupal core modules and 31 additional modules obtaining a feature model with 48 features, i.e. root feature + 8 mandatory features + 39 optional features.

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, 42 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*. We may remark that 21 out of the 42 cross–tree constraints identified were redundant when considered together with the feature relationships in the tree. For instance, the constraint *Forum* requires *Field* is unnecessary since *Field* is a core feature included in all the Drupal configurations. Similarly, the constraint *Date popup* requires *Date API* can be omitted since *Date API* has a mandatory relationship with their parent feature *Date*. We manually identified and removed all redundant cross–tree constraints. This makes a total of 21 requires cross–tree constraints shown in Figure 3. No excludes constraints were identified among the modules. Interestingly, we found that all modules in the same version of Drupal are expected to work fine together. If a Drupal module has incompatibilities with others, it is reported as a bug that must be fixed. As an example, consider the bug for Drupal 6 titled "Incompatible modules"[2].

As a sanity check, we confirmed the constraints identified using the Javascript InfoVis Toolkit (JIT) Drupal module, which shows a graphical representation of the modules and their relationships [17]. Figure 4 depicts a fragment of the dependency graph provided

[2] https://drupal.org/node/1312808

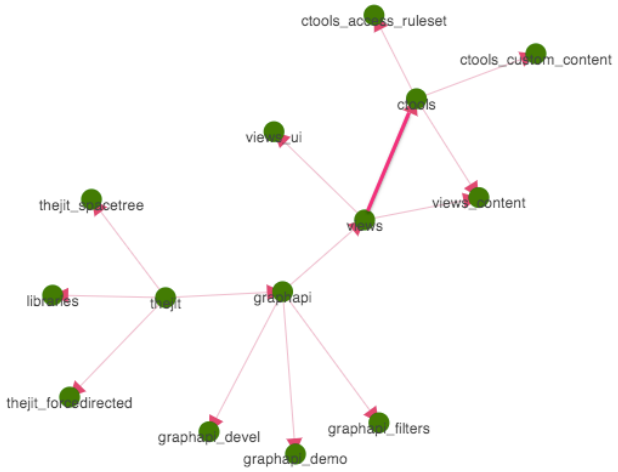


Fig. 4 Dependency graph generated by the JIT module

by the JIT module showing a dependency (i.e. directed edge) between *Views* (source node) and *Ctools* (target node). Therefore, we confirm the requires cross–tree constraint found in the *views.info* file presented in Listing 1.

The ratio of features involved in cross–tree constraints to the total number of features in the model (CTCR) is 45.8%. This metric provides a rough idea of the complexity of the model and enables comparisons. Hence, for instance, Drupal is more complex (in terms of CTCR) than the models in the SPLOT repository (Avg. CTCR = 16.1%) [44,60] and the models investigated by Bagheri et al [5] in their work about feature model complexity metrics (Avg. CTCR = 19.5%). Conversely, Drupal CTCR is less complex than the models reported by Berger et al. [10] in the context of operating systems (Avg. CTCR = 72.9%). This was expected since system software interacts with hardware in multiple ways and it is certainly more complex than Web applications. The Drupal feature model represents 2,090,833,920 configurations. In Section 12, we provide the Drupal feature model in two different formats (SXFM and FaMa).

4 Non–functional attributes

In this section, we report a number of non–functional attributes of the features presented in Figure 3 extracted from the Drupal web site and the Drupal Git repository. These data are often used as good indicators of the fault propensity of a software application. Additionally, this may provide researchers and practitioners with helpful information about the characteristics of features in a real VIS. By default, the information was extracted from the features in Drupal v7.23. For

the sake of readability, we used a tabular representation for feature attributes instead of including them in the feature model of Figure 3. Table 1 depicts the non–functional attributes collected for each Drupal feature, namely:

Feature size. This provides a rough idea of the complexity of each feature and its fault propensity [38, 43]. The size of a feature was calculated in terms of the number of Lines of Code (LoC). LoC were counted using the egrep and wc Linux commands 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 test files were excluded from the counting. The sizes range between 284 LoC (feature *Ctools custom content*) and 54,270 LoC (feature *Views*). It is noteworthy that subfeatures are significantly smaller than their respective parent features. The total size of the selected Drupal features is 336,025 LoC.

```
egrep -Rv '#|^$' name_module* other_file* | wc -l
```

Cyclomatic Complexity (CC). This metric reflects the total number of independent logic paths used in a program and provides a quantitative measure of its complexity [50,64]. We used the open source tool *phploc* [11] to compute the CC of the source code associated to each Drupal feature. Roughly speaking, the tool calculates the number of control flow statements (e.g. "if", "while") per lines of code [11]. The values for this metric ranged from 0.14 (feature *Path*) to 1.09 (feature *Entity token*). It is noteworthy that *Entity tokens*, the feature with highest CC, is one of the smallest features in terms of LoC (327).

Number of tests. Table 1 shows the total number of test cases and test assertions of each feature, obtained from the output of the *SimpleTest* module. In total, Drupal features include 352 test cases and 24,152 assertions. In features as *Ctools*, the number of test cases (7) and test assertions (121) is low considering that the size of the feature is over 17,000 LoC. It is also noteworthy that features such as *JQuery update*, with more than 50,000 LoC, have no test cases.

Number of reported installations. This depicts the number of times that a Drupal feature has been installed as reported by Drupal users. This data was extracted from the Drupal web site [12] and could be used as an indicator of the popularity or impact of a feature. Notice that the number of reported installations of parent features and their respective subfeatures is equal since they are always installed together, although they may be optionally enabled or disabled. Not surprisingly, the features with the highest number of reported installations are those included in the Drupal core (5,259,525 times) followed by *Views* (802,467 times) and *Ctools* (747,248 times), two of the most popular features in Drupal.

Number of developers. We collected the number of developers involved in the development of each Drupal feature. This could give us information about the scale and relevance of the feature as well as its propensity to faults related to the number of people working on it [43]. This information was obtained from the web site of each Drupal module as the number of committers involved [12]. The feature with the highest number of contributors is *Views* (178), followed by those included in the Drupal core (94) and *Ctools* (75).

Number of changes. Changes in the code are likely to introduce faults [28,68]. 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 tracked the commits to the Drupal Git repository[3]. The search was narrowed by focusing on the changes performed during a period of two years, from May 1^{st} 2012 to April 31^{st} 2014. 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. We collected the number of changes in Drupal v7.22 to check the correlation with the number of faults in Drupal v7.23 (see Section 6). As illustrated in Table 1, the number of changes ranged between 0 (feature *Blog*) and 90 (feature *Backup and migrate*). Interestingly, the eight features with the highest number of changes are optional. In total, we counted 557 changes in Drupal v7.22 during a two–years period.

```
git log --pretty=oneline --after={2012-05-01}
--before={2014-04-31} 7.21..7.22 name_module | wc -l
```

5 Faults in Drupal

In this section, we present the number of faults reported in the Drupal features (i.e. modules) shown in Figure 3. The information was obtained from the issue tracking systems of Drupal[4] and related modules. In particular, we used the web–based search tool of the issue systems to filter the bug reports by severity, status, date, feature name and Drupal version. The search was

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

Feature	Size	CC	Test cases	Test assertions	Installations	Developers	Changes v7.22
Backup and migrate	11,639	0.37	0	0	281,797	7	90
Blog	551	0.16	1	244	5,259,525	94	0
Captcha	3,115	0.19	4	731	226,295	43	15
CKEditor	13,483	0.59	0	0	280,919	29	40
Comment	5,627	0.23	14	3,287	5,259,525	94	2
Ctools	17,572	0.52	7	121	747,248	75	32
Ctools access ruleset	317	0.19	0	0	747,248	75	0
Ctools custom content	284	0.3	0	0	747,248	75	1
Date	2,696	0.44	4	1,724	412,324	42	9
Date API	6,312	0.6	1	106	412,324	42	11
Date popup	792	0.36	0	0	412,324	42	4
Date views	2,383	0.44	0	0	412,324	42	6
Entity API	13,088	0.41	11	851	407,569	45	14
Entity tokens	327	1.09	1	6	407,569	45	1
Features	8,483	0.56	3	16	209,653	36	72
Field	8,618	0.41	9	870	5,259,525	94	6
Field SQL storage	1,292	0.3	1	94	5,259,525	94	1
Field UI	2,996	0.28	3	287	5,259,525	94	4
File	1,894	0.67	39	2,293	5,259,525	94	1
Filter	4,497	0.17	9	958	5,259,525	94	1
Forum	2,849	0.24	2	677	5,259,525	94	3
Google analytics	2,274	0.29	4	200	348,278	21	14
Image	5,027	0.29	2	677	5,259,525	94	9
Image captcha	998	0.28	0	0	226,295	43	0
IMCE	3,940	0.47	0	0	392,705	13	9
Jquery update	50,762	0.26	0	0	286,556	17	1
Libraries API	1,627	0.55	2	135	516,333	7	7
Link	1,934	0.63	8	1,275	286,892	31	11
Node	9,945	0.27	32	1,391	5,259,525	94	9
Options	898	0.17	2	227	5,259,525	94	0
Panel nodes	480	0.35	0	0	206,805	43	2
Panels	13,390	0.35	0	0	206,805	43	34
Panels In-Place Editor	1,462	0.23	0	0	206,805	43	20
Path	1,026	0.14	5	330	5,259,525	94	0
PathAuto	3,429	0.23	5	316	622,478	33	2
Rules	13,830	0.49	5	285	238,388	52	5
Rules scheduler	1,271	0.15	1	7	238,388	52	4
Rules UI	3,306	0.39	0	0	238,388	52	1
System	20,827	0.31	58	2,138	5,259,525	94	19
Taxonomy	5,757	0.23	14	677	5,259,525	94	2
Text	1,097	0.29	3	444	5,259,525	94	0
Token	4,580	0.51	15	347	715,563	31	10
User	8,419	0.26	23	1,355	5,259,525	94	7
Views	54,270	0.41	51	1,089	802,467	178	27
Views content	2,683	0.46	0	0	747,248	75	5
Views UI	782	0.37	9	538	802,467	178	0
WebForm	13,196	0.51	4	456	402,163	46	46
Total	**336,025**	**17.41**	**352**	**24,152**	**97,342,266**	**3,060**	**557**

Table 1 Non–functional feature attributes in Drupal

Feature	Faults in Drupal v7.22						Faults in Drupal v7.23					
	Severity				Total	Total	Severity				Total	Total
	Minor	Normal	Major	Critical	Single	Integ	Minor	Normal	Major	Critical	Single	Integ
Backup migrate	8	58	9	9	80	4	8	58	9	9	80	4
Blog	0	2	2	0	1	3	0	1	2	0	0	3
Captcha	1	14	3	0	17	1	1	14	3	0	17	1
CKEditor	6	165	29	8	197	11	6	163	29	8	197	9
Comment	2	20	5	2	10	19	3	16	5	4	13	15
Ctools	17	146	39	10	181	31	17	146	39	10	181	31
Ctools access r.	0	0	0	0	0	0	0	0	0	0	0	0
Ctools custom c.	2	7	2	0	10	1	2	7	2	0	10	1
Date	4	30	12	1	44	3	4	30	12	1	44	3
Date API	3	29	9	1	41	1	3	29	9	1	41	1
Date popup	2	28	1	0	30	1	2	28	1	0	30	1
Date views	1	18	7	0	25	1	1	18	7	0	25	1
Entity API	9	128	43	13	175	18	9	128	43	13	175	18
Entity Tokens	0	19	8	1	22	6	0	19	8	1	22	6
Features	3	81	17	5	97	9	3	81	17	5	97	9
Field	6	43	12	2	45	18	7	45	11	2	48	17
Field SQL s.	0	5	0	0	3	2	0	5	0	0	3	2
Field UI	6	9	0	0	13	2	6	6	0	0	11	1
File	1	8	5	1	10	5	1	9	5	1	11	5
Filter	3	19	0	2	19	5	3	19	0	2	19	5
Forum	0	6	4	0	6	4	0	6	3	0	5	4
Google anal.	0	8	2	2	11	1	0	8	2	2	11	1
Image	1	10	6	1	10	8	1	9	4	1	9	6
Image captcha	0	3	0	0	3	0	0	3	0	0	3	0
IMCE	0	12	1	1	9	5	0	12	1	1	9	5
Jquery update	4	48	14	10	64	12	4	48	14	10	64	12
Libraries API	1	6	3	1	11	0	1	6	3	1	11	0
Link	6	68	9	3	82	4	6	68	9	3	82	4
Node	8	33	7	7	26	29	10	26	5	6	24	23
Options	0	0	0	0	0	0	0	0	0	0	0	0
Panel Nodes	1	13	2	1	16	1	1	13	2	1	16	1
Panels	5	92	9	5	87	24	5	92	9	5	87	24
Panels IPE	1	18	1	1	19	2	1	18	1	1	19	2
Path	0	3	0	1	3	1	0	2	0	1	2	1
PathAuto	4	33	18	8	54	9	4	33	18	8	54	9
Rules	5	180	54	16	240	15	5	180	54	16	240	15
Rules sched.	0	11	2	0	13	0	0	11	2	0	13	0
Rules UI	0	20	3	3	26	0	0	20	3	3	26	0
System	7	28	4	1	35	5	7	27	4	1	35	4
Taxonomy	0	27	6	4	15	22	0	31	6	4	19	22
Text	0	9	0	0	6	3	0	8	0	0	5	3
Token	6	20	15	3	37	7	6	20	15	3	37	7
User	3	36	6	0	20	25	3	32	6	0	19	22
Views	70	807	205	60	1,091	51	70	807	205	60	1,091	51
Views content	1	23	0	1	23	2	1	23	0	1	23	2
Views UI	1	15	0	0	12	4	1	15	0	0	12	4
WebForm	47	231	10	4	292	0	47	231	10	4	292	0
Total					**3,231**						**3,232**	

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

narrowed by collecting the bugs reported in a period of two years, from May 1^{st} 2012 to April 31^{st} 2014. We collected the faults of two consecutive Drupal versions, v7.22 and v7.23, to achieve a better understanding of the evolution of a real system and to enable test validations based on fault–history (see Section 7).

First, we filtered the faults by feature name (using the field "component"), framework version and the dates previously mentioned. Then, the search was refined to eliminate the faults not accepted by the Drupal community, those classified as *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. A total of 3,401 faults matched the initial search for Drupal v7.22 and 3,392 faults for Drupal v7.23.

Second, we tried to identify those faults that were caused by the interaction of several features, i.e. integration faults. Our first approach was to follow the strategy presented by Artho et al. [3] in the context of operating systems. That is, for each feature, we searched for faults descriptions containing the keywords "break", "conflict" or "overwrite". However, the search did not return any result related to the interaction among features. We hypothesize that keywords related to potential bugs caused by the interaction among features may be domain–dependent. Thus, we followed a different approach. For each feature, we searched for bug reports that included the name (not case–sensitive) of some of the other features under study in its description or tags. As an example, consider the bug report depicted in Listing 2 extracted from the Drupal issue tracking system. The bug is associated to the module *Rules* and its description contains the name of the feature *Comment*. Thus, we considered it as a candidate integration fault between both features, *Rules* and *Comment*. In total, we detected 444 candidate integration faults in Drupal v7.22 and 434 in Drupal v7.23 following this approach. Interestingly, we found candidate faults containing the name of up to 9 different features.

Listing 2 Drupal bug report (http://www.drupal.org/node/1673836)

```
Adding rule for comments causes:Fatal error: Call to undefined
function comment_node_url() in...bartik/template.php on line 164

I added rule that would react on an event `A comment is viewed
by showing up a message for every new comment added... and it
broke every page on which I added new comment (It was all
working fine before this rule was introduced, commenting also).
For all other pages where there are old comments everything is
quite fine. Tried to deleted rule, to disable rules module,
cleared cache, run cron... nothing changed. The only thing that
gets back my pages without this fatal error is if I disable
comments module, but I need comments.
```

Next, we manually checked the bug reports of each candidate integration fault and discarded those that *i*) included the name of a feature but it actually made no reference to the feature (e.g. image), *ii*) included the name of Drupal features not considered in our study, and *iii*) included users and developers' comments suggesting that the fault was not caused by the interaction among features. We may remark that in many cases the description of the fault clearly suggested an integration bug, see as an example the last sentence in Listing 2: "*The only thing that gets back my pages without this fatal error is if I disable comments module*". The final counting revealed 160 integration faults in Drupal v7.23 and 170 in Drupal v7.22. In Drupal v7.23, we found that 3 out of 160 faults were caused by the interaction of 4 features, 25 were caused by the interaction of 3 features and 132 faults were triggered by the interaction between 2 features. It is noteworthy that 51 out of the 160 integration faults were caused by the interaction of *Views* with other features. *Views* enables the configuration of all the views of Drupal sites. Also, 31 integration faults were triggered by the interaction of *Ctools* with other features. *Ctools* provides a set of APIs and tools for modules to improve the developer experience. A complete list of the integration faults detected and the features involved on them is presented in Table 7 in Appendix A. It is noteworthy that we did not find any specific keywords in the bugs' descriptions suggesting that they were caused by the interaction among features.

Table 2 summarizes the faults found in both versions of Drupal. For each feature, the total number of individual and integration faults in which it is involved are presented classifying them according to the reported severity level. We found that the bugs reported in additional Drupal modules (e.g. *Ctools*) do not discriminate among subversions of Drupal 7, i.e. they specify that the fault affects to Drupal v7.x. In those cases, we assumed that the bug equally affected to the two versions of Drupal under study, v7.22 and v7.23. This explains why the number of faults in most of the additional features is the same for both versions of the framework in Table 2. Notice, however, that in some cases the number of integration faults differs, e.g. feature *CKEditor*.

6 Correlation study

The information extracted from software repositories can be used, among other purposes, to drive the search for faults. In fact, multiple works in the field of soft-

ware repository mining have explored the correlations between certain non–functional properties and metrics and the fault propensity of software components [3,30,33,46,49]. In this section, we investigate whether the non–functional attributes presented in Section 4 could be used to estimate the fault propensity of Drupal features.

The correlation study was performed using the R statistical environment [51] in two steps. First, we checked the normality of the data using the Shapiro-Wilk test concluding that the data do not follow a normal distribution. Second, we used the Spearman's rank order coefficient to assess the relationship between the variables. We selected Spearman's correlation because it does not assume normality, it is not very sensitive to outliers, and it measures the strength of association between two variables under a monotonic relationship[5], which fits well with the purpose of this study.

The results of the correlation study are presented in Table 3. For each correlation under study the following data are shown: identifier, measured variables, Spearman's correlation coefficient (ρ) and p-value. The Spearman's correlation coefficient takes a value in the range [-1,1]. A value of zero indicates that no association exists between the measured variables. Positive values of the coefficient reflect a positive relationship between the variables, i.e. as one variable decreases, the other variable also decreases and vice versa. Conversely, negatives values of the coefficient provide evidence of a negative correlation between the variables, i.e. the value of one variable increases as the value of the other decreases. Roughly speaking, the p–value represents the probability that the coefficient obtained could be the result of mere chance, assuming that the correlation does not exist. It is commonly assumed that p–values under 0.05 are enough evidence to conclude that the variables are actually correlated and that the estimated ρ is a good indicator of the strength of such relationship, i.e. the result is statistically significant. The correlations that revealed statistically significant results are highlighted in grey in Table 3. By default, all correlations were investigated in Drupal v7.23 except several exceptions explicitly mentioned. The results of the correlation study are next discussed.

C1. Correlation between feature size and faults. The Spearman coefficient reveals a strong positive correlation (ρ = 0.78) between the LoC and the number of faults in Drupal features. This means that the number of LoC could be used as a good estimation of their fault propensity. In our study, we found that 7 out of the 10 largest features were also among the 10 features with a higher number of faults. Conversely, 5 out of the 10 smallest features were among the 6 features with a lower number of bugs.

C2. Correlation between changes and faults. We studied the correlation between the number of changes in Drupal v7.22 and the number of faults in Drupal v7.23, obtaining a Spearman coefficient of 0.68. This means that the features with a higher number of changes are likely to have a higher number of bugs in the subsequent release of the framework. In our study, we found that 7 out of the 10 features with the highest number of changes in Drupal v7.22 were also among the 10 features with the highest number of faults in Drupal v7.23.

C3. Correlation between faults in consecutive versions of the framework. We investigated the correlation between the number of reported faults in Drupal features in two consecutive versions of the framework, v7.22 and v7.23. As mentioned in Section 5, the bugs reported in Drupal additional modules are not related to a specific subversion of the framework (they just indicate 7.x). Thus, in order to avoid biased results, we studied this correlation on the features of the Drupal core (16 out of 47) where the specific version of Drupal affected by the bugs is provided. We obtained a Spearman's correlation coefficient of 0.98 reflecting a very strong correlation between the number of faults in consecutive versions of Drupal features. This provides helpful information about the across–release fault propensity of features and it could certainly be useful to drive testing decisions when moving into a new version of the framework, e.g. test case prioritization.

C4. Correlation between code complexity and faults. We studied the correlation between the cyclomatic complexity of Drupal features and the number of faults detected on them. As a result, we obtained a correlation coefficient of 0.51, which reflect a fair correlation between both parameters, i.e. features with a high cyclomatic complexity tend to have a high number of faults. It is noteworthy, however, that this correlation is weaker than the correlation observed between the number of LoC and the number of faults (C1) with a coefficient of 0.78. Despite this, we believe that the cyclomatic complexity could still be a helpful indicator to estimate the fault propensity of features with simi-

[5] A monotonic relationship implies that as the value of one variable increases, so does the value of the other variable; or as the value of one variable increases, the other variable value decreases.

Id	Variables		Correlation (ρ)	p–value
	Variable 1	Variable 2		
C1	LoC v7.23	Faults v7.23	0.78	4.16×10^{-11}
C2	Changes v7.22	Faults v7.23	0.68	8.36×10^{-8}
C3	Faults v7.22	Faults v7.23	0.98	3.26×10^{-13}
C4	CC v7.23	Faults v7.23	0.51	2.32×10^{-4}
C5	Developers v7.23	Faults v7.23	-0.27	0.066
C6	Assertions v7.23	Faults v7.23	0.16	0.268
C7	CTC v7.23	Faults v7.23	0.11	0.449
C8	CTC v7.23	Int. Faults v7.23	0.21	0.150
C9	CoC v7.23	Faults v7.23	0.28	0.051

Table 3 Spearman correlation results

lar size.

C5. Correlation between faults and number of developers. We investigated the correlation between the number of developers contributing to a Drupal feature and the number of bugs reported in that feature. We hypothesized that a large number of developers could result in coordination problems and a higher number of faults. Surprisingly, we obtained a negative correlation value (-0.27), which not only rejects our hypothesis but it suggests the opposite, those features with a higher number of developers, usually the Drupal core features, have less faults. 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. Nevertheless, we obtained a p–value over 0.05 and therefore this relationship is not statistically significant.

C6. Correlation between tests and faults. It could be expected that those features with a high number of test assertions contain a low number of faults since they are tested more exhaustively. We investigated this correlation obtaining a Spearman's coefficient of 0.16 and a p–value of 0.268. Thus, we cannot conclude that those features with a higher number of test assertions are less error–prone.

C7-8. Correlation between faults and Cross–Tree Constraints (CTCs). In [21], 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 CTCs in order to study the relation between them and their fault propensity. We obtained a correlation coefficient of 0.11 between the number of CTCs in which a feature is involved and the number of faults in that feature (C7). The correlation coefficient rose to 0.21 when considering integration faults only (C8). Therefore, we conclude that the correlation between feature involvement in CTCs and fault propensity is not confirmed in our study.

C9. Correlation between faults and Coefficient of Connectivity–Density (CoC). The CoC is a feature model complexity metric proposed by Bagheri et al. [5] and adapted by the authors for its use at the feature level [54]. The CoC measures the complexity of a feature as the number of tree edges and cross–tree constraints connected to it. We studied the correlation between the complexity of features in terms of CoC and the number of faults detected on them. The correlation study revealed a Spearman's coefficient of 0.28. Therefore, we cannot conclude that those features with a higher CoC are more error–prone. Nevertheless, we obtained a p-value very close to the threshold of statistical significance (0.051), therefore this relationship is a good candidate to be further investigated in future studies of different VISs.

Regarding feature types, we identified 326 faults in the mandatory features of the Drupal feature model and 3,540 faults in the optional features. Mandatory features have a lower ratio of faults per feature (326/8 = 40.7) than optional ones (3,540/39 = 90.7). We presume that this is due to the fact that 7 out of the 8 mandatory features represent core compulsory modules included in all Drupal configurations and thus they are better supported and more stable. Regarding fault severity, around 71.6% of the faults were classified as normal, 16.1% as major, 6.9% as minor and 5.2% as critical. We observed no apparent correlation between fault severity and the types of features and thus we did not investigate it further.

7 Evaluation

In this section, we evaluate the feasibility of the Drupal case study as a motivating experimental subject to evaluate variability testing techniques. In particular, we present an experiment to answer the following research questions:

RQ1: *Are non–functional attributes helpful to accelerate the detection of faults of combinatorial test suites?* The correlations found between the non–functional attributes of Drupal features and their fault propensity suggest that non–functional information could be helpful to drive the search for faults. In this experiment, we measure the gains in the rate of fault detection when using non–functional attributes to prioritize combinatorial test suites.

RQ2: *Are non–functional attributes more, less or equally effective than functional data at accelerating the detection of faults?* Some related works have proposed using functional information from the feature model to drive test case prioritization such as configuration similarity, commonality or variability coverage [21, 31, 54]. In this experiment, we measure whether non–functional attributes are more, less or equally effective than functional data at accelerating the detection of faults.

We next describe the experimental setup, the prioritization criteria compared, the evaluation metric used and the results of the experiment.

7.1 Experimental setup

The goal of this experiment is to use a combinatorial algorithm (e.g. ICPL) to generate a pair–wise suite for the Drupal feature model, and then, use different criteria based on functional and non–functional information to derive an ordering of test cases that allows detecting faults as soon as possible. The evaluation was performed in several steps. First, we seeded the feature model with the faults detected in Drupal v7.23, 3,392 in total. For this purpose, we created a list of faulty feature sets. Each set represents faults triggered by n features ($n \in [1,4]$). For instance, the list {{Node}{Ctools, User}} represents a fault in the feature *Node* and another fault caused by the interaction between the features *Ctools* and *User*. Second, we used the ICPL algorithm [34] to generate a pairwise suite for the Drupal feature model. We define a test case in this domain as a valid Drupal configuration, i.e. valid set of features. Among the whole testing space of Drupal (more than 2,000 millions of test cases), the ICPL algorithm returned 13 test cases 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 bug. The pairwise test suite detected all the faults.

Next, we reordered the test cases in the pairwise suite according to different prioritization criteria. In order to answer RQ1, we prioritized the suite using the non–functional attributes that correlated well with the fault propensity, namely, the feature size, number of changes and number of faults in the previous version of the framework, i.e. Drupal v7.22. To answer RQ2, we reordered the test cases using two of the functional prioritization criteria that have provided better results in related evaluations, the similarity and VC&CC metrics. These prioritization criteria are fully described in Section 7.1.1. For each prioritized test suite, we measured how fast the faults of Drupal 7.23 were detected by each ordering calculating their Average Percentage of Faults Detected (APFD) values (see Section 7.1.2).

Finally, we repeated the whole process using the CASA algorithm [27] for the generation of the pairwise test suite for more heterogeneous results. Since CASA is not deterministic, we ran the algorithm 10 times for each prioritization criterion and calculated averages. The suites generated detected all the faults.

7.1.1 Prioritization criteria

Given a test case t composed of a set of features, $t = \{f_1, f_2, f_3 ... f_n\}$, we propose the following criteria to measure its priority:

Size–driven. This criterion measures the priority of a test case as the sum of the LoC of its features. Let $loc(f)$ be a function returning the number of LoC of feature f. The Priority Value (PV) of t is calculated as follows:

$$PV_{Size}(t) = \sum_{i=1}^{|t|} loc(fi) \quad (1)$$

Fault–driven. This criterion measures the priority value of the test case as the sum of the number of bugs detected on its features. Inspired by the correlations described in Section 6, we propose counting the number of faults in Drupal v7.22 to effectively search for faults in Drupal v7.23. Let $faults(f, v)$ be the function returning the number of bugs in version v of feature f. The priority of t is calculated as follows:

$$PV_{Faults}(t) = \sum_{i=1}^{|t|} faults(fi, "7.22") \quad (2)$$

Change–driven. This criterion calculates the priority of a test case by summing up the number of changes found on its features. As with the number of faults, we propose a history–based approach and use the number of changes in Drupal v7.22. Consider the function $changes(f, v)$ returning the number of changes in version v of feature f. The priority of t is calculated as follows:

$$PV_{Changes}(t) = \sum_{i=1}^{|t|} changes(fi, "7.22") \quad (3)$$

Variability Coverage and Cyclomatic complexity (VC&CC). This criterion aims to get an acceptable trade off between fault coverage and feature coverage. It was presented by Bagheri et al. in the context of SPL test case selection [21] and later adapted by the authors for SPL test case prioritization [54]. In [54] , the authors compared several prioritization criteria for SPLs and found that VC&CC ranked first at accelerating the detection of faults. This motivated the selection of this prioritization criterion as the best of its breed. Given a feature model fm and a test case t, the VC&CC metric is calculated as follows:

$$PV_{VC\&CC}(t, fm) = \sqrt{vc(t, fm)^2 + cc(t, fm)^2} \quad (4)$$

where $vc(t, fm)$ calculates the variability coverage of a test case t for the feature model fm. The variability coverage of a test case is the number of variation points involved in it. A variation point is any feature that provides different variants to create a configuration. $cc(t, fm)$ represents the cyclomatic complexity of t. The cyclomatic complexity of a test case is calculated as the number of cross–tree constraints involved in it. We refer the reader to [54] for more details about this prioritization criterion.

Dissimilarity. The (dis)similarity metric has been proposed by several authors as an effective prioritization criterion to maximize the diversity of a test suite [31, 54]. Roughly speaking, this criterion gives a higher priority to those test cases with fewer features in common since they are likely to get a higher feature coverage than similar test cases. In [54], the authors proposed to prioritize the test cases based on the dissimilarity metric using the Jaccard distance to measure the similarity among test cases. The Jaccard distance [63] is defined as the size of the intersection divided by the size of the union of the sample sets. In our context, each set represents a test case containing a set of features. The prioritized test suite is created by progressively adding the pairs of test cases with the highest Jaccard distance between them until all test cases are included. Given two test cases t_a and t_b, the distance between them is calculated as follows:

$$Dissimilarity(t_a, t_b) = 1 - \frac{|t_a \bigcap t_b|}{|t_a \bigcup t_b|} \quad (5)$$

The resulting distance varies between 0 and 1, where 0 denotes that the test cases t_a and t_b are the same and 1 indicates that t_a and t_b share no features.

7.1.2 Evaluation metric

In order to evaluate how quickly faults are detected during testing we used the *Average Percentage of Faults Detected (APFD)* metric [18,19,52]. The APFD metric measures the weighted average of the percentage of faults detected during the execution of the test suite. 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. According to [19], the APFD metric for the test suite T' is given by the following equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_n}{n \times m} + \frac{1}{2n}$$

APFD values are in the range (0,1). The closest the value is to 1, the fastest is the suite at detecting faults. For example, consider a test suite of 4 test cases (T1-T4) and 5 faults (F1-F5) detected by those test cases, as shown in Table 4. Consider two orderings of these test cases, ordering O1: T1,T2,T3,T4 and ordering O2: T3,T2,T4,T1. According to the previous APFD equation, ordering O1 produces an APFD of 58% :

$$1 - \frac{1+1+2+3+4}{4 \times 5} + \frac{1}{2 \times 4} = 0.58$$

and ordering O2 an APFD of 78% :

$$1 - \frac{1+1+1+1+3}{4 \times 5} + \frac{1}{2 \times 4} = 0.78,$$

being O2 much faster detecting faults than O1.

Tests/Faults	F1	F2	F3	F4	F5
T1	X	X			
T2	X		X		
T3	X	X	X	X	
T4					X

Table 4 Test suite and faults exposed

7.2 Experimental results

Table 5 depicts the results of the experiment. For each combinatorial testing algorithm, ICPL and CASA, the table shows the APFD values of the pairwise test suites prioritized according to the criteria presented in previous section. The first row shows the APFD of the suites when no prioritization is applied. The top three highest APFD values of each column are highlighted in bold. For ICPL, the prioritized suites based on non–functional attributes revealed the best results with fault–driven prioritization ahead (95.5%), followed by the size–driven (95.4%) and change–driven (95%) prioritization criteria. For CASA, the best APFD value was again obtained by the fault–driven prioritization criterion (93.4%), followed by the VC&CC (92.8%), size–driven (92.7%) and change–driven (91.8%) criteria. Overall, prioritization driven by non–functional attributes revealed the best rates of early fault detection followed by the functional prioritization criteria VC&CC and dissimilarity. Not surprisingly, all prioritization criteria accelerated the detection of faults of the unprioritized suites. It is noteworthy that the APFD values of the suites generated with CASA were lower than those of ICPL in all cases. We presume this is due to the internal ordering implemented as a part of the ICPL algorithm [34].

Prioritization criterion	ICPL	CASA
None	87.7	87.4
Size–driven	**95.4**	**92.7**
Fault–driven	**95.5**	**93.4**
Change–driven	**95.0**	91.8
VC&CC	93.5	**92.8**
Dissimilarity	92.7	87.9

Table 5 APFD values of the prioritized test suites

Figure 5 shows the percentage of detected faults versus the fraction of the ICPL prioritized test suites. Roughly speaking, the graphs show how the APFD value evolves as the test suite is exercised. Interestingly, all three non–functional prioritization criteria revealed 92% of the faults (3,120 out of 3,392) by exercising just 8% of their respective suites, i.e. 1 test case out of 13. In contrast, the original suite (unprioritized) detected just 5% of the faults (169 out of 3,392) with the same number of test cases. The fault-driven prioritized suite was the fastest suite in detecting all the faults (3,392) by using just 24% of the test cases (3 out of 13). All other orderings required 31% of the suite (4 out 13) to detect all the faults with the exception of the dissimilarity criterion, which required exercising 47% of the test cases (6 out of 13).

Based on the result obtained, we can answer to the research questions as follows:

Response to RQ1. The results show that non–functional attributes are effective drivers to accelerate the detection of faults and thus the response is "*Yes, non–functional attributes are helpful to accelerate the detection of faults in VISs*".

Response to RQ2. The prioritization driven by non–functional attributes led to faster fault detection than functional criteria in all cases with the only exception of the criterion VC&CC, which ranked second for the CASA test suite. Therefore, the response derived from our study is "*Non–functional attributes are more effective than functional data at accelerating the detection of faults in most of the cases*".

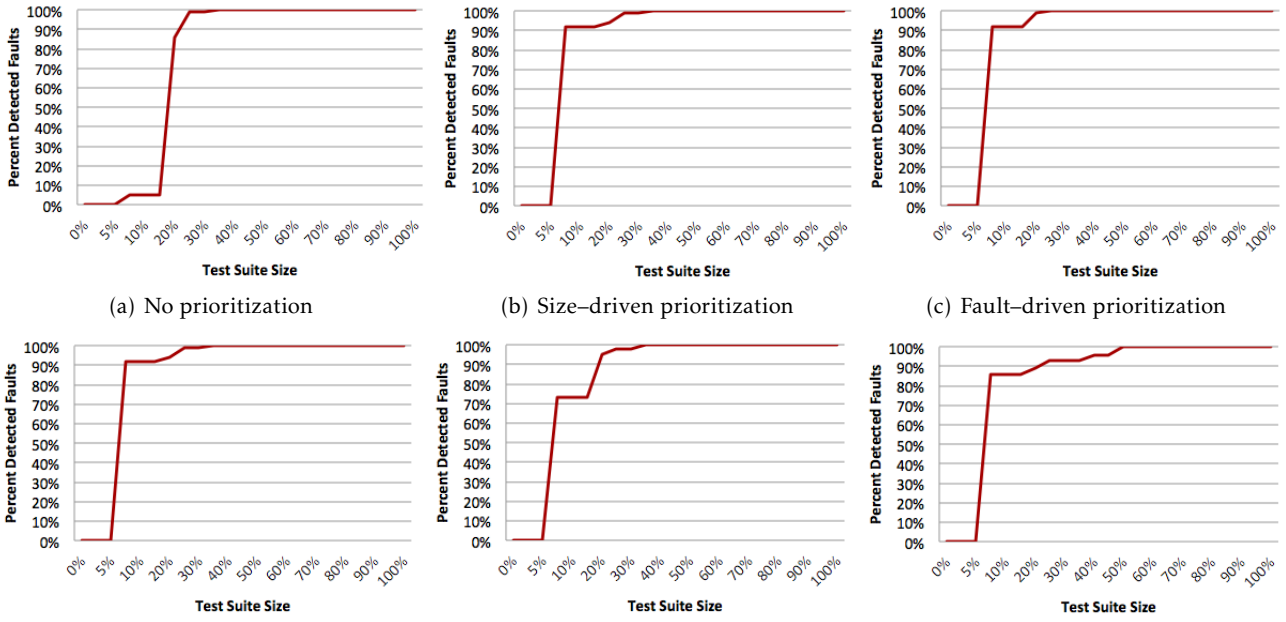
8 Applicability to other VISs

Based on our experience, we believe that the approach proposed throughout this article could be applicable to model the variability of other open-source VISs. For that purpose, we have identified a number of basic requirements that the VIS under study should fulfill and that could be helpful for researchers interested in following our steps, namely:

Identification of features. The system should be composed of units such as modules or plug-ins that can be easily related to features of the VIS.

Explicit variability constraints. The system should provide information about the constraints among the features of the VIS, either explicitly in configuration files or as a part of the documentation of the system.

Feature information. The system under study should provide extensive and updated information about its features. This may include data as the number of downloads, reported installations, test cases, number of de-



(a) No prioritization (b) Size–driven prioritization (c) Fault–driven prioritization

(d) Change–driven prioritization (e) VC&CC (f) Dissimilarity

Fig. 5 Percentage of detected faults versus the fraction of the exercise suites

velopers, etc.

Bug tracking system. The VIS should have a bug tracking system highly used by its community of users and frequently updated. It is desirable that the developers follow a rigorous bug review process updating the fields related to version, bug status and severity. Also, it is crucial that bugs are related to the features in which they were found using a standardized procedure, e.g. using labels.

Version Control System (VCS). The system should use a VCS that can be easily queried to get information about the number of commits by feature, date and version.

9 Threats to validity

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

Internal validity. This refers to whether there is sufficient evidence to support the conclusions and the sources of bias that could compromise those conclusions. The re-engineering process could have influenced the final feature model of Drupal and therefore the evaluation results. To alleviate this threat, we followed a systematic approach and mapped Drupal modules to features. This is in line with the Drupal documentation, which defines an enabled module as a feature providing certain functionality to the system [65]. This also fits in the definition of feature given by Batory, who defines a feature as an increment in product functionality [7]. In turn, submodules were mapped to subfeatures since they provide extra functionality to its parent module and they have no meaning without it. Finally, we used the dependencies defined in the information file of each Drupal module to model CTCs.

Other risk for the internal validity of our work is the approach followed to collect the data about integration faults in Drupal, which mainly relies on the bug report description. It is possible that we missed some integration faults and, conversely, we could have misclassified some individual faults as integration faults. To mitigate this threat as much as possible we manually checked each candidate integration fault trying to discard those that were clearly not caused by the interaction among features. This was an extremely time–consuming and challenging task that required a good knowledge of the framework. We may emphasize that the main author of the article has more than one year of experience in industry as a Drupal developer. Also, as a further validation, the work was discussed with two members of the Drupal core team who approved the followed approach and gave us helpful feedback.

As previously mentioned, the faults in additional modules are not related to a specific Drupal subversion, i.e. they are reported as faults in Drupal v7.x. Therefore, we assumed that the faults in those modules equally affected the versions 7.22 and 7.23 of Drupal. This is a realistic approach since it is common in open source projects that unfixed faults affect to several versions of the system. However, this may introduce a bias in the fault–driven prioritization since several of the faults in Drupal v7.22 remained in Drupal v7.23. To minimize this threat, we excluded Drupal additional modules from the correlation study between the number of faults in Drupal v7.22 and Drupal v7.23, where a very strong correlation was revealed (0.98). It is also worth mentioning that the size–driven and change–driven prioritization criteria ranked 2nd and 3rd for ICPL and 3rd and 4rd for CASA, which still shows the efficacy of non–functional attributes at driving the search for faults.

External validity. This can be mainly divided into limitations of the approach and generalizability of the conclusions. Regarding the limitations, we may mention that Drupal modules have their own versioning system, i.e. there may exists different versions of the same feature (e.g. Views 7.x-3.8). We found, however, that bug reports in Drupal rarely include information about the version of the faulty modules and thus we did not keep track of modules' versions in our work. Although this may slightly affect the realism of the case study, it still provides a fair vision of the number and distribution of faults in a real feature–based VIS.

The correlations and prioritization results reported are based on a single case study and thus cannot be generalized to other VIS. However, based on our experience, we believe that the described re–engineering process could be applicable to other open–source plug–in and module–based systems such as Wordpress or Prestashop, recently used in variability-related papers [47,57]. We admit, however, that the described process could not be applicable to other domains with poorly documented variability. Despite this, our work does confirm the results of related works in software repository mining showing that can be found correlation between non–functional data and the fault propensity of software components. Similarly, our results show the efficacy of using non–functional attributes as driver for test case prioritization in VISs.

10 Related work

Related work on variability testing mainly addresses the problems of test case selection [4,6,16,21,32,37,39,40,42,48,55,66] and test case prioritization [2,6,15,20,22,31,35,45,54,61,67]. Most approaches use functional information to drive testing such as those based on combinatorial testing [2,22,31,32,35,37,39,40,42,48,54,67,61,66], similarity [2,31,54] or other metrics extracted from the feature model [16,21,32,54,55]. Several works have also explored the use of non–functional properties during testing such as user preferences and cost [6,14,15,20,22,23,32,35,55,61,66,67]. The lack of realistic case studies often lead researchers to evaluate their approaches using synthetic variability models [2,4,20,21,22,32,37,39,54,55], faults [2,15,16,21,54] and non–functional attributes [6,20,22,32,55], which introduce threats to validity and weaken their contributions. Our work complements related approaches by providing a realistic experimental subject composed of a feature model and a full collection of non–functional feature attributes including the number and distribution of faults. We also present a novel correlation study pointing at the non–functional attributes that could drive the search for faults in VISs more effectively. Finally, we present novel results on the use of real non–functional properties to drive test case prioritization in an open source VIS. To the best of our knowledge, this is the first work comparing the effectiveness of functional and non–functional prioritization criteria to accelerate the detection of bugs.

Some other authors have explored variability in the open source community before us. In [9,10,41,58], the authors studied several programs in the operating systems domain from a variability modelling perspective. Galindo et al. [24] explored variability modelling and analysis in the Debian operating system. Johansen et al. [35] modelled the variability in the Eclipse framework and used the number of downloads of its plugins (i.e. features) to evaluate a weight–based combinatorial testing algorithm. Nguyen et al. [47] proposed a variability–aware test execution approach and evaluated it using the WordPress blogging Web application. In [57], the authors reported their experience on variability analysis and testing on the development of an online store using the Prestashop e–commerce platform. Inspired by previous works, we explore the open source Drupal framework as a motivating VIS from a variability testing perspective.

Several repositories and benchmarks are available in the context of variability and SPLs. The SPLOT repository [44,60] stores a collection of feature models commonly used in the literature. SPL2GO [59] is an

online collection of SPLs for which source code and variability model (e.g, feature model) are provided. The Variability Bug Database (VBD) [1] is a collection of faults detected in Linux. For each bug, detailed information is provided such as its location in the code, execution trace, discussion, etc. Garvin et al. [26] presented an exploratory study on two open source systems with publicly available bug databases to understand the nature of real-world interaction faults and to understand the types of mutations that would mimic these faults. Our work contributes to those repositories by providing a complete case study composed of a feature model, available source code and test cases, non–functional attributes as well as the number, type and severity of faults.

A number of works have explored the correlation between non–functional properties of software components and their fault propensity [30, 33, 49]. Although the results may vary among different systems, faults usually correlate well with properties such as code complexity, pre–release defects, test coverage, number and frequency of changes and organization structure [46]. These results are confirmed by our correlation study, which relates the number of bugs in features to the size and cyclomatic complexity of its code, number of changes and number of faults in previous versions of the framework. This may be helpful to identify those non–functional attributes that are more effective at guiding the search for faults in VISs.

11 Conclusions

In this article, we presented the Drupal framework as a motivating real VIS in the context of variability testing. We modelled the framework variability using a feature model and reported on a number of non–functional feature attributes including the number, types and severity of faults. Among other results, we found integration faults caused by the interaction of up to 4 different Drupal features. Also, we found that features providing key functionality of the framework are involved in a high percentage of integration faults, e.g. feature Views is present in 30% of the interaction faults found in Drupal v7.23. Another interesting finding is the absence of no excludes constraints in Drupal. This suggests that variability constraints may differ in different domains. Additionally, we performed a rigorous statistical correlation study to investigate how non–functional properties may be used to predict the presence of faults in Drupal. As a result, we provide helpful insights about the attributes that could (and could not) be effective bug predictors in a VIS. Finally, we presented an experimental evaluation on the use of non–functional data for test case prioritization. The results show that non–functional attributes effectively accelerate the detection of faults of combinatorial test suites, outperforming related functional prioritization criteria as test case similarity.

This case study provides variability researchers and practitioners with helpful information about the distribution of faults and test cases in a real VIS. Also, it is a valuable asset to evaluate variability testing techniques in realistic settings rather than using random variability models and simulated faults. Finally, we trust that this work encourages others to keep exploring on the use of non–functional attributes in the context of variability testing, e.g. from a multi–objective perspective.

12 Material

The Drupal feature model, non–functional attributes, source code of the evaluation and R scripts to reproduce the statistical analysis of the correlations are available at http://www.isa.us.es/anabsanchez-sosym14.

Acknowledgments

We thank the Drupal core developers Francisco José Seva Mora and Christian López Espínola and the anonymous contributors of the Drupal forum for their helpful assistance. We are also grateful to Dr. Myra Cohen and the anonymous reviewers of the 8th International Workshop on Variability Modelling of Software intensive Systems (VaMoS'14) whose comments and suggestions helped us to improve the article substantially.

This work was partially supported by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programmes (grants IPT-2012-0890-390000 (SAAS FIREWALL), TIN2012-32273 (TAPAS), TIC-5906 (THEOS), TIC-1867 (COPAS)).

References

1. I. Abal, C. Brabrand, and A. Wasowski. 42 variability bugs in the linux kernel: A qualitative analysis. In *International Conference on Automated Software Engineering*, pages 421–432, 2014.
2. M. Al-Hajjaji, T. Thum, J. Meinicke, M. Lochau, and G. Saake. Similarity-based prioritization in software product-line testing. In *Software Product Line Conference*, pages 197–206, 2014.
3. C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Why do software packages conflict? In *Conference on Mining Software Repositories*, pages 141–150. IEEE, 2012.

4. E. Bagheri, F. Ensan, and D. Gasevic. Grammar-based test generation for software product line feature models. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 87–101, 2012.
5. E. Bagheri and D. Gasevic. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Control*, 2011.
6. H. Baller, S. Lity, M. Lochau, and I. Schaefer. Multi-objective test suite optimization for incremental product family testing. In *International Conference on Software Testing, Verification, and Validation*, 2014.
7. D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Line Conference*, 2005.
8. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analyses of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.
9. T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 73–82. ACM, 2010.
10. T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, Dec 2013.
11. S. Bergmann. pholoc. `http://github.com/sebastianbergmann/phploc`, accessed March 2014.
12. D. Buytaert. Drupal framework. `http://www.drupal.org`, accessed March 2014.
13. Debian Wheezy. `http://www.debian.org/releases/wheezy/`, accessed March 2014.
14. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed. Automatic and incremental product optimization for software product lines. In *International Conference on Software Testing*, pages 31–40. IEEE, 2014.
15. X. Devroey, G. Perrouin, M. Cordy, P. Schobbens, A. Legay, and P. Heymans. Towards statistical prioritization for software product lines testing. In *Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, number 10, 2014.
16. X. Devroey, G. Perrouin, and P. Schobbens. Abstract test case generation for behavioural testing of software product lines. In *Software Product Line Conference*, volume 2, pages 86–93. ACM, 2014.
17. Drupal JIT module. `http://www.drupal.org/project/thejit`, accessed March 2014.
18. S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *Transactions on Software Engineering*, 28(2):159–182, 2002.
19. S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, 2004.
20. A. Ensan, E. Bagheri, M. Asadi, D. Gasevic, and Y. Biletskiy. Goal-oriented test case selection and prioritization for product line feature models. In *Conference Information Technology: New Generations*, pages 291–298. IEEE, 2011.
21. 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*, pages 613–628, 2012.
22. J. Ferrer, P. Kruse, F. Chicano, and E. Alba. Evolutionary algorithm for prioritized pairwise test data generation. In *Genetic and Evolutionary Computetion Conference*, pages 1213–1220, 2012.
23. J. A. Galindo, M. Alferez, M. Acher, B. Baudry, and D. Benavides. A variability-based testing approach for synthesizing video sequences. In *International Symposium on Software Testing and Analysis*, pages 293–303, 2014.
24. J. A. Galindo, D. Benavides, and S. Segura. Debian packages repositories as software product line models. towards automated analysis. In *Automated Configuration and Tailoring of Applications*, pages 29–34, 2010.
25. 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*, pages 416–426, 2013.
26. B. J. Garvin and M. B. Cohen. Feature interaction faults revisited: an exploratory study. In *International Symposium on Software Reliability Engineering*, pages 90–99, 2011.
27. B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *International Symposium on Search Based Software Engineering*, pages 13–22, 2009.
28. T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. Technical report, National Institute of Statistical Sciences, 653–661, 1998.
29. 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.
30. A. E. Hassan. Predicting faults using the complexity of code changes. In *International Conference on Software Engineering*, pages 78–88, 2009.
31. C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the combinatorial explosion: using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, 40:1, 2014.
32. C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon. Multi-objective test generation for software product lines. In *International Software Product Line Conference*, pages 62–71, 2013.
33. K. Herzig, S. Just, A. Rau, and A. Zeller. Predicting defects using change genealogies. In *International Symposium on Software Reliability Engineering*, pages 118–127, 2013.
34. M. F. Johansen, O. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Software Product Line Conference*, volume 1, pages 46–55, 2012.
35. 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 on Model Driven Engineering Languages and Systems*, pages 269–284, 2012.
36. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. In *Software Engineering Institute*, 1990.
37. B. Pérez Lamancha and M. Polo Usaola. Testing product generation in software product lines using pairwise for feature coverage. In *International conference on Testing Software and Systems*, pages 111–125, 2010.
38. K. S. Lew, T. S. Dillon, and K. E. Forward. Software complexity and its impact on software reliability. *Transactions on software engineering*, 14:1645–1655, 1988.
39. R. E. Lopez-Herrejon, F. Chicano, J. Ferrer, A. Egyed, and E. Alba. Multi-objective optimal test suite computation for software product line pairwise testing. In *IEEE International Conference on Software Maintenance*, pages 404–407, 2013.
40. R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, and E. Alba. Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of software product lines. In *IEEE Congress on Evolutionary Computation*, pages 387–396, 2014.

41. R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the linux kernel variability model. In *Software Product Line Conference*, pages 136–150, 2010.
42. D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu. Practical pairwise testing for software product lines. In *International Software Product Line Conference*, pages 227–235, New York, NY, USA, 2013. ACM.
43. S. Matsumoto, Y. kamei, A. Monden, K. Matsumoto, and M. Nakamura. An analyses of developer metrics for fault prediction. In *International Conference on Predictive Models in Software Engineering*, number 18, 2010.
44. M. Mendonca, M. Branco, and D. Cowan. S.p.l.o.t. - software product lines online tools. In *Conference Companion on Object Oriented Programming Systems Languages and Applications*, pages 761–762, 2009.
45. S. Mohanty, A. Abhinna Acharya, and D. Prasad Mohapatra. A survey on model based test case prioritization. *Computer Science and Information Technologies*, 2:1042–1047, 2011.
46. N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *International Symposium on Software Reliability Engineering*, pages 309–318, 2010.
47. H. V. Nguyen, C. Kästner, and T.N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *International Conference on Software Engineering*, pages 907–918, 6 2014.
48. S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *Software Product Line Conference*, 2010.
49. T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *International symposium on Software testing and analysis*, pages 86–96, 2004.
50. S. R Pressmen. *Software Engineering: A practitioners Approach*. McGraw Hill, International Edition-5, 2001.
51. R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.
52. G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27:929–948, 2001.
53. A. B. Sánchez and S. Segura. Automated testing on the analysis of variability-intensive artifacts: an exploratory study with sat solvers. In *XVII Jornadas de Ingeniería del Software y de Bases de Datos*, 2012.
54. A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. A comparison of test case prioritization criteria for software product lines. In *IEEE International Conference on Software Testing, Verification, and Validation*, pages 41–50, Cleveland, OH, April 2014.
55. A. Salam Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *International Conference on Software Engineering*, pages 492–501, 2013.
56. S. Segura and A. Ruiz-Cortés. Benchmarking on the automated analyses of feature models: A preliminary roadmap. In *International Workshop on Variability Modeling of Software-Intensive Systems*, pages 137–143, 2009.
57. S. Segura, A. B. Sánchez, and A. Ruiz-Cortés. Automated variability analysis and testing of an e-commerce site: An experience report. In *International Conference on Automated Software Engineering*, pages 139–150. ACM, 2014.
58. S. She, R. Lotufo, T. Berger, A. Wasowski, and k. Czarnecki. The variability model of the linux kernel. In *International Workshop on Variability Modelling of Software-intensive Systems*, 2010.
59. SPL2GO repository. `http://spl2go.cs.ovgu.de/`, accessed March 2014.
60. S.P.L.O.T.: Software Product Lines Online Tools. `http://www.splot-research.org/`, accessed March 2014.
61. H. Srikanth, M. B. Cohen, and X. Qu. Reducing field failures in system configurable software: Cost-based prioritization. In *International Symposium on Software Reliability Engineering*, pages 61–70, 2009.
62. M. Svahnberg, L. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. *Software Practice and Experience*, 35:705–754, 2005.
63. P. N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2006.
64. U. Tiwari and S. Kumar. Cyclomatic complexity metric for component based software. In *Software Engineering Notes*, volume 39, pages 1–6, January 2014.
65. T. Tomlinson and J. K. VanDyk. *Pro Drupal 7 development: third edition*. 2010.
66. S. Wang, S. Ali, and A. Gotlieb. Minimizing test suites in software product lines using weight-based genetic algorithms. In *The Genetic and Evolutionary Computation Conference*, pages 1493–1500, 2013.
67. S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, and M. Liaaen. Multi-objective test prioritization in software product line testing: An industrial case study. In *Software Product Line Conference*, pages 32–41, 2014.
68. S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. In *Software Testing, Verification and Reliability*, pages 67–120, 2012.

A Appendix

ID	Module	Type
1	Backup and migrate	Additional
2	Blog	Core optional
3	Captcha	Additional
4	CKEditor	Additional
5	Comment	Core optional
6	Ctools	Additional
7	Ctools access ruleset	Additional
8	Ctools custom content	Additional
9	Date	Additional
10	Date API	Additional
11	Date popup	Additional
12	Date views	Additional
13	Entity API	Additional
14	Entity tokens	Additional
15	Features	Additional
16	Field	Core compulsory
17	Field SQL storage	Core compulsory
18	Field UI	Core optional
19	File	Core optional
20	Filter	Core compulsory
21	Forum	Core optional
22	Google analytics	Additional
23	Image	Core optional
24	Image captcha	Additional
25	IMCE	Additional
26	Jquery update	Additional
27	Libraries API	Additional
28	Link	Additional
29	Node	Core compulsory
30	Options	Core optional
31	Panel nodes	Additional
32	Panels	Additional
33	Panels In-Place Editor	Additional
34	Path	Core optional
35	Pathauto	Additional
36	Rules	Additional
37	Rules scheduler	Additional
38	Rules UI	Additional
39	System	Core compulsory
40	Taxonomy	Core optional
41	Text	Core compulsory
42	Token	Additional
43	User	Core compulsory
44	Views	Additional
45	Views content	Additional
46	Views UI	Additional
47	WebForm	Additional

Table 6 Drupal modules included in the case study

Fault	Feature (ID from Table 6)																																													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
F1	X					X																																									
F2	X												X																																		
F3		X											X																																		
F4			X												X																																
F5				X									X																																		
F6			X	X																																											
F7				X									X																																		
F8					X								X																																		
F9	X					X																																									
F10						X									X																																
F11						X																								X																
F12	X					X								X																																	
F13						X		X																																							
F14	X					X																																									
F15	X					X																																									
F16					X	X	X																																								
F17	X					X																																									
F18						X		X																																							
F19	X				X		X																																								
F20	X					X				X																																					
F21											X	X																																			
F22	X										X	X																																			
F23											X	X																																			
F24	X	X									X	X																																			
F25													X				X																														
F26						X								X																																	
F27	X													X																																	
F28	X													X																																	
F29													X		X																																
F30													X		X																																
F31	X												X		X																																
F32													X		X																																
F33			X										X		X																																
F34	X														X																																
F35	X														X																																
F36															X		X																														
F37														X			X																														
F38	X													X			X																														
F39	X													X			X																														
F40																	X																					X									
F41														X			X																														
F42														X			X																														
F43																	X														X																
F44																	X																					X									
F45																	X	X																													
F46														X			X																														
F47											X						X																														
F48																	X				X																										
F49													X				X																														
F50	X													X			X																														
F51													X				X																														
F52																	X		X																												
F53													X				X																														
F54																	X		X																									X			
F55													X				X																														
F56	X																X																														
F57																	X		X																												
F58																	X		X																												
F59																	X																					X									
F60														X			X																														
F61																	X		X																												
F62																	X		X																												
F63	X																X																											X			
F64																	X																											X			
F65																	X		X																												
F66																	X																											X			
F67																	X	X	X																			X									
F68																	X		X																												
F69																	X		X																												
F70						X											X																														
F71																	X		X																									X			
F72																	X										X																				
F73						X											X																														
F74															X		X																														
F75																	X		X																									X			

Fault	Feature (ID from Table 6)																																													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
F76																	X		X								X																				
F77						X											X																														
F78						X												X																													
F79																		X	X																												
F80																			X																									X			
F81																			X																									X			
F82																			X				X																					X			
F83			X												X				X																									X			
F84													X						X																												
F85																			X																			X									
F86																	X		X																									X			
F86																			X																									X			
F88																			X																									X			
F89																			X																												X
F90																				X																								X			
F91																	X				X																										
F92																							X	X																							
F93													X											X																							
F94																								X																				X			
F95														X										X																							
F96														X										X																							
F97												X				X								X																							
F98		X																						X																							
F99														X										X																							
F100																										X					X																
F101																											X												X								
F102														X														X																			
F103																	X		X										X																		
F104														X																	X																
F105														X																	X																
F106						X																									X																
F107																															X	X															
F108																															X								X								
F109																		X					X								X																
F110																															X								X								
F111																			X												X																
F112																	X														X																
F113																							X	X								X															
F114																															X	X															
F115																																		X				X									
F116																																			X				X								
F117																	X																		X												
F118																																			X		X										
F119																	X																		X												
F120																																				X								X			
F121																																	X			X											
F122													X																							X											
F123	X														X																					X											
F124																																	X			X											
F125																																	X			X											
F126																																				X								X			
F127																																	X				X										
F128																							X														X										
F129																							X														X										
F130																																	X			X		X									
F131													X																									X									
F132																	X																					X	X								
F133					X																															X		X									
F134					X																																	X									
F135													X																										X								
F136																															X								X								
F137																															X								X								
F138																																							X								X
F139																															X	X							X								
F140																							X																X								
F141							X																																X								
F142													X																										X								
F143																															X	X							X								
F144															X																								X								
F145				X															X																							X					
F146																			X																									X			
F147																																												X			X
F148																	X																											X			
F149	X														X																													X			
F150																			X																									X			
F151																																						X						X			

Fault	Feature (ID from Table 6)																																													
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**
F152																																X													X		
F153																			X																											X	
F154																																												X		X	
F155														X																																	X
F156													X																																		X
F157														X																																	X
F158														X																																	X
F159											X																																				X
F160														X																																	X

Table 7: Integration faults in Drupal v7.23