The Journal of Systems and Software 122 (2016) 287–310

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Multi-objective test case prioritization in highly configurable systems: A case study

José A. Parejo[a,*], Ana B. Sánchez[a], Sergio Segura[a], Antonio Ruiz-Cortés[a], Roberto E. Lopez-Herrejon[b], Alexander Egyed[b]

[a] ETS Ingeniería Informática, Universidad de Sevilla, Spain
[b] Institute for Software System Engineering, Johannes Kepler University, Austria

A R T I C L E I N F O

Article history:
Received 23 December 2015
Revised 8 September 2016
Accepted 25 September 2016
Available online 26 September 2016

Keywords:
Variability
Test case prioritization
Automated software testing
Highly-configurable systems

A B S T R A C T

Test case prioritization schedules test cases for execution in an order that attempts to accelerate the detection of faults. The order of test cases is determined by prioritization *objectives* such as covering code or critical components as rapidly as possible. The importance of this technique has been recognized in the context of Highly-Configurable Systems (HCSs), where the potentially huge number of configurations makes testing extremely challenging. However, current approaches for test case prioritization in HCSs suffer from two main limitations. First, the prioritization is usually driven by a single objective which neglects the potential benefits of combining multiple criteria to guide the detection of faults. Second, instead of using industry-strength case studies, evaluations are conducted using synthetic data, which provides no information about the effectiveness of different prioritization objectives. In this paper, we address both limitations by studying 63 combinations of up to three prioritization objectives in accelerating the detection of faults in the Drupal framework. Results show that non–functional properties such as the number of changes in the features are more effective than functional metrics extracted from the configuration model. Results also suggest that multi-objective prioritization typically results in faster fault detection than mono-objective prioritization.

1. Introduction

Highly-Configurable Systems (HCSs) provide a common core functionality and a set of optional features to tailor variants of the system according to a given set of requirements (Cohen et al., 2008; von Rhein et al., 2015). For instance, operating systems such as *Linux* or *eCos* are examples of HCSs where functionality is added or removed by installing and uninstalling packages, e.g. *Debian Wheezy* offers more than 37,000 available packages (Debian, 2013). Content management systems are also examples of HCSs were configuration is managed in terms of modules, e.g. the e-commerce platform *Prestashop* has more than 3500 modules and visual templates (Segura et al., 2014). Recently, cloud applications are also being presented as configurable systems, e.g. the *Amazon Elastic Compute Cloud* (EC2) service offers 1758 different possible configurations (García-Galán et al., 2013).

HCSs are usually represented in terms of features. A *feature* depicts a choice to include a certain functionality in a system configuration (von Rhein et al., 2015). It is common that not all combinations of features are allowed or meaningful. In this case, additional constraints are defined between them, normally using a variability model, such as a feature model. A *feature model* represents all the possible configurations of the HCS in terms of features and constraints among them (Kang et al., 1990). A *configuration* is a valid composition of features satisfying all the constraints. Fig. 1 depicts a feature model representing a simplified family of mobile phones. The model illustrates how features and relationships among them are used to specify the commonalities and variabilities of the mobile phones. The following set of features represents a valid configuration of the model: {Mobile Phone, Calls, Screen, HD, GPS, Media, Camera}.

HCS testing is about deriving a set of configurations and testing each configuration (Perrouin et al., 2011). In this context, a *test case* is defined as a configuration of the HCS under test (i.e. a set of features) and a *test suite* is a set of test cases (Perrouin et al., 2011). Henceforth, the terms test case and configuration are used indistinctly. Testing HCSs is extremely challenging due to the potentially huge number of configurations under test. As an example,

* Corresponding author.
E-mail addresses: japarejo@us.es (J.A. Parejo), anabsanchez@us.es (A.B. Sánchez), sergiosegura@us.es (S. Segura), aruiz@us.es (A. Ruiz-Cortés), roberto.lopez@jku.at (R.E. Lopez-Herrejon), alexander.egyed@jku.at (A. Egyed).

http://dx.doi.org/10.1016/j.jss.2016.09.045
0164-1212/© 2016 Elsevier Inc. All rights reserved.

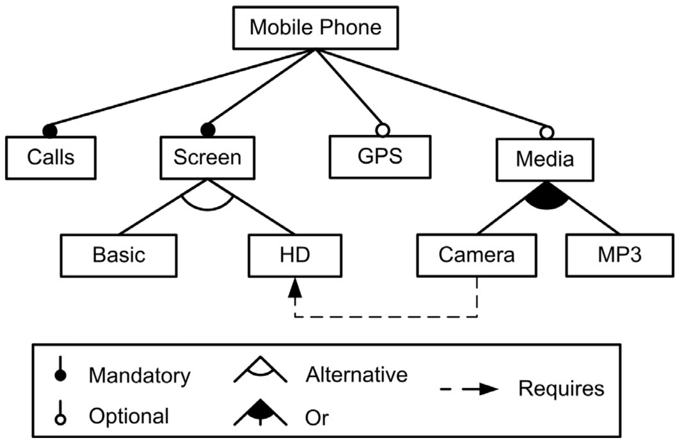

Fig. 1. Mobile phone feature model.

Eclipse (Debian, 2013) has more than 1650 plugins that can be combined (with restrictions) to form millions of different configurations of the development environment. This makes exhaustive testing of HCSs infeasible, that is, testing every single configuration is too expensive in general. Also, even when a manageable set of configurations is available, testing is irremediably limited by time and budget constraints which requires making tough decisions with the goal of finding as many faults as possible.

Typical approaches for HCS testing use a model-based approach, that is, they take an input feature model representing the HCS and return a valid set of feature configurations to be tested, i.e. a test suite. In particular, two main strategies have been adopted: test case selection and test case prioritization. *Test case selection* reduces the test space by selecting an effective and manageable subset of configurations to be tested (Devroey et al., 2014b; Henard et al., 2013; Marijan et al., 2013). *Test case prioritization* schedules test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goal, typically detecting faults as soon as possible (Al-Hajjaji et al., 2014; Lopez-Herrejon et al., 2014; Wang et al., 2014b). Both strategies are complementary and are often combined.

Test case prioritization in HCSs can be driven by different functional and non–functional objectives. Functional prioritization objectives are those based on the functional features of the system and their interactions. Some examples are those based on combinatorial interaction testing (Wang et al., 2014b), configuration dissimilarity (Al-Hajjaji et al., 2014; Henard et al., 2014; Sánchez et al., 2014) or feature model complexity metrics (Sánchez et al., 2015b; Sánchez et al., 2014). Non–functional prioritization objectives consider extra–functional information such as user preferences (Ensan et al., 2011; Johansen et al., 2012), cost (Wang et al., 2014b), memory consumption (Lopez-Herrejon et al., 2014) or execution probability (Devroey et al., 2014a) to find the best ordering for test cases. In a previous work (Sánchez et al., 2015b), we performed a preliminary evaluation comparing the effectiveness of several functional and non–functional prioritization objectives in accelerating the detection of faults in an HCS. Results suggested that non–functional properties such as the number of changes or the number of defects in a previous version of the system were among the most effective prioritization criteria.

Challenges. Current approaches for test case prioritization in HCSs follow a single objective approach (Al-Hajjaji et al., 2014; Johansen et al., 2012; Devroey et al., 2014a; Ensan et al., 2011; Henard et al., 2014; Lopez-Herrejon et al., 2014; Sánchez et al., 2015b), that is, they either aim to maximize or minimize an objective (e.g. feature coverage) or another (e.g. suite size) but not both at the same time. Other works (Wang et al., 2013; 2014b) combine several objectives into a single function by assigning them weights proportional to their relative importance. While this may be acceptable in certain scenarios, it may be unrealistic in others where users may wish to study the trade-offs among several objectives (Lopez-Herrejon et al., 2014). Thus, the potential benefits of optimizing multiple prioritization objectives simultaneously, both functional and non–functional, is a topic that remains unexplored.

A further challenge is related to the lack of HCSs with available code, variability models and fault reports that can be used to assess the effectiveness of testing approaches. As a result, authors typically evaluate their contributions in terms of performance (e.g. execution time) using synthetic feature models and data (Al-Hajjaji et al., 2014; Henard et al., 2013; Qu et al., 2008; Xu et al., 2013). This introduces significant threats to validity, limit the scope of their conclusions and, more importantly, it raises questions regarding the fault–detection effectiveness of the different algorithms and prioritization objectives.

Contributions. In this paper, we present a case study on multi–objective test case prioritization in HCSs. In particular, we model test case prioritization in HCSs as a multi–objective optimization problem, and we present a search–based algorithm to solve it based on the classical NSGA-II evolutionary algorithm. Additionally, we present seven objective functions based on both functional and non–functional properties of the HCS under test. Then, we report a comparison of 63 different combinations of up to three objectives in accelerating the detection of faults in the Drupal framework. Drupal is a highly modular open source web content management system for which we have mined a feature model and extracted real data from its issue tracking system and Git repository (Sánchez et al., 2015b). Results reveal that non–functional properties, such as the number of defects in previous versions of the system, accelerate the detection of faults more effectively than functional properties extracted from the feature model. Results also suggest that multi-objective prioritization is more effective at accelerating the detection of faults than mono-objective prioritization.

The rest of the paper is structured as follows: Section 2 introduces the concepts of feature models and multi-objective evolutionary algorithms. Section 3 presents the Drupal case study used to perform this work. In Section 4 and Section 5 we respectively describe the overview and definition of our approach and the multi-objective optimization algorithm proposed. Section 6 defines seven objective functions for HCSs based on functional and non-functional goals. The evaluation of our approach is described in Section 7. Section 8 presents the threats to validity of our work. The related work is discussed in Section 9. Finally, we summarize our conclusions and outline our future work in Section 10.

2. Background

2.1. Feature models

A *feature model* defines all the possible configurations of a system or family of related systems (Benavides et al., 2010; Kang et al., 1990). A feature model is visually represented as a tree–like structure in which nodes represent features, and edges denote the relationships among them. A *feature* can be defined as any increment in the functionality of the system (Batory, 2005). A *configuration* of the system is composed of a set of features satisfying all the constraints of the model. Fig. 1 shows a feature model describing a simplified family of mobile phones. The hierarchical relationship among features can be divided into:

- *Mandatory*. If a feature has a mandatory relationship with its parent feature, it must be included in all the configurations in which its parent feature appears. In Fig. 1, all mobile phones must provide support for `Calls`.

Table 1
Mobile phone feature attributes.

Feature	Changes	Faults	Size
Basic	1	0	270
Calls	6	10	1,000
Camera	11	8	680
GPS	8	6	460
HD	3	3	510
Media	9	5	1,100
MP3	11	8	390
Screen	2	4	930

- *Optional*. If a feature has an optional relationship with its parent feature, it can be optionally included in all the configurations including its parent feature. For example, GPS is defined as an optional feature of mobile phones.
- *Alternative*. A set of child features has an alternative relationship with their parent feature when only one of them can be selected when its parent feature is part of the configuration. In Fig. 1, mobile phones can provide support for `Basic` or HD (High Definition) screen, but not both of them at the same time.
- *Or*. A set of child features has an or-relationship with their parent when one or more of them can be included in the configurations in which its parent feature appears. In Fig. 1, software for mobile phones can provide support for `Camera`, `MP3` or both in the same configuration.

In addition to the hierarchical relationships between features, a feature model can also contain cross-tree constraints. These are usually of the form:

- *Requires*. If a feature A requires a feature B, the inclusion of A in a configuration implies the inclusion of B in such configuration. In Fig. 1, mobile phones including the feature `Camera` must include support for a HD screen.
- *Excludes*. If a feature A excludes a feature B, both features cannot appear in the same configuration.

The following is a sample configuration derived from the feature model in Fig. 1: {`Mobile Phone`, `Calls`, `Screen`, HD, `Media`, `Camera`}. This configuration includes all the mandatory features (`Mobile Phone`, `Calls`, `Screen`) and some extra features (HD, `Media`, `Camera`) meeting all the constraints of the model, e.g. `Camera` *requires* HD. Feature models can be automatically analysed to extract all its possible configurations or to determine whether a given configuration is valid (it fulfils all the constraints of the model), among other analysis operations (Benavides et al., 2010). Some tool supporting the analysis of feature models are FaMa, SPLAR (Mendonca et al., 2009) and FeatureIDE (Thüm et al., 2014).

Feature models can be extended with additional information by means of feature attributes, these are called *attributed or extended feature models* (Benavides et al., 2010). Feature attributes are often defined as tuples $< name, value >$ specifying non–functional information of features such as cost or memory consumption. As an example, Table 1 depicts three different feature attributes (number of changes, number of faults and lines of code) and their values on the features of the model in Fig. 1.

Feature models are often used to represent the test space of an HCS where each configuration of the model represents a potential test case. Since typical HCSs can have thousands or even millions of different configurations, several sampling techniques have been proposed to reduce the number of configurations to be tested (e.g. Lopez-Herrejon et al., 2015; Marijan et al., 2013; Perrouin et al., 2010). Salient among them is *pairwise testing* whose goal is to select test suites that contain all possible combinations of pairs of features (Lopez-Herrejon et al., 2015). As an example,

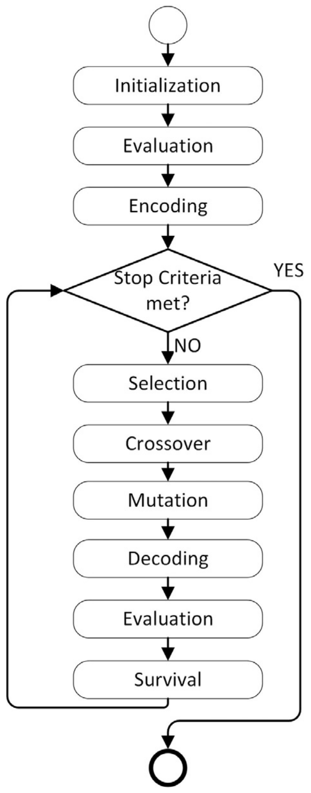

Fig. 2. Working scheme of evolutionary algorithm.

Table 3 shows the set of configurations obtained when applying pairwise testing to the model in Fig. 1. The test suite is reduced from 13 (total number of configurations of the feature model) to five in the pairwise suite. Once a set of configurations are selected for testing, their behaviour has to be tested using standard testing mechanisms, e.g. executable unit tests. However, in this article we focus only on the first step: obtaining a set of high-level test cases respect to different testing objectives. In Section 4 we present in further detail the role of feature models in our work.

2.2. Multi–objective evolutionary algorithms

Evolutionary algorithms are a widely used strategy to solve multi–objective optimization problems. These algorithms manage a set of candidate solutions to an optimization problem that are combined and modified iteratively to obtain better solutions. This process simulates the natural selection of the better adapted individuals that survive and generate offspring improving species. In evolutionary algorithms each solution is referred to as individual or *chromosome*, and objectives are referred to as *fitness functions*.

The working scheme of an evolutionary algorithm is depicted in Fig. 2. Initialization generates the set of individuals that the algorithm will use as starting point. Such initial population is usually generated randomly. Next, the fitness functions are used to assess the individuals. In order to create offspring, individuals need to be encoded, expressing its characteristics in a form that facilitates its manipulation during the rest of the algorithm. Then, the main loop of the evolutionary algorithm is executed until meeting a termination criterion as follows. First, individuals are selected from current population in order to create new offspring. In this process, better individuals usually have higher probability of being selected resembling the natural evolution where stronger individuals have more chances of reproduction. Next, crossover is performed to combine the characteristics of a pair of the chosen individuals to produce new ones in an analogous way to biological reproduction. Crossover mechanisms depend strongly on the scheme used for the encoding. Mutation generates random changes on the new individuals. Changes are performed with certain probability where small modifications are more likely than larger ones. In order to evaluate the fitness of new and modified individuals, decoding is performed and fitness functions are evaluated. Finally, the next population is conformed in such a way that individuals with better fitness values are more likely to remain in the next population.

Multi–Objective Evolutionary Algorithms (MOEAs) are a specific type of evolutionary algorithm where more than one objective are optimized simultaneously. However, except in trivial systems, there rarely exist a single solution that simultaneously optimizes all the objectives. In that case, the objectives are said to be conflicting, and there exists a (possibly infinite) number of so-called Pareto optimal solutions. A solution is said to be a *Pareto optimal* (a.k.a. *non-dominated*) if none of the objectives can be improved without degrading some of the others objectives. Analogously, the solutions where all the objectives can be improved are referred to as *dominated solutions*. The surface obtained from connecting all the Pareto optimal solutions is the so-called *Pareto Front*. Among the many MOEAs proposed in the literature, the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) (Deb et al., 2002) has become very popular due to its effectiveness in many of the benchmarks in multi–objective optimization (Deb and Deb, 2014; Zhou et al., 2011).

3. The drupal case study

In this section, we present the Drupal case study fully reported by the authors in a previous work (Sánchez et al., 2015b). Drupal is a highly modular open source web content management framework written in PHP (Buytaert, 2015; Tomlinson and VanDyk, 2010). This tool can be used to build a variety of websites including internet portals, e-commerce applications and online newspapers (Tomlinson and VanDyk, 2010). Drupal has more than 30,000 modules that can be composed to form valid configurations of the system. The size of the Drupal community (more than 630,000 users and developers) together with its extensive documentation are strengths to choose this framework as our empirical case study. More importantly, the Drupal Git repository and the Drupal issue tracking systems are publicly available sources of valuable functional and non-functional information about the framework and its modules.

Fig. 3 depicts the feature model of Drupal v7.23. Nodes in the tree represent features where a feature corresponds to a Drupal module. A *module* is a collection of functions that provides certain functionality to the system. Some modules extend the functionality of other modules and are modelled as subfeatures, e.g. `Views UI` extends the functionality of `Views`. The feature model includes the core modules of Drupal, modelled as mandatory features, plus some optional modules, modelled as optional features. In addition, the cross-tree constraints of the features in the model are depicted in Fig. 3. These are of the form `X` requires `Y`, which means that configurations including the feature `X` must also include the feature `Y`. A *Drupal configuration* is a combination of features consistent with the hierarchical and cross-tree constraints of the model. In total, the Drupal feature model has 48 features, 21 non-redundant cross-tree constraints and it represents 2.09*E*9 different configurations (Sánchez et al., 2015b).

In this paper, we model the non-functional data from Drupal as feature attributes, depicted in Table 2. These data were obtained from the Drupal website, the Drupal Git repository and the Drupal issue tracking system (Sánchez et al., 2015b). In particular, we use the following attributes:

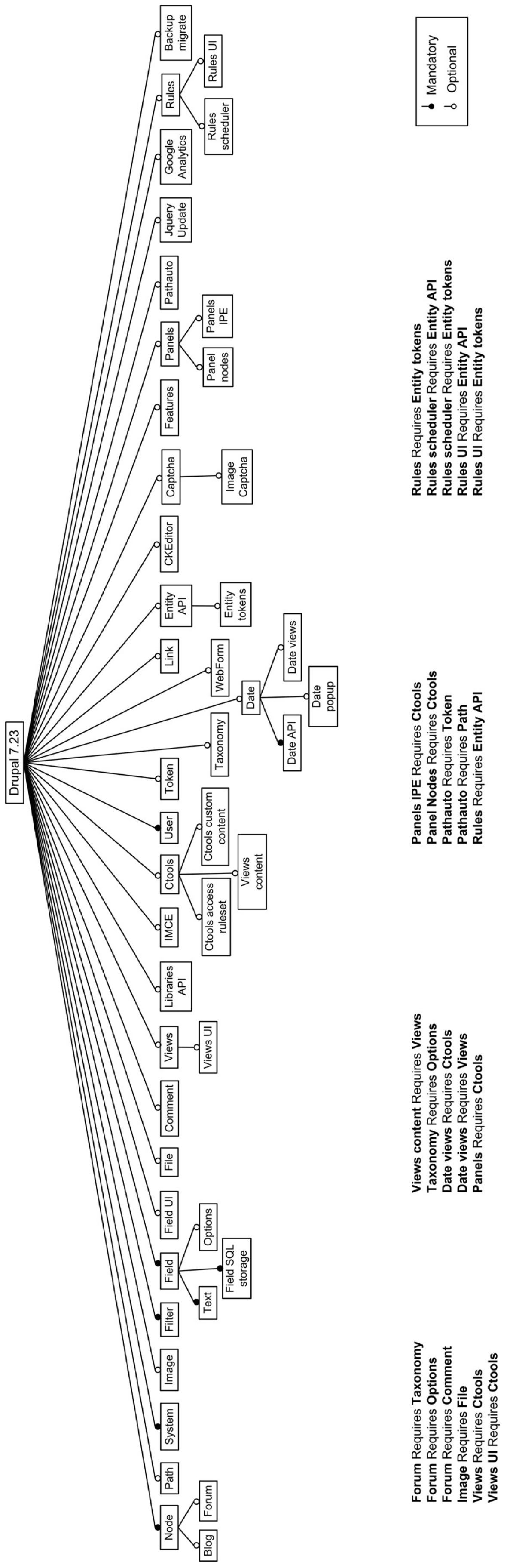

Fig. 3. Drupal feature model.

Table 2
Non-functional feature attributes in Drupal.

Feature	Size	Changes	Faults (v7.22)		Faults (v7.23)	
			Single	Integration	Single	Integration
Backup migrate	11,639	90	80	4	80	4
Blog	551	0	1	3	0	3
Captcha	3115	15	17	1	17	1
CKEditor	13,483	40	197	11	197	9
Comment	5627	1	10	19	13	15
Ctools	17,572	32	181	31	181	31
Ctools acc. rul.	317	0	0	0	0	0
Ctools cus. con.	284	1	10	1	10	1
Date	2696	9	44	3	44	3
Date API	6312	11	41	1	41	1
Date popup	792	4	30	1	30	1
Date views	2383	6	25	1	25	1
Entity API	13,088	14	175	18	175	18
Entity tokens	327	1	22	6	22	6
Features	8483	72	97	9	97	9
Field	8618	7	45	18	48	17
Field SQL sto.	1292	2	3	2	3	2
Field UI	2996	3	13	2	11	1
File	1894	1	10	5	11	5
Filter	4497	3	19	5	19	5
Forum	2849	2	6	4	5	4
Google ana.	2274	14	11	1	11	1
Image	5027	3	10	8	9	6
Image captcha	998	0	3	0	3	0
IMCE	3940	9	9	5	9	5
Jquery update	50,762	1	64	12	64	12
Libraries API	1627	7	11	0	11	0
Link	1934	11	82	4	82	4
Node	9945	4	26	29	24	23
Options	898	1	0	0	0	0
Panel nodes	480	2	16	1	16	1
Panels	13,390	34	87	24	87	24
Panels IPE	1462	20	19	2	19	2
Path	1026	20	3	1	2	1
Pathauto	3429	2	54	9	54	9
Rules	13,830	5	240	15	240	15
Rules sch.	1271	4	13	0	13	0
Rules UI	3306	1	26	0	26	0
System	20,827	16	35	5	35	4
Taxonomy	5757	4	15	22	19	22
Text	1097	1	6	3	5	3
Token	4580	10	37	7	37	7
User	8419	12	20	25	19	22
Views	54,270	27	1091	51	1091	51
Views content	2683	5	23	2	23	2
Views UI	782	0	12	4	12	4
WebForm	13,196	46	292	0	292	0
Total	**336,025**	**573**	**3,231**		**3,232**	

- *Feature size*. Number of Lines of Code (LoC) of the source code associated to the feature (blank lines and test files were excluded from the counting). The sizes range from 284 LoC (feature `Ctools custom content`) to 54,270 LoC (feature `Views`).
- *Number of changes*. Number of commits made by the contributors to the feature in the Drupal Git repository[1] during a period of two years, from 1 May 2012 to 31 April 2014. As illustrated, the number of changes ranges from 0 (feature `Blog`) to 90 (feature `Backup migrate`).
- *Single faults*. Number of faults reported in the Drupal issue tracking system[2]. Faults were collected for two consecutive versions of the framework v7.22 and v7.23 in a period of two years, from 1 May 2012 to 31 April 2014. For instance, we found 19 reported bugs related to the Drupal module Taxonomy (feature `Taxonomy`) in Drupal v7.23. The number of total faults ranges from 0 in features as `Options` to 1091 in the feature `Views`.
- *Integration faults*. List of features for which integration faults have been reported in the Drupal issue tracking system. In total, we identified three faults triggered by the interaction of four features, 25 caused by the interaction of three features and 132 faults triggered by the interaction between two features. These faults have been computed on the features that triggered them in Table 2. For instance, the fault caused by the interaction of `Blog` and `Entity API` is computed as one integration fault in the feature `Blog` and one integration fault in the feature `Entity API`. We refer the reader to Sánchez et al. (2015b) for detailed information about the bug mining process in Drupal.

4. Approach overview

In this section, we define the problem addressed and our approach illustrating it with an example.

[1] http://drupalcode.org/project/drupal.git.
[2] https://drupal.org/project/issues.

Table 3
Mobile phone test suite.

ID	Test case
TC1	Mobile Phone, Calls, Screen, Basic, Media, MP3
TC2	Mobile Phone, Calls, Screen, HD, GPS, Media, Camera, MP3
TC3	Mobile Phone, Calls, Screen, HD, Media, Camera
TC4	Mobile Phone, Calls, Screen, HD
TC5	Mobile Phone, Calls, Screen, Basic, GPS

4.1. Problem

The classical problem of test case prioritization consists in scheduling test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goal (Rothermel et al., 2001). A typical goal is to increase the so-called rate of fault detection, a measure of how quickly faults are detected during testing. In order to meet a goal, prioritization can be driven by one or more objectives. For instance, in order to accelerate the detection of faults, a sample objective could be to increase the code coverage in the system under test at a faster rate, under the assumption that faster code coverage implies faster fault detection.

Inspired by the previous definition, we next define the multi-objective test case prioritization problem in HCSs. Given the set of configurations of an HCS represented by a feature model *fm*, we present the following definitions.

Test case. A test case is a set of features of *fm*, i.e., a configuration. A test case is valid if its features satisfy the constraints represented by the feature model. As an example the following set of features represent a valid test case of the model presented in Fig. 1: {`Mobile Phone, Calls, Screen, Basic, Media, MP3`}.

Test suite. A test suite is an ordered set of test cases. Table 3 depicts a sample test suite of the model presented in Fig. 1.

Objective function. An objective function represents a goal to optimize. In this work, objective functions receive an attributed feature model (*fm*) and a test suite as inputs and return a numerical value measuring the quality of the suite with respect to the optimization goal.

Given a feature model representing the HCS under test and an objective function, the problem of test case prioritization in HCSs consists in generating a test suite that optimizes the target objective. This problem can be generalized to a multi–objective problem by considering more than one objective. In this case, the problem may have more than one solution (i.e., test suites) if there not exist a single solution that simultaneously optimizes all the objectives.

4.2. Our approach

Our approach can be divided in two parts described in the next sections.

4.2.1. Multi-objective test case prioritization

We propose to model the multi–objective test case prioritization problem in HCSs as a multi–objective optimization problem. Fig. 4 illustrates our approach. Given an input attributed feature model, the problem consists in finding a set of solutions (i.e., test suites) that optimize the target objectives. In this paper, we propose seven objective functions based on both functional and non–functional properties of the HCS under test.

4.2.2. Comparison of prioritization objectives

We propose to compare the effectiveness of different combinations of prioritization objectives at accelerating the detection of faults in the Drupal framework. To that purpose, we used historical data collected from a previous version of Drupal as detailed in Section 3. In particular, we propose using the *Average Percentage of Faults Detected (APFD)* (Elbaum et al., 2004; Rothermel et al., 2001; Srikanth et al., 2009) metric to check which one of the Pareto optimal solutions obtained accelerates the detection of faults more effectively. This enables the selection of a global solution and makes it possible to identify the objectives that lead to better test suites.

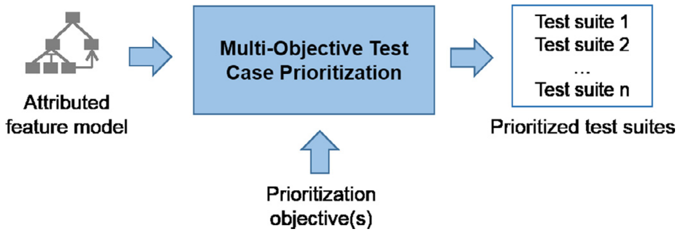

Fig. 4. Our multi–objective test case prioritization approach for HCSs.

Table 4
A set of test suites for the mobile phone.

ID	Test cases	Changes	Faults
TS1	TC4, TC1, TC5, TC3	109	49
TS2	TC1, TC2, TC3, TC4, TC5	80	52
TS3	TC3, TC4, TC5, TC2, TC1	77	57
TS4	TC5, TC4, TC2, TC3, TC1	59	53

The *Average Percentage of Faults Detected (APFD)* (Elbaum et al., 2004; Rothermel et al., 2001; Srikanth et al., 2009) metric measures the weighted average of the percentage of faults detected during the execution of the test suite. To formally define APFD, let *T* be a test suite which contains *n* test cases, and let *F* be a set of *m* faults revealed by *T*. Let *TFi* be the position of the first test case in ordering *T'* of *T* which reveals the fault *i*. The APFD metric for the test suite *T'* is given by the following equation:

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

APFD value ranges from 0 to 1. The closer the value is to 1, the better is the fault detection rate, i.e., the faster is the suite at detecting faults.

4.3. Illustrative example

Table 4 shows the information of four test suites, using the test cases of Table 3. Note that the order of test cases matters. Along with the test cases that compose each suite, the table also shows the value of the objective functions `Changes` and `Faults` defined in Section 6. Roughly speaking, these functions measures the ability of the suite to test those features with a greater number of code changes or reported bugs as quickly as possible.

Fig. 5 depicts the Pareto front obtained when trying to find a test suite that maximizes both objectives. As denoted in the call-out of Fig. 5, TS4 is dominated by TS3, since TS3 detects more faults and covers more changes faster; i.e. TS3 is better than TS4 according to both objectives. Once the optimal test suites are generated, we calculate their APFD to evaluate how quickly they detect faults (based on historical data from a previous version of the system). Consider the faults detected by each test case shown in Table 5. According to the previous APFD equation, test suite TS1 produces an APFD of 46%:

$$1 - \frac{2+2+4+4+1+3}{4 \times 6} + \frac{1}{2 \times 4} = 0.46,$$

TS2 an APFD of 57%:

$$1 - \frac{1+1+2+3+4+5}{5 \times 6} + \frac{1}{2 \times 5} = 0.57$$

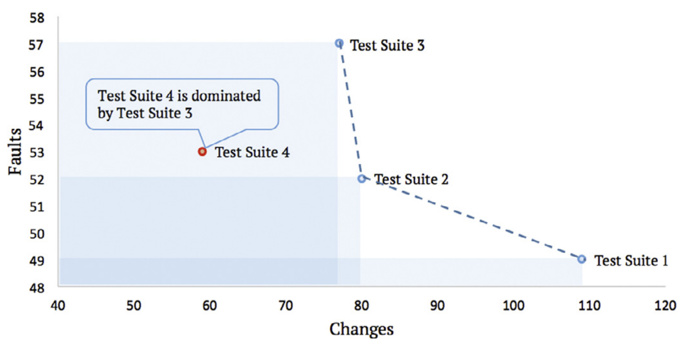

Fig. 5. Test suites of Table 4 as a pareto front for objectives `Changes` and `Faults` (both to be maximized).

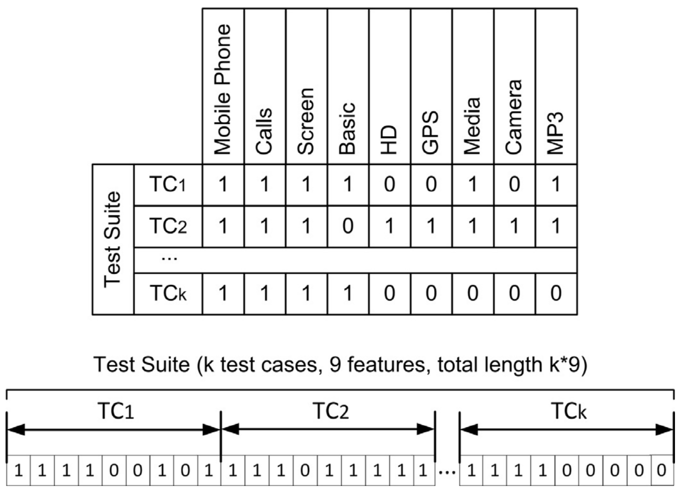

Fig. 6. Test suite encoding as a binary vector.

Table 5
Test suite and faults exposed.

Tests/Faults	F1	F2	F3	F4	F5	F6
TC1	X	X				
TC2	X		X			
TC3	X	X	X	X		
TC4					X	
TC5						X

TS3 an APFD of 80%:

$$1 - \frac{1+1+1+1+2+3}{5 \times 6} + \frac{1}{2 \times 5} = 0.8$$

and TS4 an APFD of 53%:

$$1 - \frac{3+4+3+4+2+1}{5 \times 6} + \frac{1}{2 \times 5} = 0.53$$

Based on the previous results, TS3 is better than TS1 and TS2 and therefore it is the best solution at accelerating the detection of faults. The process could then be repeated with different groups of objectives comparing their effectiveness in terms of the APFD values achieved.

5. Multi-objective optimization algorithm

We used a MOEA to solve the multi–objective test case prioritization problem in HCSs. In particular, we adapted NSGA-II due to its popularity and good performance for many multi-objective optimization problems. In short, the algorithm receives an attributed feature model as input and returns a set of prioritized test suites optimizing the target objectives. In the following, we describe the specific adaptations performed to NSGA-II to solve the multi–objective test case prioritization problem for HCSs.

5.1. Solution encoding

In order to create offspring, individuals need to be encoded expressing their characteristics in a form that facilitates their manipulation during the optimization process. To represent test suites as individuals (chromosomes) we used a binary vector. The vector stores the information of the different test cases sequentially, where each test case is represented by N bits, being N the number of features in the feature model. Thus, the total length of a test suite with k test cases is $k{*}N$ bits, where the first test case is represented by the bits between position 0 and $N-1$, the second test case is represented by the bits between position N and $2*N-1$, and so on. The order of each feature in each test case corresponds to the depth-first traversal order of the tree. A value of 0 in the vector means that the corresponding feature is not included in the test case while a value of 1 means that such feature is included. For efficiency reasons, mandatory features are safely removed from input feature models using atomic sets (Segura, 2008). Fig. 6 illustrates a test suite with its corresponding encoding based on the feature model showed in Fig. 1 (including mandatory features). Note that the length of the vector that encodes the solutions may differ depending on the number of test cases contained in the test suite.

5.2. Initial population

The generation of an appropriate set of initial solutions to the problem (a.k.a. *seeding*) may have a strong impact to the final performance of the algorithm. Lopez-Herrejon et al. (2014) compared several seeding strategies for MOEAs in the context of test case selection in software product lines and concluded that those test suites including all the possible pairs of features (i.e. pairwise coverage) led to better results than random suites. Based on their finding, our initial population is composed of different orderings of a pairwise test suite generated by the CASA tool (Garvin et al., 2011; 2009) from the input feature model.

5.3. Crossover operator

The algorithm uses a customized one–point crossover operator. First, two parent chromosomes (i.e. test suites) are selected to be combined. Then, a random point is chosen in the vector (so-called crossover point) and a new offspring is created by copying the contents of the vectors from the beginning to the crossover point from one parent and the rest from the other one. To avoid creating test suites with non-valid test cases, the crossover point is rounded to the nearest multiple of N in the range [1, SP], being N the number of features in the model and SP the size of the smallest parent. Fig. 7 illustrates a sample crossover operation between two chromosomes of different sizes.

5.4. Mutation operators

We implemented three different mutation operators detailed below.

- *Test case swap*. This mutation operation exchanges the ordering of two randomly chosen test cases.
- *Test case addition/removal*. This mutation operation adds (or removes) a random test case at a randomly chosen index multiple of N in the suite, being N the number of features in the model.

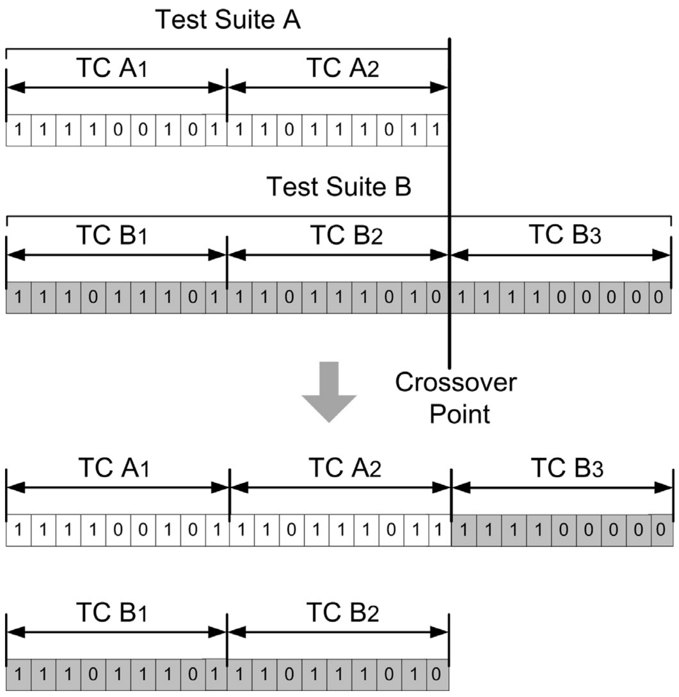

Fig. 7. Crossover operator.

- *Test case substitution*. This mutation operation substitutes a randomly chosen test case from the test suite by another valid test case randomly generated.

Note that all three operators generate feasible solutions, that is, vectors that encode test cases fulfilling all the constraints of the input feature model. Test suites including duplicated test cases as a result of crossover and mutation are discarded.

6. Objective functions

In this section, we propose and formalize different objective functions for test case prioritization in HCSs. All the functions receive an attributed feature model representing the HCS under test (*fm*) and a test suite (*ts*) as inputs and return an integer value measuring the quality of the suite with respect to the optimization goal. Note that the following functions will be later combined to form multi-objective goals (see Section 7). To illustrate each function, we use the feature model in Fig. 1 as *fm* and the test suite $ts = [TC1, TC2]$ with two of the test cases shown in Table 3, which we reproduce next:

```
TC1 = {Mobile Phone,Calls,Screen,Basic,Media,
       MP3}
TC2 = {Mobile Phone,Calls,Screen,HD,GPS,
       Media,Camera,MP3}
```

6.1. Functional objective functions

We propose the following functional objective functions based on the information extracted from the feature model.

Coefficient of Connectivity-Density (CoC). This metric calculates the complexity of a feature model in terms of the number of edges and constraints of the model (Bagheri and Gasevic, 2011). In our previous work (Sánchez et al., 2014), we adapted CoC to HCS configurations achieving good results in accelerating the detection of faults. Now we propose to measure the complexity of features in terms of the number of edges and constraints in which they are involved. This function calculates and accelerates the CoC of a test suite, giving priority to those test cases covering features with higher CoC more quickly. Formally, let the function *coc*(*fm, ts.tc*$_i$) return a value indicating the complexity of the features included in the test case tc_i at position i in test suite *ts*, considering only those features not included in preceding test cases $tc_1..tc_{i-1}$ of test suite *ts*. This objective function is defined as follows:

$$Connectivity(fm, ts) = \sum_{i=1}^{|ts|} \frac{coc(fm, ts.tc_i)}{i} \tag{1}$$

As example, test case TC1 has a CoC of 13 computed as follows: 4 edges in `Mobile Phone`, 1 edge in `Calls`, 3 edges in `Screen`, 1 edge in `Basic`, 3 edges in `Media` and 1 edge in `MP3`. Let us now consider TC2. Notice that the selected features in TC2 that have not already been considered by TC1 are `HD`, `GPS`, and `Camera`. Hence TC2 has a value of 5 computed as follows: 2 edges in `HD`, 1 edge in `GPS`, and 2 edges in `Camera`. Now considering that TC1 is placed in the position 1 and TC2 in position 2, we calculate the function *Connectivity* as follows:

$$\begin{aligned} Connectivity(fm, ts) &= (13/1) \ + \ (5/2) \\ &= 13 + 2.5 = 15.5 \end{aligned}$$

Dissimilarity. Some pieces of work have shown that two dissimilar test cases have a higher fault detection rate than similar ones since the former ones are more likely to cover more components than the latter (Henard et al., 2014; Sánchez et al., 2014). This function favors a test suite with the most different test cases in order to cover more features and improve the rate and acceleration of fault detection. Formally, let the function *df*(*fm, tc*$_i$) return the number of different features found in the test case tc_i that were not considered in preceding test cases $tc_1..tc_{i-1}$. This objective function is defined as follows:

$$Dissimilarity(fm, ts) = \sum_{i=1}^{|ts|} \frac{df(fm, ts.tc_i)}{i} \tag{2}$$

Test case TC1 has a Dissimilarity value of 6 because it considers the following features: `Mobile Phone`, `Calls`, `Screen`, `Basic`, `Media` and `MP3`. Test case TC2 has Dissimilarity value of 3 because it considers the following features that were not part of TC1: `HD`, `GPS` and `Camera`. Now considering that TC1 is placed in the position 1 and TC2 in position 2, we calculate the function *Dissimilarity* as follows:

$$\begin{aligned} Dissimilarity(fm, ts) &= (6/1) \ + \ (3/2) \\ &= 6 + 1.5 = 7.5 \end{aligned}$$

Pairwise coverage. Many pieces of work have used pairwise coverage based on the evidence that a high percentage of detected faults are mainly due to the interactions between two features (e.g. Ferrer et al., 2012; Henard et al., 2012; Sánchez et al., 2014). This objective function measures and accelerates the pairwise coverage of a test suite, giving priority to those test cases that cover a higher number of pairs of features more quickly. Formally, let the function *pc*(*fm, tc*$_i$) return the number of pairs of features covered by the test case tc_i that were not covered by preceding test cases $tc_1..tc_{i-1}$. This objective function is defined as follows:

$$Pairwise(fm, ts) = \sum_{i=1}^{|ts|} \frac{pc(fm, ts.tc_i)}{i} \tag{3}$$

Test case TC1, covers 36 different pairs of features such as the pair [`Calls`, ¬`GPS`] that indicates the feature `Calls` is selected in TC1 and the feature `GPS` is not selected. Test case TC2 covers 27 different pairs of features such as the pair [`HD`, `GPS`] which indicates that both features `HD` and `GPS` are selected. Now considering that

TC1 is placed in the position 1 and TC2 in position 2, we calculate the function *Pairwise* as follows:

$$\begin{aligned} Pairwise(fm, ts) &= (36/1) \; + \; (27/2) \\ &= 36 + 13.5 = 49.5 \end{aligned}$$

Variability coverage and cyclomatic complexity. From a feature model, *Cyclomatic Complexity* measures the number of cross-tree constraints (Bagheri and Gasevic, 2011), while *Variability Coverage* measures the number of variation points (Ensan et al., 2012). A *variation point* is any feature that provides different variants to create a product, i.e. optional features and non-leaf features with one or more non-mandatory subfeatures. These metrics have been jointly used in previous works as a way to identify the most effective test cases in exposing faults, i.e. the higher the sum of both metrics, the better the test case (Ensan et al., 2012; Sánchez et al., 2014). Now, we propose a function that calculates these metrics and gives priority to those test cases obtaining higher values more quickly. Formally, let function *vc(fm,* tc_i) return the number of different cross-tree constraints and the number of variation points involved on the features included in the test case tc_i that were not included in preceding test cases $tc_1..tc_{i-1}$. This objective function is defined as follows:

$$VCoverage(fm, ts) = \sum_{i=1}^{|ts|} \frac{vc(fm, ts.tc_i)}{i} \qquad (4)$$

The features in test case TC1 have 3 variation points in `Mobile Phone`, `Screen` and `Media` features. The features in test case TC2 that were not included in test case TC1 are `GPS`, `HD` and `Camera`. From these three features: `GPS` has one variation point (adds 1), and `HD` and `Camera` are involved in a cross-tree constraint (add 2). Now considering that TC1 is placed in the position 1 and TC2 in position 2, we calculate the function *VCoverage* as follows:

$$\begin{aligned} VCoverage(fm, ts) &= (3/1) \; + \; (3/2) \\ &= 3 + 1.5 = 4.5 \end{aligned}$$

6.2. Non-functional objectives functions

We propose the following non-functional objective functions based on extra–functional information of the features of an HCS.

Number of changes. The number of changes has been shown to be a good indicator of error proneness and can be helpful to predict faults in later versions of systems (e.g. Graves et al., 1998; Yoo and Harman, 2012a). Our work adapts this metric for features in HCSs. This objective function measures the number of changes covered by a test suite and the speed covering those changes, giving a higher value to those test cases that exercise the features with greater number of changes earlier. Therefore, this objective function uses historical data of the HCS under test. Formally, let the function *nc(fm,* tc_i) return the number of code changes covered by features of the test case tc_i at position i that were not covered by preceding test cases $tc_1..tc_{i-1}$. Note that we consider a test case to cover a change if it includes the features where the change was made. This objective function is defined as follows:

$$Changes(fm, ts) = \sum_{i=1}^{|ts|} \frac{nc(fm, ts.tc_i)}{i} \qquad (5)$$

Please refer to Table 1. Test case TC1 covers the following number of changes: 6 changes in the feature `Calls`, 2 changes in `Screen`, 1 change in `Basic`, 9 changes in `Media` and 11 in the feature `MP3`. In total TC1 covers 29 changes. Test case TC2 considers three new features `HD`, `GPS` and `Camera`, which respectively cover 3, 8, and 11 changes. In total TC2 covers 22 changes. Now considering that TC1 is placed in the position 1 and TC2 in position 2, we calculate the function *Changes* as follows:

$$\begin{aligned} Changes(fm, ts) &= (29/1) \; + \; (22/2) \\ &= 29 + 11 = 40 \end{aligned}$$

Number of faults. Earlier studies have shown that the detection of faults in an application can be accelerated by testing first those components that showed to be more error-prone in previous versions of the software. This is referred to as history-based test case prioritization (Huang et al., 2010; Simons and Paraiso, 2010). Our work adapts this metric for features in HCSs. This objective function calculates the number of faults detected by a test suite and its speed revealing those faults, giving a higher value to those test cases that detect more faults faster. This objective uses historical data about the faults reported in a previous version of the HCS under test. Formally, let function *nf(fm,* tc_i) return the number of faults detected by the test case tc_i that were not detected by preceding test cases $tc_1..tc_{i-1}$. Note that we consider a test case to detect a fault if it includes the feature(s) that triggered the fault. This objective function is defined as follows:

$$Faults(fm, ts) = \sum_{i=1}^{|ts|} \frac{nf(fm, ts.tc_i)}{i} \qquad (6)$$

Please refer to Table 1. Test case TC1 detects: 10 faults in the feature `Calls`, 4 faults in feature `Screen`, 0 faults in feature `Basic`, 5 faults in feature `Media` and 8 faults in feature `MP3`. The total number of faults detected by TC1 is 27. Test case TC2 considers three new features `HD`, `GPS` and `Camera` which respectively detect 3, 6 and 8 faults. In total TC2 detects 17 faults. Now considering that TC1 is placed in the position 1 and TC2 in position 2, we calculate the function *Faults* as follows:

$$\begin{aligned} Faults(fm, ts) &= (27/1) \; + \; (17/2) \\ &= 27 + 8.5 = 35.5 \end{aligned}$$

Feature size. The size of a feature, in terms of its number of Lines of Code (LoC), has been shown to provide a rough idea of the complexity of the feature and its error proneness (Lew et al., 1988; Matsumoto et al., 2010; Sánchez et al., 2015b). This objective function measures the size of the features involved in a test suite, giving priority to those test cases covering higher portions of code faster. Formally, let function *fs(fm,* tc_i) return the size of the features included in the test case tc_i that were not included in preceding test cases $tc_1..tc_{i-1}$. This objective function is defined as follows:

$$Size(fm, ts) = \sum_{i=1}^{|ts|} \frac{fs(fm, ts.tc_i)}{i} \qquad (7)$$

Please refer to Table 1. The size contributed by test case TC1 is 3,690 LoC computed by adding: 1000 for feature `Calls`, 930 for feature `Screen`, 270 for feature `Basic`, 1100 for feature `Media` and 390 for feature `MP3`. The new features that test case TC2 considers are: feature `HD` with size 510, feature `GPS` with size 460 and feature `Camera` with size 680. Hence, the total for test case TC2 is 1,650 LoC. Now considering that TC1 is placed in the position 1 and TC2 in position 2, we calculate the function *Size* as follows:

$$\begin{aligned} Size(fm, ts) &= (3690/1) \; + \; (1650/2) \\ &= 3690 + 825 = 4515 \end{aligned}$$

7. Evaluation

This section explains the experiments conducted to explore the effectiveness of multi–objective test case prioritization in Drupal. First, we introduce the target research questions and the general experimental setup. Second, the results of the different experiments and the statistical results are reported.

Table 6
Parameter settings for the evolutionary algorithm.

Parameter	Value
Population size	100
Number of generations	50
Crossover probability	0.9
Test case swap mutation probability	$0.4*(1/N)$
Test case addition/removal mutation probability	$0.3*(1/N)$
Test case substitution mutation probability	$0.3*(1/N)$

7.1. Research questions

In previous works, we investigated the effectiveness of functional (Sánchez et al., 2014) and non–functional (Sánchez et al., 2015b) test case prioritization criteria for HCSs from a single–objective perspective. In this paper, we go a step further in order to answer the following Research Questions (RQs):

RQ1: *Can multi-objective prioritization with functional objective functions accelerate the detection of faults in HCSs?*
RQ2: *Can multi-objective prioritization with non-functional objective functions accelerate the detection of faults in HCSs?*
RQ3: *Can multi-objective prioritization with combinations of functional and non-functional objective functions accelerate the detection of faults in HCSs?*
RQ4: *Are non-functional prioritization objectives (either in a single or multi-objective perspective) more, less or equally effective than functional prioritization objectives in accelerating the detection of faults in HCSs?*
RQ5: *What is the performance of the proposed MOEA compared to related algorithms?*

7.2. Experimental setup

To answer our research questions, we implemented the algorithm and the objective functions described in Sections 5 and 6 respectively. To put it simply, our algorithm takes the Drupal attributed feature model as input and generates a set of prioritized test suites according to the target objective functions. In particular, the algorithms were executed with all the possible combinations of 1, 2 and 3 of the objectives functions described in Section 6, yielding 63 combinations in total. In all cases, the goal was to generate prioritized test suites that maximize each objective function, e.g. max(Changes) and max(VCoverage). For each combination of objectives, the algorithms were executed 40 times to perform statistical analysis of the data. The configuration parameters of the NSGA-II algorithm are depicted in Table 6. These were selected based on the recommended parameters for NSGA-II (Deb et al., 2002) and the results of some preliminary tuning experiments. Note that the recommended default mutation probability for NSGA-II is $1/N$, where N is the number of variables of the problem, i.e. number of test cases in the suite. The average number of test cases in the pairwise suites generated by CASA and used as seed was 13.

The search-based algorithms were implemented using jMetal (Durillo and Nebro, 2011), a Java framework to solve multi–objective optimization problems. The non-functional objective functions were calculated using the Drupal feature attributes reported in Table 2. In particular, the objective function `Faults` was calculated on the basis of the faults detected in Drupal v7.22. The function `Pairwise` was implemented using the tool SPLCAT (Johansen et al., 2011) which generates all the possible pairs of features of an input feature model. Random valid products (used in one of our mutation operators) were generated using the tool PLEDGE (Henard et al., 2014), which internally uses a SAT solver.

The prioritized test suites generated by the algorithm were evaluated according to their ability to accelerate the detection of faults in Drupal. To that purpose, we used the information about the faults reported in Drupal v7.23 (3,392 in total, including single and integration faults) to measure how quickly they would be detected by the generated suites. More specifically, we created a list of faulty feature sets simulating the faults reported in the bug tracking system of Drupal v7.23. Each set represents faults caused by n features ($n \in [1, 4]$). For instance, the list {{`Node`},{`Views`, `Ctools`}} represents a fault in the feature `Node` and another fault triggered by the interaction between the features `Views` and `Ctools`. We considered that a test case detects a fault if the test case includes the feature(s) that trigger the fault. As a further example, consider the list of faulty features {{`Media`},{`HD`},{`Camera`, `GPS`}} and the following test case for the feature model in Fig. 1: {`Mobile Phone`,`Calls`,`Screen`,`HD`,`Media`,`Camera`}. The test case would detect the fault in `Media` and `HD` but not the interaction fault between `Camera` and `GPS` since `GPS` is not included in the configuration.

In order to evaluate how quickly faults are detected during testing (i.e., rate of fault detection) we used the Average Percentage of Faults Detected (APFD) metric described in Section 4.2.2. Given a prioritized test suite, this metric was used to measure how quickly it would detect the faults in Drupal v7.23. For comparative reasons, we measured the APFD values of both, the prioritized suites generated by our adaptation of NSGA-II and the initial pairwise suite generated by the CASA algorithm (Garvin et al., 2011; 2009) on each execution.

In addition to the comparison between NSGA-II and CASA, we compared NSGA-II with a random search algorithm and a deterministic state of the art prioritization algorithm. The details of this comparison are presented in Section 7.7.

We ran our tests on an Ubuntu 14.04 machine equipped with INTEL i7 with 8 cores running at 3.4 Ghz and 16 GB of RAM.

7.3. Experiment 1. Functional objectives

In this experiment, we evaluated the rate of fault detection achieved by each group of 1, 2 and 3 functional objectives, 14 combinations in total. The results of the experiment are shown in Table 7. For each set of objectives, the table shows the results of 40 different executions of NSGA-II and CASA respectively. For NSGA-II, the table depicts the average APFD value of all the test suites generated (i.e., Pareto optimal solutions), average of the maximum APFD value achieved on each execution and maximum APFD value obtained in all the executions respectively. For CASA, the table shows the average and maximum APFD values achieved in all the executions. The top three best average and maximum APFD values of the table are highlighted in boldface. We must remark that all the test suites generated detected at least 99% of the emulated faults. Thus, we omit the results related to the number of faults detected and focus on how quickly they were detected.

The results in Table 7 show that all the functional prioritization objectives, single or combined, outperformed CASA on both the average and maximum APFD values obtained. In total, NSGA-II achieved an average APFD value of 0.918 while CASA achieved 0.872. This was expected since CASA was not conceived as a test case prioritization algorithm. It is also noteworthy that the `Pairwise` objective produced the worst results. This finding is also observed in the box plot of Fig. 8 which illustrates the distributions of the maximum APFD values found on each execution of NSGA-II (40 in total). The `Pairwise` objective function obtained the lowest minimum, maximum and median values. This is lined with the results of CASA and it suggests that pairwise coverage is not an effective prioritization criterion. Interestingly, however, despite the bad performance of `Pairwise` as a single objective, its combination with other objectives provides good results in general, since it is involved in the objective combinations with better

Table 7
APFD values achieved by functional prioritization objectives.

Objectives	NSGA-II			CASA	
	Avg	Avg Max	Max	Avg	Max
Connectivity	0.923	0.936	0.957	0.874	0.939
Dissimilarity	0.905	0.928	0.956	0.865	0.934
Pairwise	0.887	0.887	0.951	0.862	0.934
VCoverage	0.888	0.934	0.956	0.874	0.946
Connectivity + Pairwise	0.932	0.951	**0.959**	0.883	0.939
Connectivity + Dissimilarity	0.906	0.935	**0.960**	0.863	0.947
Connectivity + VCoverage	0.919	0.936	0.957	0.874	0.944
Dissimilarity + Pairwise	**0.941**	0.952	0.958	0.888	0.948
Dissimilarity + VCoverage	0.909	0.934	**0.959**	0.865	0.944
Pairwise + VCoverage	0.933	0.948	0.957	0.867	0.941
Connectivity + Dissimilarity + Pairwise	0.933	0.953	0.957	0.878	0.946
Connectivity + Dissimilarity + VCoverage	0.908	0.935	0.957	0.872	0.937
Connectivity + Pairwise + VCoverage	**0.935**	0.951	**0.959**	0.876	0.940
Dissimilarity + Pairwise + VCoverage	**0.937**	0.953	**0.959**	0.865	0.937
Average	**0.918**	**0.938**	**0.957**	**0.872**	**0.941**

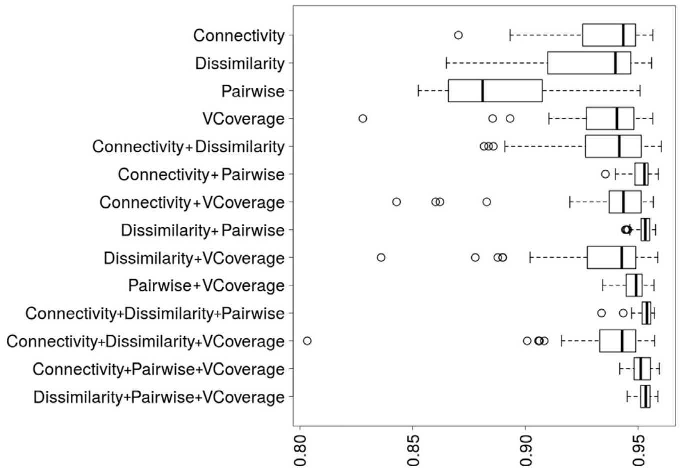

Fig. 8. Box plot of the maximum APFD achieved on each execution (40 in total).

medians and averages. It is also observed that multi-objective combinations provide better distributions of APFD values than single objectives.

In order to accurately answer the research questions we performed several hypothesis statistical tests. Specifically, for each single functional objective (e.g. Connectivity) and combination of two or three functional objectives (e.g. Pairwise and Dissimilarity) we stated a null and alternative hypothesis. The null hypothesis (H^0) states that there is not a statistically significant difference between the results obtained by both sets of objectives while the alternative hypothesis (H^1) states that such difference is statistically significant. Statistical tests provide a probability (named p-value) ranging in [0, 1]. Researchers have established by convention that p-values under 0.05 are so-called statistically significant and are sufficient to reject the null hypothesis. Since the results do not follow a normal distribution, we used the Mann-Whitney U Tests for the analysis (Mann and Whitney, 1947). Additionally, a correction of the p-values was performed using the Holms post-hoc procedure (Holm, 1979) as recommended in Derrac et al. (2011). The tables of specific p-values are provided as supplementary material.

As a further analysis, we used Vargha and Delaney's $\widehat{A_{12}}$ statistic (Arcuri and Briand, 2014) to evaluate the effect size, i.e., determine which mono or multi–objective combinations perform better and to what extent. Table 8 shows the effect size statistic. Each cell shows the $\widehat{A_{12}}$ value obtained when comparing the single objectives in the columns against the combination of objectives in the rows. Note that CASA was considered as another prioritization objective in our analysis. Vargha and Delaney (2000) suggested thresholds for interpreting the effect size: 0.5 means no difference at all; values over 0.5 indicates a small (0.5-0.56), medium (0.57-0.64), large (0.65-0.71) or very large (0.72-1) difference in favour of the multiple objective in the row; values below 0.5 indicates a small (0.5-0.44), medium (0.43-0.36), large (0.36-0.29) or very large (0.29-0.0) difference in favour of the single objective in the column. Cells revealing very large differences are highlighted in light grey (in favour of the row) and dark grey (in favour of the column). Values in boldface are those where hypothesis test revealed statistical differences (p-value $<$ 0.05). Statistical results confirm the bad performance of CASA and the Pairwise objective function compared to the rest of objectives. Since values in Table 8 are in general above 0.5 and most of the cells are shaded in light gray, general results confirm that multi–objective prioritization provides better results for the rate of fault detection than mono–objective prioritization when using functional objectives.

The average execution time of NSGA-II for all the functional objectives was 12.1 minutes, with a maximum average execution time of 3.6 hours for the combination of objectives Connectivity + Pairwise + VCoverage, and a minimum execution time of 69 seconds for the objective Dissimilarity. It is noticeable that all the executions including the objective Pairwise took an average execution time longer than 20 minutes, due to the overhead introduced by the calculation of the pairwise coverage. The average execution time of CASA was 5 seconds.

7.4. Experiment 2. Non–functional objectives

In this experiment, we evaluated the rate of fault detection achieved by each group of 1, 2 and 3 non–functional prioritization objectives, 7 combinations in total. Table 9 presents the APFD values achieved by NSGA-II and CASA with each set of objectives. As in the previous experiment, the average and maximum APFD values achieved by NSGA-II (with any objective) were higher than those achieved by CASA. This confirms the poor performance of pairwise coverage as a prioritization criterion. Interestingly, the Faults objective function is involved in the best average and maximum APFD values. This suggests that the number of faults in previous versions of the system is a key factor to accelerate the detection of faults. All the test suites generated detected more than 99.9% of the emulated faults.

Fig. 9 depicts a box plot of the distributions of the maximum APFD value achieved on each execution of NSGA-II. The graph

Table 8
$\widehat{A}_{12}$ values for mono vs. multi–objective prioritization using functional objectives. Cells revealing very large statistical differences are highlighted in light grey (in favour of the row) and dark grey (in favour of the column). Values in boldface reveal statistically significant differences (the *p*-value with Holm's correction < 0.05).

Functional Multi-Objective	Functional Mono-Objective				CASA
	Connectivity	Dissimilarity	Pairwise	VCoverage	
Connectivity + Dissimilarity	0.484	0.587	**0.923**	0.527	**0.946**
Connectivity + Pairwise	0.801	**0.880**	**0.992**	**0.839**	**0.999**
Connectivity + VCoverage	0.528	0.631	**0.893**	**0.577**	**0.924**
Dissimilarity + Pairwise	**0.840**	**0.911**	**0.994**	**0.873**	**0.994**
Dissimilarity + VCoverage	0.489	0.593	**0.901**	0.530	**0.914**
Pairwise + VCoverage	0.698	**0.791**	**0.983**	**0.757**	**0.997**
Connectivity + Dissimilarity + Pairwise	0.851	**0.921**	**0.996**	**0.884**	**0.998**
Connectivity + Dissimilarity + VCoverage	0.504	0.606	**0.921**	0.552	**0.924**
Connectivity + Pairwise + VCoverage	0.789	0.870	0.988	0.831	1.000
Dissimilarity + Pairwise + VCoverage	**0.855**	**0.924**	**0.994**	**0.892**	**1.000**
CASA	**0.064**	**0.054**	**0.276**	**0.073**	-

Table 9
APFD values achieved by non-functional prioritization objectives.

Objectives	NSGA-II			CASA	
	Avg	Avg Max	Max	Avg	Max
Changes	0.902	0.922	**0.959**	0.871	0.927
Faults	**0.955**	0.955	**0.959**	0.873	0.944
Size	0.921	0.934	0.955	0.868	0.932
Changes + Faults	**0.953**	0.955	**0.959**	0.865	0.952
Changes + Size	0.915	0.936	0.956	0.868	0.938
Faults + Size	**0.955**	0.955	**0.959**	0.876	0.940
Changes + Faults + Size	0.952	0.955	**0.959**	0.871	0.942
Average	**0.936**	**0.945**	**0.958**	**0.871**	**0.939**

Table 10
$\widehat{A}_{12}$ values for mono vs. multi-objective prioritization using non–functional objectives. Cells revealing very large statistical differences are highlighted in light grey (in favour of the row). Values in boldface reveal statistically significant differences (the p-value with Holms correction < 0.05).

Non-Functional Multi-Objective	Mono-Objective		
	Changes	Faults	Size
Changes + Faults	**0.955**	0.565	**0.968**
Changes + Size	0.670	**0.084**	0.549
Faults + Size	**0.960**	0.597	**0.978**
Changes + Faults + Size	**0.951**	**0.536**	**0.960**

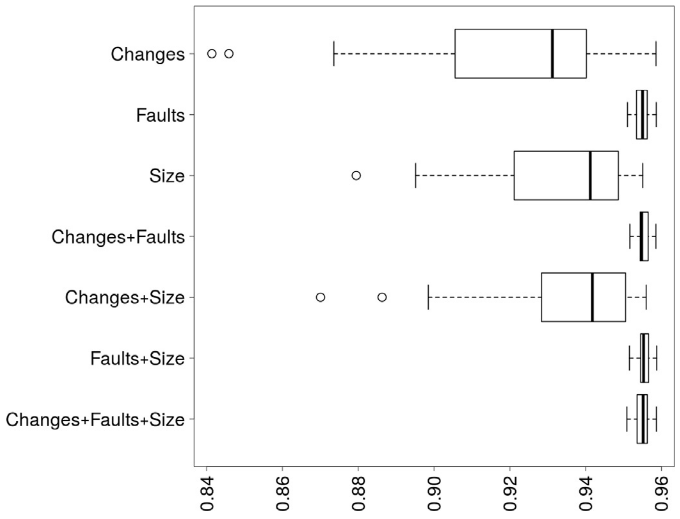

Fig. 9. Box plot of the maximum APFD achieved on each execution (40 in total).

clearly shows the dominance of the Faults objective function, both in isolation and in combination with other objectives. This was confirmed by the statistical tests, where p-values revealed significant differences between the groups of objectives including Faults and the rest of objectives.

Table 10 shows the values of the $\widehat{A}_{12}$ effect size. CASA is excluded from the table since it was clearly outperformed by all other objectives. Again, the results show the superiority of Faults, either in isolation or combined, when compared to any other group of objectives. As in the previous experiment, all the multi–objective combinations improve the results obtained by single objectives, with $\widehat{A}_{12}$ values over 0.5 in all cells except one. No clear differences were found between the use of multi–objective prioritization with two or three objectives.

The average execution time of NSGA-II for all the combinations of non-functional objectives was 3.7 minutes, with a maximum average execution time of 4.6 minutes for Faults + Size and a minimum average execution time of 2.5 seconds for Size.

7.5. Experiment 3. Functional and non–functional objectives

In this experiment, we evaluated the rate of fault detection achieved by each mixed combination of 2 and 3 functional and non–functional prioritization objectives, 48 combinations in total. The results of the experiment are presented in Table 11. The cells with the top three best average, average maximum, and global maximum APFD values of the table are highlighted in boldface. The results show that all the multi–objective combinations greatly improved CASA on the average, average maximum, and global maximum APFD values obtained. As in the previous experiment, 10 out of the 12 top best APFD values were achieved by multi–objective combinations including the objective Faults, which confirms the effectiveness of fault history in accelerating the detection of faults in Drupal. Analogously, 6 out of the 10 best APFD values include the objective Dissimilarity which confirms the findings of previous studies on the effectiveness of promoting the differences among test cases to detect faults more quickly. As in the previous experiments, all the test suites generated detected at least the 99% of the seeded faults.

Table 12 shows the values of the $\widehat{A}_{12}$ effect size on the comparison between single and multi–objective combinations of

Table 11
APFD values achieved by functional and non-functional prioritization objectives.

Objectives	NSGA-II			CASA	
	Avg	Avg Max	Max	Avg	Max
Changes + Connectivity	0.911	0.936	0.959	0.871	0.942
Changes + Dissimilarity	0.905	0.935	0.959	0.873	0.943
Changes + Pairwise	0.938	0.952	0.959	0.878	0.950
Changes + VCoverage	0.919	0.940	0.958	0.867	0.941
Connectivity + Faults	**0.954**	0.955	0.959	0.884	0.935
Connectivity + Size	0.915	0.941	0.959	0.881	0.946
Dissimilarity + Faults	**0.954**	**0.956**	0.959	0.871	0.947
Dissimilarity + Size	0.904	0.930	0.957	0.858	0.921
Faults + Pairwise	0.944	0.954	0.959	0.868	0.944
Faults + VCoverage	**0.954**	0.955	0.959	0.869	0.940
Pairwise + Size	0.940	0.953	0.960	0.878	0.943
Size + VCoverage	0.914	0.937	0.958	0.876	0.948
Changes + Connectivity + Dissimilarity	0.908	0.938	0.958	0.873	0.935
Changes + Connectivity + Faults	0.950	0.955	0.959	0.875	0.935
Changes + Connectivity + Pairwise	0.936	0.953	0.958	0.862	0.933
Changes + Connectivity + Size	0.914	0.942	0.959	0.878	0.929
Changes + Connectivity + VCoverage	0.916	0.942	0.959	0.876	0.947
Changes + Dissimilarity + Faults	0.952	0.955	0.959	0.880	0.936
Changes + Dissimilarity + Pairwise	0.939	0.954	0.958	0.874	0.936
Changes + Dissimilarity + Size	0.911	0.941	0.957	0.867	0.943
Changes + Dissimilarity + VCoverage	0.910	0.946	0.957	0.869	0.945
Changes + Faults + Pairwise	0.944	0.954	0.958	0.874	0.941
Changes + Faults + VCoverage	0.951	0.955	0.959	0.883	0.946
Changes + Pairwise + Size	0.941	0.955	**0.963**	0.866	0.952
Changes + Pairwise + VCoverage	0.937	0.954	0.958	0.875	0.947
Changes + Size + VCoverage	0.909	0.940	0.957	0.874	0.940
Connectivity + Dissimilarity + Faults	**0.954**	**0.956**	0.960	0.877	0.941
Connectivity + Dissimilarity + Size	0.913	0.941	0.959	0.879	0.941
Connectivity + Faults + Pairwise	0.944	0.954	**0.964**	0.867	0.932
Connectivity + Faults + Size	**0.954**	0.955	0.959	0.876	0.947
Connectivity + Faults + VCoverage	0.953	0.955	0.959	0.858	0.925
Connectivity + Pairwise + Size	0.937	0.954	**0.962**	0.870	0.937
Connectivity + Size + VCoverage	0.908	0.936	0.959	0.881	0.954
Dissimilarity + Faults + Pairwise	0.944	0.955	0.959	0.871	0.947
Dissimilarity + Faults + Size	0.953	0.955	0.959	0.875	0.938
Dissimilarity + Faults + VCoverage	0.953	**0.956**	**0.964**	0.876	0.947
Dissimilarity + Pairwise + Size	0.940	0.954	0.959	0.868	0.944
Dissimilarity + Size + VCoverage	0.913	0.941	0.957	0.873	0.936
Faults + Pairwise + Size	0.942	0.955	0.959	0.863	0.937
Faults + Pairwise + VCoverage	0.944	0.954	0.958	0.866	0.938
Faults + Size + VCoverage	0.953	**0.956**	0.959	0.874	0.931
Pairwise + Size + VCoverage	0.941	0.954	0.959	0.877	0.938
Average	**0.934**	**0.949**	**0.959**	**0.873**	**0.940**

functional and non–functional objectives. Values indicate a better performance of multi–objective prioritization compared to single–objective prioritization with the exception of `Faults` where most cells were under 0.5. The overall dominance, however, was observed in the combination of objectives `Dissimilarity + Faults` followed by `Dissimilarity + Faults + VCoverage`, with values over 0.6 in all cells and over 0.93 in 6 out of 7 columns.

Table 13 depicts the effect size on the comparison between multi–objective prioritization using functional objectives and multi–objective prioritization using both functional and non–functional objectives. In general, $\widehat{A_{12}}$ values show statistical differences in favour or the combination of functional and non-functional objectives, especially those including `Faults`. Interestingly, all the cells revealing differences in favour of functional-objectives include the objective `Pairwise`, which supports its potential when combined with other prioritization objectives, as observed in Experiment 1.

Finally, Table 14 depicts the effect size on the comparison between multi–objective prioritization using non–functional objectives and multi–objective prioritization using both functional and non–functional objectives. $\widehat{A_{12}}$ values reveal that when `Faults` is present in the combination of non–functional objectives, mixed combinations are outperformed in general, showing large effect sizes and statistically significant differences. On the contrary, mixed objective combinations including `Faults` clearly outperform `Changes + Size`, but behave slightly worse than the other combinations of non-functional objectives. Therefore, the objective `Faults` seems to have a key influence in the performance of prioritization providing slightly better result when combined with other non-functional objectives. It is remarkable, however, that some mixed combinations of objectives such as `Dissimilarity + Faults` provided the best overall results of this experiment.

The average execution time of NSGA-II for the mixed combinations of functional and non–functional prioritization objectives was 9.2 minutes. The maximum average execution time was 23.6 minutes reached by the objectives `Connectivity + Faults + Pairwise`. The minimum execution time, 73.8 seconds, was obtained by the combination of objectives `Connectivity + Size + VCoverage`.

7.6. Experiment 4. Functional vs non–functional objectives

In this experiment, we performed a further statistical analysis of the data obtained in previous experiments to measure the effect size on the comparison of functional objectives against non–

Table 12
$\widehat{A}_{12}$ values for mono vs. multi–objective combinations of functional and non-functional objectives. Cells revealing very large statistical differences are highlighted in light grey (in favour of the row) and dark grey (in favour of the column). Values in boldface reveal statistically significant differences (the p-value with Holm's correction is < 0.05).

Mixed Multi-Objective	Functional Objectives				Non-Functional Objectives		
	Connectivity	Dissimilarity	Pairwise	VCoverage	Changes	Faults	Size
Changes + Connectivity	0.550	0.639	**0.902**	0.588	0.693	**0.124**	0.596
Changes + Dissimilarity	0.518	0.614	0.906	0.573	0.671	**0.118**	0.563
Changes + Pairwise	0.814	**0.893**	**0.993**	**0.856**	**0.899**	**0.298**	**0.858**
Changes + VCoverage	0.530	0.633	**0.954**	**0.571**	0.701	**0.131**	0.583
Connectivity + Faults	**0.924**	**0.967**	**1.000**	**0.946**	**0.948**	0.556	**0.966**
Connectivity + Size	0.566	0.684	**0.956**	**0.616**	0.736	**0.092**	0.630
Dissimilarity + Faults	**0.951**	**0.967**	**0.999**	**0.958**	**0.962**	0.686	**0.971**
Dissimilarity + Size	0.455	0.553	0.879	0.481	0.601	**0.108**	0.497
Faults + Pairwise	**0.885**	**0.947**	**0.997**	**0.914**	**0.931**	**0.418**	**0.931**
Faults + VCoverage	**0.936**	**0.964**	**0.999**	**0.949**	**0.956**	0.589	**0.972**
Pairwise + Size	**0.863**	**0.931**	**0.994**	**0.903**	**0.914**	**0.336**	**0.893**
Size + VCoverage	0.542	0.631	**0.924**	0.584	0.685	**0.101**	0.580
Changes + Connectivity + Dissimilarity	0.534	0.634	**0.939**	0.570	0.681	**0.128**	0.578
Changes + Connectivity + Faults	**0.918**	**0.961**	**0.998**	**0.938**	**0.944**	**0.509**	**0.953**
Changes + Connectivity + Pairwise	0.877	**0.939**	**0.995**	**0.907**	**0.921**	**0.370**	**0.911**
Changes + Connectivity + Size	0.593	0.693	**0.958**	0.646	0.744	**0.143**	0.644
Changes + Connectivity + VCoverage	0.573	0.673	**0.960**	**0.623**	0.743	**0.176**	0.633
Changes + Dissimilarity + Faults	**0.932**	**0.965**	**0.999**	**0.948**	**0.952**	0.570	**0.963**
Changes + Dissimilarity + Pairwise	**0.873**	**0.936**	**0.995**	**0.911**	**0.925**	**0.384**	**0.910**
Changes + Dissimilarity + Size	0.541	0.639	**0.963**	0.589	0.728	**0.121**	0.600
Changes + Dissimilarity + VCoverage	0.677	0.761	**0.970**	0.731	0.812	**0.189**	0.730
Changes + Faults + Pairwise	**0.910**	**0.948**	**0.996**	**0.930**	**0.941**	**0.471**	**0.942**
Changes + Faults + VCoverage	**0.926**	**0.963**	**0.999**	**0.944**	**0.950**	**0.554**	**0.964**
Changes + Pairwise + Size	**0.909**	**0.953**	**0.996**	**0.930**	**0.939**	**0.494**	**0.938**
Changes + Pairwise + VCoverage	0.888	**0.947**	**0.997**	**0.917**	**0.932**	**0.429**	**0.925**
Changes + Size + VCoverage	0.573	0.682	**0.943**	0.620	0.729	**0.093**	0.625
Connectivity + Dissimilarity + Faults	**0.940**	**0.963**	**0.999**	**0.956**	**0.959**	0.624	**0.973**
Connectivity + Dissimilarity + Size	0.558	0.659	0.951	0.606	0.728	**0.133**	0.622
Connectivity + Faults + Pairwise	**0.899**	**0.946**	**0.998**	**0.922**	**0.938**	**0.457**	**0.931**
Connectivity + Faults + VCoverage	**0.929**	**0.970**	**0.999**	**0.949**	**0.949**	0.616	**0.961**
Connectivity + Faults + Size	**0.934**	**0.964**	**1.000**	**0.948**	**0.954**	0.577	**0.968**
Connectivity + Pairwise + Size	0.903	**0.949**	**0.997**	**0.926**	**0.939**	**0.450**	**0.942**
Connectivity + Size + VCoverage	0.521	0.626	0.919	0.548	0.665	**0.144**	0.570
Dissimilarity + Faults + Pairwise	**0.915**	**0.957**	**0.998**	**0.934**	**0.942**	**0.496**	**0.947**
Dissimilarity + Faults + Size	**0.936**	**0.968**	**0.999**	**0.953**	**0.955**	0.620	**0.971**
Dissimilarity + Faults + VCoverage	**0.940**	**0.963**	**0.998**	**0.957**	**0.960**	0.637	**0.968**
Dissimilarity + Pairwise + Size	**0.895**	**0.950**	**0.998**	**0.921**	**0.931**	**0.391**	**0.926**
Dissimilarity + Size + VCoverage	0.572	0.680	**0.953**	0.622	0.742	**0.123**	0.628
Faults + Pairwise + Size	**0.914**	**0.958**	**0.999**	**0.938**	**0.946**	**0.479**	**0.953**
Faults + Pairwise + VCoverage	**0.905**	**0.951**	**0.997**	**0.930**	**0.938**	**0.442**	**0.935**
Faults + Size + VCoverage	**0.937**	**0.968**	**0.999**	**0.956**	**0.959**	0.616	**0.969**
Pairwise + Size + VCoverage	**0.887**	**0.946**	**0.997**	**0.914**	**0.931**	**0.447**	**0.932**

functional objectives, both single and combined. Table 15 shows the values of the $\widehat{A}_{12}$ effect size. A majority of cells show $\widehat{A}_{12}$ values under 0.5 indicating that non-functional objectives are in general more effective than functional objectives for test case prioritization in HCSs. As observed in Experiment 2 and 3, the objective `Faults`, and those combinations including it consistently show the largest differences. Also, as observed in Experiment 1, the objective `Pairwise` is consistently outperformed by all non-functional objectives, but it provides the best results in favour of functional objectives when combined with others.

7.7. Experiment 5. Algorithm comparison

In this experiment, we compared the performance of NSGA-II with a random search algorithm and a deterministic test case prioritization algorithm. The experimental setup and results of both comparisons are described in the following sections.

7.7.1. Comparison with random search

The pseudo-code of our implementation of Random Search (RS) algorithm is described in Algorithm 1. The algorithm takes an input attributed feature model *afm* and returns a set of test suites optimizing the target objectives. The algorithm has two configuration parameters: the number of iterations to be performed *nIterations*, and the maximum size of the random test suites to be generated *maxSize*. We set *maxSize* to the ceiling for the average size of the pairwise suites generated by *CASA* (13 test cases). Regarding the value of *nIterations*, we set its values to 5000, in order to ensure a

Algorithm 1 Random search algorithm

```
1:  procedure RS($afm$)
2:      $i \leftarrow 1$
3:      $pFront \leftarrow \{\}$
4:      while $i \leq nIterations$ do
5:          $sol \leftarrow randomSuite(afm, maxSize)$
6:          if $isNotDomiated(sol, pFront)$ then
7:              $pFront \leftarrow notDominated(pFront, sol) \bigcup sol$
8:          end if
9:          $i \leftarrow i + 1$
10:     end while
11:     return $pFront$
12: end procedure
```

Table 13
$\widehat{A}_{12}$ values for combinations of functional objectives vs. mixed combinations of functional and non–functional prioritization objectives. Cells revealing very large statistical differences are highlighted using dark grey (in favour of the column) or light grey (in favour of the row). Values in boldface reveal statistically significant differences (the p-value with Holm's correction is < 0.05).

Mixed Multi-Objective	Functional Multi-Objective									
	Con+Dis	Con+Pai	Con+VCo	Dis+Pai	Dis+VCo	Pai+VCo	Con+Dis+Pai	Con+Dis+VCo	Con+Pai+VCo	Dis+Pai+VCo
Changes + Con	0.556	0.267	0.529	0.221	0.559	0.386	0.203	0.547	0.284	0.208
Changes + Dis	0.518	0.242	0.489	0.198	0.531	0.363	0.187	0.513	0.268	0.199
Changes + Pai	0.800	0.523	0.792	0.463	0.827	0.703	0.418	0.810	0.551	0.438
Changes + VCo	0.536	0.245	0.488	0.211	0.541	0.338	0.200	0.517	0.248	0.201
Con + Faults	0.900	0.738	0.921	0.713	0.938	0.893	0.667	0.909	0.753	0.683
Con + Size	0.573	0.242	0.534	0.189	0.580	0.388	0.168	0.571	0.271	0.178
Dis + Faults	0.915	0.823	0.947	0.811	0.956	0.919	0.778	0.924	0.803	0.769
Dis + Size	0.463	0.208	0.424	0.176	0.467	0.294	0.156	0.442	0.216	0.162
Faults + Pai	0.864	0.641	0.878	0.600	0.910	0.830	0.560	0.879	0.676	0.580
Faults + VCo	0.906	0.780	0.932	0.754	0.956	0.907	0.721	0.916	0.767	0.719
Pai + Size	0.834	0.571	0.849	0.519	0.878	0.790	0.473	0.854	0.637	0.503
Size + VCo	0.544	0.239	0.515	0.196	0.549	0.363	0.175	0.537	0.272	0.180
Changes + Con + Dis	0.528	0.253	0.492	0.220	0.540	0.367	0.198	0.519	0.270	0.214
Changes + Con + Faults	0.888	0.730	0.917	0.694	0.940	0.877	0.654	0.900	0.736	0.668
Changes + Con + Pai	0.849	0.618	0.864	0.563	0.896	0.798	0.508	0.869	0.651	0.543
Changes + Con + Size	0.602	0.281	0.559	0.238	0.606	0.420	0.217	0.592	0.318	0.227
Changes + Con + VCo	0.585	0.293	0.540	0.256	0.586	0.394	0.244	0.573	0.299	0.237
Changes + Dis + Faults	0.898	0.758	0.927	0.736	0.947	0.898	0.688	0.908	0.761	0.701
Changes + Dis + Pai	0.849	0.621	0.864	0.570	0.896	0.802	0.524	0.865	0.648	0.543
Changes + Dis + Size	0.557	0.224	0.497	0.189	0.548	0.323	0.181	0.526	0.231	0.171
Changes + Dis + VCo	0.674	0.373	0.645	0.319	0.690	0.517	0.296	0.669	0.389	0.298
Changes + Faults + Pai	0.877	0.708	0.902	0.672	0.933	0.854	0.622	0.890	0.707	0.643
Changes + Faults + VCo	0.901	0.754	0.933	0.735	0.946	0.898	0.687	0.912	0.761	0.705
Changes + Pai + Size	0.876	0.706	0.900	0.662	0.924	0.848	0.624	0.893	0.716	0.634
Changes + Pai + VCo	0.859	0.650	0.874	0.603	0.906	0.823	0.549	0.876	0.678	0.580
Changes + Size + VCo	0.581	0.242	0.549	0.189	0.590	0.395	0.171	0.585	0.277	0.171
Con + Dis + Faults	0.911	0.798	0.944	0.783	0.957	0.914	0.742	0.922	0.780	0.745
Con + Dis + Size	0.570	0.268	0.520	0.227	0.569	0.388	0.213	0.554	0.278	0.215
Con + Faults + Pai	0.868	0.689	0.891	0.645	0.920	0.844	0.604	0.887	0.700	0.619
Con + Faults + Size	0.906	0.772	0.935	0.753	0.954	0.909	0.713	0.917	0.770	0.722
Con + Faults + VCo	0.900	0.757	0.925	0.742	0.938	0.894	0.706	0.917	0.772	0.718
Con + Pai + Size	0.875	0.698	0.888	0.646	0.927	0.846	0.604	0.888	0.700	0.620
Con + Size + VCo	0.519	0.268	0.495	0.236	0.537	0.372	0.216	0.524	0.284	0.220
Dis + Faults + Pai	0.886	0.719	0.911	0.689	0.936	0.871	0.643	0.898	0.727	0.664
Dis + Faults + Size	0.907	0.779	0.941	0.764	0.951	0.907	0.723	0.917	0.779	0.734
Dis + Faults + VCo	0.909	0.792	0.936	0.779	0.950	0.906	0.736	0.917	0.777	0.738
Dis + Pai + Size	0.866	0.655	0.883	0.602	0.914	0.838	0.543	0.878	0.688	0.576
Dis + Size + VCo	0.584	0.257	0.540	0.213	0.588	0.399	0.199	0.578	0.284	0.198
Faults + Pai + Size	0.883	0.716	0.915	0.676	0.936	0.876	0.628	0.898	0.723	0.649
Faults + Pai + VCo	0.872	0.691	0.901	0.647	0.924	0.856	0.595	0.885	0.707	0.619
Faults + Size + VCo	0.908	0.783	0.937	0.765	0.950	0.910	0.729	0.923	0.780	0.737
Pai + Size + VCo	0.871	0.663	0.891	0.632	0.911	0.835	0.586	0.876	0.691	0.606

fair comparison with NSGA-II, which performs 50 iterations with a population of 100 individuals ($50 * 100 = 5000$ evaluations).

Three functions are invoked in the pseudo-code of Algorithm 1: i) *randomSuite*, that generates a suite of random size (between 1 and *maxSize*), comprising of random products generated using the PLEDGE tool (Henard et al., 2014); ii) *isNotDominated* that checks if the solution *sol* provided as parameter is not dominated by any solution in the Pareto front estimation *pFront*; and iii) *notDominated* that returns the solutions from *pFront* that are dominated by *sol*.

Table 16 shows the results of the comparison. For each group of objectives and algorithm under comparison, the table shows the execution time, average and maximum APFD value of the test suites in the Pareto front. The best values of each metric on each row are highlighted in boldface. As illustrated, NSGA-II outperforms RS in 58 out of the 63 combinations of objectives in terms of average APDF value. Interestingly, there is not a clear winner in terms of execution time: RS was faster in 36 out of the 63 objective groups (57.2%) meanwhile NSGA-II achieved lower average execution times in 42.8% of the objectives. We found that the overhead in the RS algorithm was due to the cost of generating random valid test cases. In terms of fault detection, NSGA-II detected slightly more faults than RS. However, both algorithms detected more than 99.5% of the emulated faults and thus the differences are not significant.

Table 17 shows the average, standard deviation and maximum values of the hypervolume achieved by each algorithm and objective group under comparison. The hypervolume is the n-dimensional space contained by a set of solutions with respect to a reference point (Beume et al., 2009). The hypervolume metric is widely used to compare the performance of multi-objective algorithms, where solutions with a larger hypervolume provide a better trade-offs among objectives than solutions with a smaller hypervolume. The best values of each metric on each row are highlighted in boldface. NSGA-II provides better results for the majority of executions both in terms of average hypervolume (38 out of 63 objective sets) and maximum hypervolume (41 out of 63 objective sets). Interestingly, RS provides better results than NSGA-II for a fair percentage of objective sets. This is probably because NSGA-II mainly focuses on re-ordering the test cases in the initial population, while RS performs a wider exploration of the search space generating solutions with different (random) test cases. It is noteworthy, however, that a larger hypervolume does not necessarily implies a better rate of fault detection, as observed in Table 16.

In summary, NSGA-II outperforms RS in accelerating the detection of faults in HCSs since NSGA-II provides higher average and maximum APFD values, it usually generates Pareto fronts approximations with better hypervolumes, and both algorithms have similar executions times.

7.7.2. Comparison with a coverage-based prioritization algorithm

For a further validation, we compared the results of our MOEA with the deterministic coverage-based prioritization algorithm for software product lines proposed by Sánchez et al. (2014). The algorithm takes an attributed feature model *afm* as input,

Table 14
$\widehat{A_{12}}$ values for combinations of non-functional objectives vs. mixed combinations of functional and non–functional prioritization objectives. Cells revealing very large statistical differences are highlighted using dark grey (in favour of the column) or light grey (in favour of the row). Values in boldface reveal statistically significant differences (the *p*-value with Holm's correction is < 0.05).

Mixed Multi-Objective	**Non-Functional Multi-Objective**			
	Changes + Faults	Changes + Size	Faults + Size	Changes + Faults + Size
Changes + Connectivity	**0.098**	0.549	**0.087**	**0.116**
Changes + Dissimilarity	**0.100**	0.517	**0.093**	**0.110**
Changes + Pairwise	**0.239**	**0.807**	**0.222**	**0.273**
Changes + VCoverage	**0.117**	0.543	**0.105**	**0.126**
Connectivity + Faults	0.470	**0.924**	0.438	0.514
Connectivity + Size	**0.063**	0.569	**0.055**	**0.083**
Dissimilarity + Faults	**0.643**	**0.947**	0.571	**0.639**
Dissimilarity + Size	**0.078**	0.451	**0.072**	**0.095**
Faults + Pairwise	**0.343**	**0.882**	**0.318**	**0.391**
Faults + VCoverage	0.555	**0.932**	0.473	0.548
Pairwise + Size	**0.269**	**0.853**	**0.251**	**0.314**
Size + VCoverage	**0.083**	0.535	**0.073**	**0.094**
Changes + Connectivity + Dissimilarity	**0.113**	0.528	**0.100**	**0.118**
Changes + Connectivity + Faults	0.454	**0.917**	**0.408**	0.480
Changes + Connectivity + Pairwise	**0.290**	0.865	**0.269**	**0.350**
Changes + Connectivity + Size	**0.113**	0.595	**0.108**	**0.130**
Changes + Connectivity + VCoverage	0.165	0.584	**0.149**	0.167
Changes + Dissimilarity + Faults	0.528	**0.929**	**0.469**	0.537
Changes + Dissimilarity + Pairwise	**0.327**	**0.866**	**0.302**	**0.359**
Changes + Dissimilarity + Size	**0.122**	0.547	**0.101**	**0.113**
Changes + Dissimilarity + VCoverage	**0.166**	0.680	**0.136**	**0.173**
Changes + Faults + Pairwise	**0.443**	**0.904**	**0.392**	**0.446**
Changes + Faults + VCoverage	0.505	**0.929**	**0.458**	0.523
Changes + Pairwise + Size	**0.435**	**0.903**	**0.394**	**0.467**
Changes + Pairwise + VCoverage	**0.367**	0.882	**0.330**	**0.401**
Changes + Size + VCoverage	**0.077**	0.578	**0.067**	**0.088**
Connectivity + Dissimilarity + Faults	0.598	**0.941**	0.535	**0.589**
Connectivity + Dissimilarity + Size	**0.097**	0.563	**0.095**	**0.124**
Connectivity + Faults + Pairwise	**0.424**	**0.891**	**0.378**	**0.434**
Connectivity + Faults + Size	0.548	**0.933**	0.476	0.546
Connectivity + Faults + VCoverage	0.565	**0.929**	0.520	0.578
Connectivity + Pairwise + Size	**0.404**	**0.897**	**0.348**	**0.418**
Connectivity + Size + VCoverage	**0.111**	0.521	**0.106**	**0.134**
Dissimilarity + Faults + Pairwise	**0.444**	**0.913**	**0.403**	**0.469**
Dissimilarity + Faults + Size	0.567	**0.942**	0.515	0.583
Dissimilarity + Faults + VCoverage	0.607	**0.942**	0.548	0.595
Dissimilarity + Pairwise + Size	**0.295**	**0.884**	**0.279**	**0.364**
Dissimilarity + Size + VCoverage	**0.096**	0.584	**0.082**	**0.111**
Faults + Pairwise + Size	**0.419**	**0.914**	**0.373**	**0.452**
Faults + Pairwise + VCoverage	**0.388**	**0.901**	**0.348**	**0.419**
Faults + Size + VCoverage	0.590	**0.937**	0.531	0.586
Pairwise + Size + VCoverage	**0.388**	**0.893**	**0.359**	**0.434**

Table 15
$\widehat{A_{12}}$ values for functional vs. non–functional prioritization objectives. Cells revealing very large statistical differences are highlighted in dark grey (in favour of the column). Values in boldface reveal statistically significant differences (the *p*-value with Holm's correction is < 0.05).

Functional Objectives	**Non-Functional Objectives**						
	Changes	Faults	Size	Changes + Faults	Changes + Size	Faults + Size	Changes + Faults + Size
Connectivity	**0.675**	**0.083**	0.550	**0.060**	0.509	**0.057**	**0.078**
Dissimilarity	0.583	**0.037**	0.442	**0.036**	0.403	**0.035**	**0.037**
Pairwise	0.166	**0.000**	**0.083**	**0.000**	**0.076**	**0.000**	**0.000**
VCoverage	0.637	**0.061**	**0.509**	**0.051**	**0.463**	**0.045**	**0.056**
Connectivity + Dissimilarity	0.657	**0.115**	0.536	**0.093**	0.493	**0.087**	**0.108**
Connectivity + Pairwise	**0.895**	**0.278**	**0.851**	0.212	**0.802**	**0.201**	**0.265**
Connectivity + VCoverage	**0.704**	**0.088**	0.589	**0.070**	0.541	**0.054**	**0.080**
Dissimilarity + Pairwise	**0.912**	**0.314**	**0.887**	**0.242**	**0.838**	**0.225**	**0.290**
Dissimilarity + VCoverage	0.654	**0.064**	0.534	**0.039**	0.497	**0.039**	**0.058**
Pairwise + VCoverage	**0.848**	**0.116**	0.741	**0.090**	0.697	**0.081**	**0.108**
Connectivity + Dissimilarity + Pairwise	**0.915**	**0.363**	**0.898**	**0.289**	0.849	**0.274**	**0.342**
Connectivity + Dissimilarity + VCoverage	0.688	**0.104**	0.553	**0.089**	0.516	**0.079**	**0.097**
Connectivity + Pairwise + VCoverage	**0.891**	**0.273**	**0.831**	**0.234**	0.792	**0.221**	**0.261**
Dissimilarity + Pairwise + VCoverage	**0.918**	**0.342**	**0.896**	**0.286**	0.848	**0.258**	**0.322**

Table 16
APFD values and execution times achieved by NSGA-II and Random Search (40 executions in total).

Objectives	NSGA-II			Random search		
	Ex.Time	Avg	Avg Max	Ex.Time	Avg	Avg Max
Connectivity	**77251.6**	**0.923**	**0.935**	119627.9	0.911	0.915
Dissimilarity	**75856.1**	**0.912**	**0.931**	112480.8	0.885	0.894
Pairwise	1478882.8	0.880	0.880	**714078.2**	**0.913**	**0.913**
VCoverage	**79107.7**	0.888	**0.933**	102457.4	**0.893**	0.918
Connectivity + Dissimilarity	**83767.4**	**0.912**	**0.937**	106497.4	0.898	0.914
Connectivity + Pairwise	1421967.3	**0.938**	0.951	**709398.8**	0.933	**0.952**
Connectivity + VCoverage	**78199.5**	**0.914**	**0.934**	120115.3	0.891	0.899
Dissimilarity + Pairwise	1400685.8	**0.941**	0.952	**715289.6**	0.931	**0.953**
Dissimilarity + VCoverage	**77182.5**	**0.916**	**0.936**	128498.2	0.900	0.919
Pairwise + VCoverage	1453296.7	**0.936**	0.948	**714764.8**	0.927	**0.952**
Connectivity + Dissimilarity + Pairwise	1414754.9	**0.929**	**0.954**	**711827.3**	0.927	0.953
Connectivity + Dissimilarity + VCoverage	**80922.4**	0.906	0.929	111031.5	**0.917**	**0.933**
Connectivity + Pairwise + VCoverage	1434254.1	**0.932**	0.949	**720098.3**	0.928	**0.953**
Dissimilarity + Pairwise + VCoverage	1405714.7	**0.938**	**0.952**	**715958.6**	0.926	0.952
Changes	**154553.9**	0.902	**0.915**	255249.5	**0.907**	0.907
Faults	**266983.7**	**0.955**	**0.955**	311060.0	0.953	0.953
Size	**147161.1**	**0.917**	**0.931**	267076.5	**0.917**	0.917
Changes + Faults	**267046.7**	**0.953**	**0.955**	315937.0	0.933	0.954
Changes + Size	**150685.2**	**0.918**	**0.942**	250050.4	0.909	0.942
Faults + Size	**266313.0**	**0.955**	**0.956**	317911.9	0.941	0.954
Changes + Faults + Size	**267405.4**	**0.951**	**0.955**	319803.3	0.935	0.954
Changes + Connectivity	**89517.2**	**0.916**	0.939	134150.5	0.907	**0.944**
Changes + Dissimilarity	**89095.0**	**0.907**	0.939	143702.3	0.905	**0.940**
Changes + Pairwise	1174272.6	**0.937**	**0.952**	**602875.9**	0.928	0.951
Changes + VCoverage	**86759.7**	**0.917**	0.936	133983.3	0.906	**0.950**
Connectivity + Faults	248148.2	**0.953**	**0.955**	**221543.1**	0.944	0.954
Connectivity + Size	**95670.1**	**0.915**	0.944	134395.5	0.910	**0.948**
Dissimilarity + Faults	253215.6	**0.954**	**0.955**	**219950.5**	0.940	0.954
Dissimilarity + Size	**85769.7**	0.903	0.923	129575.3	**0.913**	**0.946**
Faults + Pairwise	1328146.5	**0.943**	0.954	**681334.3**	0.938	**0.954**
Faults + VCoverage	258827.7	**0.955**	0.956	**211328.7**	0.944	0.954
Pairwise + Size	1190707.6	**0.942**	**0.953**	**608338.8**	0.932	0.953
Size + VCoverage	**82624.5**	**0.913**	0.935	148241.0	0.908	**0.950**
Changes + Connectivity + Dissimilarity	**90333.7**	**0.912**	0.939	141020.4	0.899	**0.947**
Changes + Connectivity + Faults	242090.7	**0.951**	**0.955**	**213817.3**	0.932	0.955
Changes + Connectivity + VCoverage	**91098.6**	**0.915**	0.943	147589.7	0.904	**0.942**
Changes + Dissimilarity + Faults	258500.8	**0.952**	**0.955**	**237274.7**	0.932	0.954
Changes + Dissimilarity + VCoverage	**79023.9**	**0.912**	**0.948**	149348.8	0.887	0.946
Changes + Faults + Pairwise	1314596.2	**0.942**	0.954	**685275.2**	0.929	**0.955**
Changes + Faults + VCoverage	253188.5	**0.952**	**0.955**	**227043.4**	0.927	0.954
Changes + Pairwise + Connectivity	1159383.0	**0.935**	0.954	**609231.8**	0.926	**0.954**
Changes + Pairwise + Dissimilarity	1156172.8	**0.935**	**0.953**	**608587.6**	0.923	0.953
Changes + Pairwise + VCoverage	1154166.3	**0.936**	0.954	**601289.3**	0.924	**0.954**
Changes + Size + Connectivity	**81688.1**	**0.911**	0.939	131841.9	0.903	**0.947**
Changes + Size + Dissimilarity	**87032.2**	**0.913**	0.942	131280.7	0.913	**0.950**
Changes + Size + Pairwise	1185686.4	**0.940**	0.955	**609731.4**	0.925	**0.955**
Changes + Size + VCoverage	**93134.0**	**0.908**	0.940	138131.6	0.906	**0.952**
Connectivity + Dissimilarity + Faults	254147.6	**0.954**	**0.955**	**232744.1**	0.938	0.954
Connectivity + Dissimilarity + Size	**77954.7**	**0.910**	0.944	130562.3	0.902	**0.945**
Connectivity + Faults + Pairwise	1324924.1	**0.942**	0.954	**689787.5**	0.935	**0.955**
Connectivity + Faults + Size	254000.3	**0.954**	**0.955**	**221594.3**	0.925	0.954
Connectivity + Faults + VCoverage	245731.1	**0.951**	**0.955**	**217372.5**	0.941	0.954
Connectivity + Size + Pairwise	1167713.3	**0.938**	0.955	**604217.1**	0.924	**0.955**
Connectivity + Size + VCoverage	**83282.2**	**0.914**	0.944	156937.0	0.904	**0.951**
Dissimilarity + Faults + Pairwise	1324060.1	**0.945**	**0.955**	**677402.6**	0.929	0.954
Dissimilarity + Faults + Size	243985.7	**0.952**	**0.955**	**226250.4**	0.931	0.954
Dissimilarity + Faults + VCoverage	251145.3	**0.952**	**0.955**	**225094.4**	0.934	0.954
Dissimilarity + Pairwise + Size	1130349.7	**0.942**	**0.954**	**607200.3**	0.927	0.953
Dissimilarity + Size + VCoverage	**95906.8**	**0.912**	0.945	137942.8	0.906	**0.949**
Faults + Pairwise + VCoverage	1299913.7	**0.944**	**0.955**	**682123.5**	0.935	0.954
Faults + Pairwise + Size	1315391.8	**0.942**	**0.955**	**686569.3**	0.939	0.954
Faults + Size + VCoverage	246233.0	**0.953**	**0.956**	**218511.5**	0.924	0.955
Pairwise + Size + VCoverage	1169411.1	**0.939**	**0.955**	**603278.3**	0.923	0.953
Average	**552301.4**	**0.930**	**0.946**	**351709.2**	**0.920**	**0.945**

generates a pairwise suite from the model and re-arranges its products in descending order of pairwise coverage using bubble search. For the deterministic generation of a pairwise suite, we used the SPLCAT tool (Johansen et al., 2011). The pseudo-code of our implementation is described in Algorithm 2. Two functions are invoked in the pseudo-code of this algorithm: i) *ICPL* which represents the ICPL algorithm for generating pairwise suites as implemented by the tool SPLCAT, and *cov* that is equivalent to the *PairwiseCoverage* objective function defined in Section 6 assuming that the suite has a single test case, specified as a parameter.

Table 17
Comparison of hypervolumes obtained by NSGA-II and Random Search.

Objective	NSGA-II			Random search		
	Avg HV	StdDev HV	Max HV	Avg HV	StdDev HV	Max HV
Connectivity	**0.062**	0.031	**0.144**	0.026	**0.007**	0.060
Dissimilarity	**0.055**	0.024	**0.129**	0.036	**0.006**	0.050
Pairwise	0.151	**0.003**	0.157	**0.318**	0.004	**0.324**
VCoverage	**0.062**	0.026	**0.115**	0.021	**0.003**	0.033
Connectivity + Dissimilarity	**0.103**	0.040	**0.166**	0.068	**0.014**	0.107
Connectivity + Pairwise	0.217	0.034	0.303	**0.349**	**0.006**	**0.360**
Connectivity + VCoverage	**0.108**	0.047	**0.206**	0.049	**0.010**	0.071
Dissimilarity + Pairwise	0.206	0.032	0.261	**0.355**	**0.008**	**0.374**
Dissimilarity + VCoverage	**0.109**	0.039	**0.185**	0.060	**0.010**	0.089
Pairwise + VCoverage	0.218	0.026	0.257	**0.344**	**0.007**	**0.364**
Dissimilarity + Pairwise + VCoverage	0.269	0.049	0.356	**0.379**	**0.009**	**0.396**
Connectivity + Dissimilarity + Pairwise	0.281	0.062	**0.428**	**0.379**	**0.010**	0.398
Connectivity + Dissimilarity + VCoverage	**0.167**	0.062	**0.297**	0.083	**0.012**	0.105
Connectivity + Pairwise + VCoverage	0.278	0.052	0.393	**0.372**	**0.014**	**0.407**
Changes	**0.110**	0.025	**0.178**	0.082	**0.008**	0.103
Faults	0.157	0.017	0.188	**0.166**	**0.015**	**0.201**
Size	**0.065**	0.015	**0.093**	0.050	**0.003**	0.058
Changes + Faults	**0.260**	0.026	**0.308**	0.245	**0.015**	0.273
Changes + Size	**0.169**	0.044	**0.299**	0.133	**0.009**	0.160
Faults + Size	0.207	0.025	**0.251**	**0.213**	**0.017**	0.248
Changes + Faults + Size	**0.305**	0.040	**0.404**	0.290	**0.019**	0.330
Changes + Connectivity	**0.168**	0.039	**0.304**	0.113	**0.009**	0.137
Changes + Dissimilarity	**0.165**	0.038	**0.291**	0.115	**0.008**	0.131
Changes + Pairwise	0.245	0.028	0.311	**0.386**	**0.008**	**0.400**
Changes + VCoverage	**0.173**	0.037	**0.266**	0.106	**0.007**	0.124
Connectivity + Faults	**0.214**	0.041	**0.309**	0.197	**0.019**	0.254
Connectivity + Size	**0.129**	0.043	**0.256**	0.083	**0.006**	0.097
Dissimilarity + Faults	**0.199**	0.031	**0.295**	0.201	**0.016**	0.244
Dissimilarity + Size	**0.111**	0.028	**0.179**	0.087	**0.009**	0.106
Faults + Pairwise	0.293	0.019	0.360	**0.435**	**0.012**	**0.470**
Faults + VCoverage	**0.204**	0.040	**0.298**	0.194	**0.023**	0.249
Pairwise + Size	0.213	0.022	0.256	**0.360**	**0.004**	**0.373**
Size + VCoverage	**0.124**	0.033	**0.196**	0.077	**0.004**	0.092
Changes + Connectivity + Dissimilarity	**0.212**	0.054	**0.385**	0.147	**0.014**	0.182
Changes + Connectivity + Faults	**0.317**	0.044	**0.416**	0.273	**0.019**	0.315
Changes + Connectivity + Pairwise	0.324	0.039	0.415	**0.418**	**0.009**	**0.433**
Changes + Connectivity + Size	**0.226**	0.043	**0.341**	0.162	**0.011**	0.188
Changes + Connectivity + VCoverage	**0.221**	0.053	**0.346**	0.134	**0.009**	0.158
Changes + Dissimilarity + Faults	**0.297**	0.042	**0.421**	0.283	**0.018**	0.319
Changes + Dissimilarity + Pairwise	0.307	0.048	0.425	**0.419**	**0.009**	**0.445**
Changes + Dissimilarity + Size	**0.211**	0.041	**0.303**	0.164	**0.009**	0.181
Changes + Dissimilarity + VCoverage	**0.212**	0.040	**0.334**	0.145	**0.012**	0.189
Changes + Faults + Pairwise	0.374	0.034	0.452	**0.499**	**0.012**	**0.530**
Changes + Faults + VCoverage	**0.303**	0.043	**0.391**	0.271	**0.017**	0.303
Changes + Pairwise + Size	0.300	0.051	0.417	**0.423**	**0.007**	**0.435**
Changes + Pairwise + VCoverage	0.323	0.039	0.410	**0.412**	**0.008**	**0.430**
Changes + Size + VCoverage	**0.226**	0.051	**0.364**	0.160	**0.009**	0.183
Connectivity + Dissimilarity + Faults	**0.256**	0.044	**0.366**	0.237	**0.021**	0.279
Connectivity + Dissimilarity + Size	**0.166**	0.046	**0.249**	0.115	**0.012**	0.138
Connectivity + Faults + Pairwise	0.340	0.054	0.444	**0.462**	**0.014**	**0.489**
Connectivity + Faults + Size	**0.266**	0.046	**0.368**	0.249	**0.019**	0.288
Connectivity + Faults + VCoverage	**0.252**	0.057	**0.394**	0.218	**0.018**	0.258
Connectivity + Pairwise + Size	0.281	0.053	0.388	**0.391**	**0.007**	**0.410**
Connectivity + Size + VCoverage	**0.184**	0.046	**0.288**	0.107	**0.009**	0.127
Dissimilarity + Faults + Pairwise	0.342	0.040	0.422	**0.471**	**0.015**	**0.514**
Dissimilarity + Faults + Size	**0.265**	0.044	**0.435**	0.251	**0.020**	0.282
Dissimilarity + Faults + VCoverage	**0.248**	0.058	**0.403**	0.231	**0.020**	0.290
Dissimilarity + Pairwise + Size	0.267	0.034	0.359	**0.394**	**0.008**	**0.412**
Dissimilarity + Size + VCoverage	**0.166**	**0.030**	**0.215**	0.113	0.010	0.157
Faults + Pairwise + Size	0.339	0.021	0.382	**0.473**	**0.013**	**0.497**
Faults + Pairwise + VCoverage	0.345	0.029	0.392	**0.456**	**0.009**	**0.479**
Faults + Size + VCoverage	**0.263**	0.045	**0.375**	0.246	**0.017**	0.273
Pairwise + Size + VCoverage	0.275	0.030	0.350	**0.390**	**0.007**	**0.404**

The APFD value achieved by the coverage-based algorithm is 0.946, which is less than the average AFPD value of the best solution in the Pareto front found by NSGA-II for 38 out of the 63 objective sets, i.e. column "Avg Max" in Table 16. More importantly, NSGA-II achieved better results than the coverage-based algorithm in all the 40 executions for 29 out of the 63 objective sets. This means that, for almost half of the objective sets, our algorithm was always better than the coverage-based algorithm. It is noteworthy, however, that the differences between the APFD values of NSGA-II and the coverage-based algorithm are small, probably due to the size of the generated suites (13 test cases on average). We conjecture that these differences would be larger when dealing with

Algorithm 2 Coverage-based prioritization algorithm

```
1:  procedure PWCMax($afm$)
2:      $suite \leftarrow ICPL(afm)$
3:      $size \leftarrow size(suite)$
4:      repeat
5:          $swapped \leftarrow false$
6:          for $i \leftarrow 2, size$ do
7:              if $cov(suite[i-1], afm) < cov(suite[i], afm)$ then
8:                  $aux \leftarrow suite[i]$
9:                  $suite[i] \leftarrow suite[i-1]$
10:                 $suite[i-1] \leftarrow aux$
11:                 $swapped \leftarrow true$
12:             end if
13:         end for
14:     until $\neg swapped$
15:     return $suite$
16: end procedure
```

bigger test suites (e.g. 3-wise), but this is something that requires further research. In terms of execution time, the coverage-based algorithm was executed in less than 2 seconds, which makes it appropriate when fast response times are required.

7.8. Discussion

We now summarize the results and what they tell us about the research questions.

RQ1: Mono vs. multi–objective prioritization using functional objectives. Experiment 1 revealed that multi–objective prioritization outperforms mono-objective prioritization when using functional objectives. The superiority was especially noticeable in the comparison with Pairwise as a single-objective, which consistently achieved the worse rate of fault detection. Interestingly, however, Pairwise performed very well when combined with other functional objectives. In the light of these results, RQ1 is answered as follows:

Multi–objective prioritization using functional objectives is more effective than mono–objective prioritization with functional objectives in accelerating the detection of faults in HCSs.

RQ2: Mono vs. multi–objective prioritization using non–functional objectives. The results of experiment 2 showed significant differences in favour of multi–objective prioritization over mono–objective prioritization using non–functional objectives. It also revealed a clear superiority of the objective function Faults, single or in combination with other objectives, over the rest of the non-functional objectives. We conjecture that this result could be caused by the nature of the case study. In particular, we used the bugs detected in Drupal v7.22 to accelerate the detection of faults in Drupal v7.23. Being two consecutive versions of the framework, we found that some of the faults in Drupal v7.22 remained in Drupal v7.23, which means that the prioritization could be overfitted. While this is a realistic scenario, we think the results could not be generalizable to non–consecutive versions of the framework and thus the results must be taken with caution. Based on the global results, however, RQ2 is answered as follows:

Multi–objective prioritization using non–functional objectives is, in general, more effective than mono–objective prioritization with non–functional objectives in accelerating the detection of faults in HCSs.

RQ3: Combination of functional and non–functional objectives. Experiment 3 revealed that the multi–objective prioritization using functional and non–functional objectives outperform prioritization driven by a single objective, either functional or non-functional. Similarly, mixed combinations of objectives achieved better results than the combination of functional objectives, but slightly worse than the combination of non-functional objectives. It was observed that the objective Faults has a key influence in the results of prioritization, probably explained, as detailed above, by the use of two consecutive versions of the framework. It is remarkable, however, that the best overall results were achieved by the combination of the functional objective Dissimilarity and the non-functional objective Faults. In the light of these results, RQ3 is answered as follows:

Multi–objective prioritization driven by functional and non–functional objectives perform better than mono–objective prioritization, and better than multi–objective prioritization using functional objectives, but slightly worse than multi–objective prioritization using non–functional objectives in accelerating the detection of faults in HCSs.

RQ4: Functional vs non–functional objectives. The results of experiment 4 show a clear dominance of non–functional objectives over functional objectives, especially noticeable when these are combined in a multi-objective perspective. This is consistent with our previous results on mono–objective comparison of functional and non–functional objectives (Sánchez et al., 2015b). Based on these results, RQ4 is answered as follows:

Non–functional prioritization objectives are more effective in accelerating the detection of faults in HCSs than functional objectives, especially when they are combined in a multi-objective perspective.

RQ5: What is the performance of the proposed MOEA compared to related algorithms?. The results of experiment 5 reveal that NSGA-II outperforms Random Search and coverage-based prioritization in terms of hypervolume and rate of detected faults. Although the coverage-based algorithm provides solutions with a good detection speed in a short execution time, our adaptation of NSGA-II outperforms it with 60% of the objective sets under comparison. More importantly, our algorithm achieved better results than the coverage-based approach in all the executions for 29 out of 63 objective groups. This means that, for almost half of the objective sets, NSGA-II was always better than the coverage-based algorithm. Based on these results, RQ5 is answered as follows:

NSGA-II outperforms random search and coverage-based prioritization in accelerating the detection of faults in HCSs, although coverage-based prioritization is significantly faster.

8. 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. Inadequate parameter setting is a common internal validity threat. In this paper, we used standard parameter values for the NSGA-II algorithm (Deb et al., 2002). Furthermore, to consider the effect of stochasticity, the algorithm was executed multiple times with each combination of objective functions and their results analysed using statistical tests.

For the evaluation of our approach we seeded our algorithm with pairwise test suites from the Drupal feature model in Fig. 3. Each pairwise was composed of 13 test cases on average. The output test suites generated by the prioritization algorithms under study had a similar size. Due to the small number of test cases in the suites, we found that the absolute differences among the APFD values achieved by different algorithms and objectives where small, which may suggest that their performance is similar. We remark, however, that the observed differences are statistically significant showing the superiority of our approach. Results also suggest that the observed differences would be noticeably larger when prioritizing larger test suites, but that is something that requires further research. Finally, we may remark that the main goal of our work is to compare the effectiveness of different prioritization objectives for HCSs, rather than comparing the performance of different prioritization algorithms.

External validity. This can be mainly divided into limitations of the approach and generalizability of the conclusions. Regarding the limitations, the Drupal feature model and their attributes were manually mined from different sources and therefore they could slightly differ from their real shape (Sánchez et al., 2015b). Other risk for the validity of our work is that a number of the faults in Drupal v7.22 remained in Drupal v7.23, which may introduce a bias in the fault–driven prioritization. Note, however, that this is a realistic scenario since it is common in open-source projects that unfixed faults affect several versions of the system.

The statistical and prioritization results reported are based on a single case study and thus cannot be generalized to other HCSs. Nevertheless, our results show the efficacy of using combinations of functional and non-functional goals in a multi-objective problem as good drivers for test case prioritization in open–source HCSs as the Drupal framework.

9. Related work

In this section we summarize the pieces of work that most closely relate to us. We divide them into HCSs testing and general software testing.

HCSs testing. Within the context of HCSs, there has been a stark and recent interest in the area of *Software Product Lines (SPLs)* testing as evidenced by several systematic mapping studies (e.g. da Mota Silveira Neto et al., 2011; do Carmo Machado et al., 2014; Engström and Runeson, 2011). These studies focus on categorizing SPL approaches along criteria within the realm of SPLs such as handling of variability and variant binding times, as well as other aspects like test organization and process. Among their findings, all identified *Combinatorial Interaction Testing (CIT)*, as the leading approach of SPL testing. Recent work by Yilmaz et al. divides CIT approaches in two big phases (Yilmaz et al., 2014): *i) what phase* whose purpose is to select a group of products for testing, and *ii) how phase* whose purpose is to perform the test on the selected products. When CIT is applied to SPLs the goal is to obtain a sample of products, i.e. feature combinations, as representative exemplars on which to perform the testing tasks. Recently, we performed a systematic mapping study to delve into more detail on the subject (Lopez-Herrejon et al., 2015). This mapping study identified over forty different approaches that rely on diverse techniques, such as genetic and greedy algorithms, that were evaluated also with multiple problem domains of different characteristics. Among other findings, this study revealed that the large majority of approaches focuses only on computing the samples of products based purely on variability models (e.g. feature models), that is, the main focus is the *what phase* of CIT.

In addition, most of the approaches found focus on pairwise testing and only few have higher coverage strengths (i.e. $t>3$). A salient example is the work of Henard et al. who compute covering arrays of up to 6 features (i.e. $t=6$) for some of the largest variability models available (Henard et al., 2014). They employ an evolutionary algorithm with an objective function based on Jaccard's dissimilarity metric, and compute samples of fixed size within certain fixed constraints regarding computation time and number of iterations. For their larger case studies and for the smaller case studies from 3-wise upwards, they analyze the effectiveness of their approach based on the *estimated* number of feature interactions (i.e. t-sets) as the actual number is intractable to compute. To the best of our knowledge, this and other approaches that consider higher coverage strengths for SPLs do not provide empirical evidence that higher coverage strengths are in fact more effective for fault detection to actually pay off for their typically more expensive computation. This is so, because for their analysis they do not consider actual faults found in actual systems like our work does with Drupal.

Our study also revealed very few instances of prioritization in SPL testing. Salient among them is our previous work that studied different approaches to prioritize test suites obtained with a single-objective greedy algorithm and their impact for fault detection (Sánchez et al., 2014). Another approach was proposed by Johansen et al. who attach arbitrary weights to products to reflect for instance market relevance and compute the covering arrays using a greedy approach (Johansen et al., 2012). This approach was formalized by Lopez-Herrejon et al. who also propose a parallel genetic algorithm that achieved better performance in a larger number of case studies (Lopez-Herrejon et al., 2014). In sharp contrast with these approaches, our current work employs a multi-objective algorithm to analyze different combinations of metrics and their impact for detecting faults in a real-world case study.

Devroey et al. proposed a model-based testing approach to prioritize SPL testing (Devroey et al., 2014a). Their approach relies on a feature model, a feature transition system (a transition system enhanced with feature information to indicate what products can execute a transition), and a usage model with the probabilities of executing relevant transitions. This approach computes the probabilities of execution of products which could be used to prioritize their testing. It was empirically evaluated on logged information of a web-system. In contrast with our work, their prioritization is based on statistical probabilities per product (not on functional and non-functional data), and does not consider multiple optimization objectives.

Recent work by Wang et al. use a preference indicator to assign different weights to objective functions depending on their relevance for the users (Wang et al., 2015b). They modify the NSGA-II algorithm by substituting its crowd distance indicator with their preference indicator. In contrast with our work, their focus is on finding more effective weight assignments that reflect user preference rather than focusing on different combinations of objective functions to speed up fault localization. Also recent work by Epitropakis et al. study multi-objective test case prioritization but focus on standard software systems (i.e. not on HCSs), and use a different set of objective functions (Epitropakis et al., 2015).

Similar to test prioritization, only a few studies have been conducted on employing multi-objective optimization of SPL testing. The work by Wang et al. describe an approach to minimize test suites using three objectives (Wang et al., 2013), namely, test minimization percentage, pairwise coverage, and fault detection capability that works by assigning weights to these objectives – a process called *scalarization* (Zitzler, 2012). Their work was extended to generate weights from a uniform distribution while still satisfying the user-defined constraints (Wang et al., 2014a). More recently, they have extended this work to consider several multi-objective algorithms that also includes resource awareness (Wang et al., 2016).

Work by Henard et al. presents an ad-hoc multi-objective algorithm whose fitness functions are maximizing coverage, minimizing test suite size, and minimizing cost (Henard et al., 2013). However, they also use scalarization. This is important because there is an extensive body of work on the downsides of scalarization in multi-objective optimization (Marler and Arora, 2004). Among the shortcomings are the fact that weights may show a preference of one objective over the other and, most importantly, the impossibility of reaching some parts of the Pareto front when dealing with convex fronts.

In contrast, work by Lopez-Herrejon et al. propose an approach for computing the exact Pareto front for pairwise coverage of two objective functions, maximization of coverage and minimization of test suite size (Lopez-Herrejon et al., 2013). Subsequent work also by Lopez-Herrejon et al. studied four classical multi-objective algorithms and the impact of seeding for computing pairwise covering arrays (Lopez-Herrejon et al., 2014). These approaches have in common that they do not consider prioritization and are not evaluated with actual real-world case studies. Closets to our work, Wang et al. compare SPL testing techniques that include different weight-assignment approaches for scalarization into single objective problems, multi-objective evolutionary algorithms, swarm particle algorithms, and hybrid algorithms (Wang et al., 2015a). In contrast with our work, they use a different set of objective functions, and their industrial case study considers only a handful of products for which they aim to minimize the cost of selecting already existing test cases. Multi-objective techniques have also been used at other stages of the SPL development life cycle, for instance for product configuration. For a summary please refer to a recent mapping study on the application of Search-Based Software Engineering techniques to SPLs (Lopez-Herrejon et al., 2015).

Drupal is, to the best of our knowledge, the first documented case study that provides detailed information regarding faults and their relation to features and their interactions based on developers' logs. Recent work by Abal et al. collected a database with similar information for several versions of the Linux kernel (Abal et al., 2014). We plan to look at this case study to further corroborate or refute our findings.

General software testing. In the general context of software testing there is extensive literature that relates to our work in the sense of using multi-objective algorithms or prioritization schemes but not particularly applied to HCSs. For example, Harman et al. provide an overview of the area of Search-Based Software Engineering which shows the prevalence of testing as the development activity where search-based techniques are commonly used (Harman et al., 2012). Similarly, Yoo and Harman present a general overview of regression testing that includes prioritization (Yoo and Harman, 2012b). Among their findings is the work by Li et al. that applied and compared several metaheuristics for test case prioritization (Li et al., 2007). They show that even though genetic algorithms work well, greedy approaches are also effective.

Regarding multi-objective algorithms, Sayyad and Ammar performed a survey on pareto-optimal SBSE which identified the growing interest and use of classical multi-objective algorithms (Sayyad and Ammar, 2013). The articles that they identified in the testing area do not, however, deal with HCSs. Yoo and Harman propose treating test case selection as a multi-objective problem with a Pareto efficient approach (Yoo and Harman, 2007). Islam et al. propose an approach that uses traceability links among source code and system requirements, recovered via the Latent Semantic Indexing (LSI) technique, as one of the multi-objective functions to optimize (Islam et al., 2012). More recently, Marchetto et al. (2015) extend on this work to provide a more thorough analysis and evaluation of using LSI in combination with more metrics for test case prioritization.

10. Conclusions

This article presented a real–world case study on multi–objective test case prioritization in Drupal, a highly configurable web framework. In particular, we adapted the NSGA-II evolutionary algorithm to solve the multi-objective prioritization problem in HCSs. Our algorithm uses seven novel objective functions based on functional and non-functional properties of the HCS under test. We performed several experiments comparing the effectiveness of 63 different combinations of up to three of these objectives in accelerating the detection of faults in Drupal. Results revealed that prioritization driven by non-functional objectives, such as the number of faults found in a previous version of the system, accelerate the detection of bugs more effectively than functional prioritization objectives. Furthermore, it was observed that the prioritization objective based on pairwise coverage, when combined with other objectives, is usually effective in detecting bugs quickly. Finally, results showed that multi-objective prioritization performs better than mono-objective prioritization in general. To the best of our knowledge, this is the first comparison of test case prioritization objectives for HCSs using industry–strength data.

Several challenges remain for future work. First, the development of similar case studies in other HCSs would be a nice complement to study the generalizability of our conclusions. Also, the result of combining more than three objectives is a topic that remains unexplored and for which other algorithms (so-called many–objectives algorithms) are probably more suited. Finally, we may remark that part of the results of this work have been integrated into SmarTest, a Drupal test prioritization module developed by some of the authors and recently presented at the International Drupal Conference (Sánchez et al., 2015a) with very positive feedback from the community.

Material

For the sake of replicability, the source code of our algorithm, the Drupal attributed feature model, experimental results and statistical analysis scripts in R are publicly available at http://exemplar.us.es/demo/SanchezJSS2016 (100Mb).

Acknowledgments

This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT projects TAPAS (TIN2012-32273) and BELI (TIN2015-70560-R), and the excellence network SEBASE (TIN2015-71841-REDT), the Andalusian Government project COPAS (P12-TIC-1867) and the Austrian Science Fund (FWF) projects P25513-N15.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.jss.2016.09.045

References

Abal, I., Brabrand, C., Wasowski, A., 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In: Crnkovic, I., Chechik, M., Grünbacher, P. (Eds.), ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15– 19, 2014. ACM, pp. 421–432.

Al-Hajjaji, M., Thum, T., Meinicke, J., Lochau, M., Saake, G., 2014. Similarity-based prioritization in software product-line testing. In: Software Product Line Conference, pp. 197–206.

Arcuri, A., Briand, L., 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Rel. 24 (3), 219–250.

Bagheri, E., Gasevic, D., 2011. Assessing the maintainability of software product line feature models using structural metrics. Softw. Qual. Control.

Batory, D., 2005. Feature models, grammars, and propositional formulas. In: Software Product Lines Conference (SPLC). Springer–Verlag, pp. 7–20.

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

Beume, N., Fonseca, C.M., Lopez-Ibanez, M., Paquete, L., Vahrenhold, J., 2009. On the complexity of computing the hypervolume indicator. IEEE Trans. Evol. Comput. 13 (5), 1075–1082.

Buytaert, D., 2015. Drupal Framework. accessed in October 2015, http://www.drupal.org.

Cohen, M.B., Dwyer, M.B., Shi, J., 2008. Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. Trans. Softw. Eng.

da Mota Silveira Neto, P.A., do Carmo Machado, I., McGregor, J.D., de Almeida, E.S., de Lemos Meira, S.R., 2011. A systematic mapping study of software product lines testing. Inf. Softw. Technol. 53 (5), 407–423.

Deb, K., Deb, K., 2014. Multi-objective optimization. In: Burke, E.K., Kendall, G. (Eds.), Search Methodologies. Springer US, pp. 403–449.

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. Evol. Comput. IEEE Trans. 6 (2), 182–197.

Debian, 2013. Debian 7.0 Wheezy Released. Accessed November 2013.

Derrac, J., Garca, S., Molina, D., Herrera, F., 2011. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. Swarm Evol. Comput. 1 (1), 3–18.

Devroey, X., Perrouin, G., Cordy, M., Schobbens, P.-Y., Legay, A., Heymans, P., 2014a. Towards statistical prioritization for software product lines testing. In: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive (VAMOS).

Devroey, X., Perrouin, G., Schobbens, P., 2014b. Abstract test case generation for behavioural testing of software product lines. In: Software Product Line Conference. ACM, pp. 86–93.

do Carmo Machado, I., McGregor, J.D., Cavalcanti, Y.C., de Almeida, E.S., 2014. On strategies for testing software product lines: a systematic literature review. Inf. Softw. Technol. 56 (10), 1183–1199.

Durillo, J.J., Nebro, A.J., 2011. jMetal: a java framework for multi-objective optimization. Adv. Eng. Softw. 42, 760–771.

Elbaum, S., Rothermel, G., Kanduri, S., Malishevsky, A.G., 2004. Selecting a cost-effective test case prioritization technique. Softw. Qual. J.

Engström, E., Runeson, P., 2011. Software product line testing - a systematic mapping study. Inf. Softw. Technol. 53 (1), 2–13.

Ensan, A., Bagheri, E., Asadi, M., Gasevic, D., Biletskiy, Y., 2011. Goal-oriented test case selection and prioritization for product line feature models. In: Conference Information Technology: New Generations.

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

Epitropakis, M.G., Yoo, S., Harman, M., Burke, E.K., 2015. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. International Symposium on Software Testing and Analysis.

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

Ferrer, J., Chicano, F., Alba, E., 2012. Evolutionary algorithms for the multi-objective test data generation problem. Softw. Pract. Exp.

García-Galán, J., Rana, O., Trinidad, P., Ruiz-Cortés, A., 2013. Migrating to the cloud: a software product line based analysis. In: 3rd International Conference on Cloud Computing and Services Science (CLOSER'13).

Garvin, B., Cohen, M., Dwyer, M., 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empir. Softw. Eng. 16 (1), 61–102.

Garvin, B.J., Cohen, M.B., Dwyer, M.B., 2009. An improved meta-heuristic search for constrained interaction testing. In: Search Based Software Engineering, 2009 1st International Symposium on. IEEE, pp. 13–22.

Graves, T.L., Karr, A.F., Marron, J.S., Siy, H., 1998. Predicting Fault Incidence Using Software Change History. Technical Report. National Institute of Statistical Sciences. 653–661.

Harman, M., Mansouri, S.A., Zhang, Y., 2012. Search-based software engineering: trends, techniques and applications. ACM Comput. Surv. 45 (1), 11.

Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Traon, Y.L., 2012. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-wise Test Suites for Large Software Product Lines. Technical Report.

Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Traon, Y.L., 2014. Bypassing the combinatorial explosion: using similarity to generate and prioritize t-wise test configurations for software product lines. IEEE Trans. Softw. Eng. 40, 1.

Henard, C., Papadakis, M., Perrouin, G., Klein, J., Traon, Y.L., 2013. Multi-objective test generation for software product lines. In: International Software Product Line Conference (SPLC).

Holm, S., 1979. A simple sequentially rejective multiple test procedure. Scand. J. Statist. 6 (2), 65–70.

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

Islam, M., Marchetto, A., Susi, A., Scanniello, G., 2012. A multi-objective technique to prioritize test cases based on latent semantic indexing. In: Mens, T., Cleve, A., Ferenc, R. (Eds.), 16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27–30, 2012. IEEE Computer Society, pp. 21–30.

Johansen, M.F., Haugen, O., Fleurey, F., 2011. Properties of realistic feature models make combinatorial testing of product lines feasible. MODELS.

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

Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S., 1990. Featureoriented domain analysis (FODA) feasibility study. SEI.

Lew, K.S., Dillon, T.S., Forward, K.E., 1988. Software complexity and its impact on software reliability. Trans. Softw. Eng. 14, 1645–1655.

Li, Z., Harman, M., Hierons, R.M., 2007. Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. 33 (4), 225–237.

Lopez-Herrejon, R., Chicano, F., Ferrer, J., Egyed, A., Alba, E., 2013. Multi-objective optimal test suite computation for software product line pairwise testing. In: Proceedings of the 29th IEEE International Conference on Software Maintenance.

Lopez-Herrejon, R.E., Ferrer, J., Chicano, F., Egyed, A., Alba, E., 2014. Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of software product lines. In: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, July 6–11, 2014. IEEE, pp. 387–396.

Lopez-Herrejon, R.E., Ferrer, J., Chicano, F., Haslinger, E.N., Egyed, A., Alba, E., 2014. A parallel evolutionary algorithm for prioritized pairwise testing of software product lines. In: Arnold, D.V. (Ed.), GECCO. ACM, pp. 1255–1262.

Lopez-Herrejon, R.E., Fischer, S., Ramler, R., Egyed, A., 2015. A first systematic mapping study on combinatorial interaction testing for software product lines. In: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015. IEEE Computer Society, pp. 1–10.

Lopez-Herrejon, R.E., Linsbauer, L., Egyed, A., 2015. A systematic mapping study of search-based software engineering for software product lines. J. Inf. Softw. Technol.

Mann, H.B., Whitney, D.R., 1947. On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Statist. 18 (1), 50–60.

Marchetto, A., Islam, M., Asghar, W., Susi, A., Scanniello, G., 2015. A multi-objective technique to prioritize test cases. IEEE Trans. Softw. Eng. PP (99). 1.

Marijan, D., Gotlieb, A., Sen, S., Hervieu, A., 2013. Practical pairwise testing for software product lines. In: Proceedings of the 17th International Software Product Line Conference. ACM, New York, NY, USA, pp. 227–235.

Marler, R., Arora, J., 2004. Survey of multi-objective optimization methods for engineering. Structural and Multidisciplinary Optimization 26 (6), 369–395.

Matsumoto, S., kamei, Y., Monden, A., Matsumoto, K., Nakamura, M., 2010. An analyses of developer metrics for fault prediction. In: International Conference on Predictive Models in Software Engineering.

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

Perrouin, G., Oster, S., Sen, S., Klein, J., Budry, B., le Traon, Y., 2011. Pairwise testing for software product lines: comparison of two approaches. Softw. Qual. J.

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

Qu, X., Cohen, M.B., Rothermel, G., 2008. Configuration-aware regression testing: an empirical study of sampling and prioritization. International Symposium in Software Testing and Analysis.

Rothermel, G., Untch, R., Chu, C., Harrold, M., 2001. Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. 27, 929–948.

Sánchez, A.B., Segura, S., Cortés, A.R., 2015a. SmarTest: accelerating the detection of faults in drupal. DrupalConEurope 2015.

Sánchez, A.B., Segura, S., Parejo, J.A., Ruiz-Cortés, A., 2015b. Variability testing in the wild: the drupal case study. Softw. Syst. Model. J. 1–22.

Sánchez, A.B., Segura, S., Ruiz-Cortés, A., 2014. A comparison of test case prioritization criteria for software product lines. In: IEEE International Conference on Software Testing, Verification, and Validation, pp. 41–50.

Sayyad, A.S., Ammar, H., 2013. Pareto-optimal search-based software engineering (POSBSE): a literature survey. In: 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE 2013, San Francisco, CA, USA, May 25–26, 2013. IEEE Computer Society, pp. 21–27.

Segura, S., 2008. Automated analysis of feature models using atomic sets. In: First Workshop on Analyses of Software Product Lines (ASPL). Limerick, Ireland, pp. 201–207.

Segura, S., Sánchez, A.B., Ruiz-Cortés, A., 2014. Automated variability analysis and testing of an e-commerce site: an experience report. In: International Conference on Automated Software Engineering. ACM, pp. 139–150.

Simons, C., Paraiso, E.C., 2010. Regression test cases prioritization using failure pursuit sampling. In: International Conference on Intelligent Systems Design and Applications, pp. 923–928.

SmarTest, http://www.isa.us.es/smartest/index.html, Accessed October 2015.

Srikanth, H., Cohen, M.B., Qu, X., 2009. Reducing field failures in system configurable software: cost-based prioritization. In: 20th International Symposium on Software Reliability Engineering, pp. 61–70.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T., 2014. Featureide: an extensible framework for feature-oriented software development. Sci. Comput. Program. 79, 70–85.

Tomlinson, T., VanDyk, J.K., 2010. Pro Drupal 7 Development: Third Edition.

Vargha, A., Delaney, H.D., 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educat. Behav. Stat. 25 (2), 101–132.

von Rhein, A., Grebhahn, A., Apel, S., Siegmund, N., Beyer, D., Berger, T., 2015. Presence-condition simplification in highly configurable systems. In: International Conference on Software Engineering.

Wang, S., Ali, S., Gotlieb, A., 2013. Minimizing test suites in software product lines using weight-based genetic algorithms. In: Genetic and Evolutionary Computation Conference (GECCO).

Wang, S., Ali, S., Gotlieb, A., 2014a. Random-weighted search-based multi-objective optimization revisited. In: Goues, C.L., Yoo, S. (Eds.), Search-Based Software Engineering - 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26–29, 2014. Proceedings. Springer, pp. 199–214.

Wang, S., Ali, S., Gotlieb, A., 2015a. Cost-effective test suite minimization in product lines using search techniques. J. Syst. Softw. 103, 370–391.

Wang, S., Ali, S., Yue, T., Bakkeli, Ø., Liaaen, M., 2016. Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search. In: Dillon, L.K., Visser, W., Williams, L. (Eds.), Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016 - Companion Volume. ACM, pp. 182–191.

Wang, S., Ali, S., Yue, T., Liaaen, M., 2015b. Upmoa: an improved search algorithm to support user-preference multi-objective optimization. ISSRE.

Wang, S., Buchmann, D., Ali, S., Gotlieb, A., Pradhan, D., Liaaen, M., 2014b. Multi--objective test prioritization in software product line testing: an industrial case study. In: Software Product Line Conference, pp. 32–41.

Xu, Z., Cohen, M.B., Motycka, W., Rothermel, G., 2013. Continuous test suite augmentation in software product lines. In: Software Product Line Conference.

Yilmaz, C., Fouché, S., Cohen, M.B., Porter, A.A., Demiröz, G., Koc, U., 2014. Moving forward with combinatorial interaction testing. IEEE Comput. 47 (2), 37–45.

Yoo, S., Harman, M., 2007. Pareto efficient multi-objective test case selection. In: Rosenblum, D.S., Elbaum, S.G. (Eds.), Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9–12, 2007. ACM, pp. 140–150.

Yoo, S., Harman, M., 2012a. Regression testing minimisation, selection and prioritisation: a survey. Softw. Test. Verif. Rel. 22, 67–120.

Yoo, S., Harman, M., 2012b. Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Rel. 22 (2), 67–120.

Zhou, A., Qu, B.-Y., Li, H., Zhao, S.-Z., Suganthan, P.N., Zhang, Q., 2011. Multiobjective evolutionary algorithms: a survey of the state of the art. Swarm Evol. Comput. 1 (1), 32–49.

Zitzler, E., 2012. Evolutionary multiobjective optimization. In: Handbook of Natural Computing, pp. 871–904.

Jose A. Parejo received his Ph.D. (with honors) from University of Sevilla in 2103, where he currenly works as senior lecturer of software engineering. He has worked in the industry as developer, architect and project manager from 2001 to 2007. His research interests include metaheuristic optimization and software engineering, focusing mainly on search-based software engineering. He serves regularly as reviewer for international journals and conferences.

Ana B. Sánchez received her Ph.D. (with honors) from University of Seville in May 2016. She has worked as a research assistant in the Applied Software Engineering research group (ISA, www.isa.us.es) at the University of Seville in the area of automated testing of highly configurable systems. Contact her at anabsanchez@us.es.

Sergio Segura received his Ph.D. in software engineering (with honours) from Seville University, where he currently works as a senior lecturer. His research interests include software testing, software variability and search-based software engineering. He has co-authored some highly cited papers as well as tools used by universities and companies in various countries. He also serves regularly as a reviewer for international journals and conferences.

Antonio Ruiz-Cortés is an accredited full professor and Head of the Applied Software Engineering research group at the University of Seville, in which he received his Ph.D. (with honours). and M.Sc. in Computer Science. His research interests are in the areas of service oriented computing, software variability, software testing, and business process management. Further information about his publications, research projects and activities can be found at www.isa.us.es

Roberto Erick Lopez-Herrejon is an associate professor at the Department of Software Engineering and Information Technology of the École de Technologie Supérieure of the University of Quebec in Montreal, Canada. Prior he was a senior postdoctoral researcher at the Johannes Kepler University in Linz, Austria. He was an Austrian Science Fund (FWF) Lise Meitner Fellow (2012–2014) at the same institution. From 2008 to 2014 he was an External Lecturer at the Software Engineering Masters Programme of the University of Oxford, England. From 2010 to 2012 he held an FP7 Intra-European Marie Curie Fellowship sponsored by the European Commission. He obtained his Ph.D. from the University of Texas at Austin in 2006, funded in part by a Fulbright Fellowship. From 2005 to 2008, he was a Career Development Fellow at the Software Engineering Centre of the University of Oxford. His main expertise is in software customization, software product lines, and search based software engineering. roberto.lopez@etsmtl.ca Department of Software Engineering and IT. École de Technologie Supérieure, (ÉTS), Notre-Dame Street Ouest. 1100, H3C 1K3. Montreal, Canada.

Alexander Egyed heads the Institute for Software Engineering and Automation at the Johannes Kepler University, Austria. He is also an adjunct assistant professor at the University of Southern California, USA. Before joining the JKU, he worked as a research scientist for Teknowledge Corporation, USA (2000–2007) and then as a Research Fellow at the University College London, UK (2007–2008). He received a doctorate degree in 2000 and a master of science degree in 1996, both in computer science, from the University of Southern California, USA under the mentorship of Dr. Barry Boehm. His research interests include software design modeling, requirements engineering, consistency checking and resolution, traceability, and change impact analysis. He is a member of ACM, ACM SigSoft, IEEE, and IEEE Computer Society.