Test Case Prioritization in Highly-Configurable Systems

Ana Belén Sánchez Jerez

PhD Dissertation

Supervised by

Dr. Sergio Segura Rueda

and

Dr. Antonio Ruiz Cortés

Universidad de Sevilla

March 2016

Support: PhD dissertation granted by the Andalusian Government project THEOS (TIC-5906) and COPAS (P12-TIC-1867)

Dedicado a todos los que me han apoyado

Y en especial a mis padres y a mi prometido

Abstract

Highly-configurable software systems (HCSs) provide a common core functionality and a set of optional features, where a feature represents an increment in system functionality. Development tools as Eclipse (with more than 3,000 plug-ins) or operating systems as Debian Wheezy (with more than 37,000 packages) have been reported as examples of HCSs. The large number of features that can be combined leading to thousands or even millions of individual software systems (a.k.a. configurations) makes the testing of HCSs a challenge. That is, testing every single configuration is too expensive in general. To overcome this problem, researchers have proposed numerous techniques mainly focused on reducing the space of testing to a manageable but representative subset of configurations to be tested. Nevertheless, even after reduce the test space, the number of configurations under test may still be large and expensive to run.

In this dissertation, we address the testing of HCSs using test case prioritization, which schedules test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goal, typically detect faults as quickly as possible. Test case prioritization approaches help to improve the effectiveness of testing allowing faster feedback to software testers and ensuring that test cases with the highest fault detection ability will have been executed if testing is stopped by any circumstance. Test case prioritization in HCSs can be driven by different functional and non–functional objectives, being an objective the order criterion used to accelerate the detection of faults. Functional prioritization objectives are based on the functional features of the system and their interactions. Non–functional prioritization objectives consider extra–functional information such as user preferences or cost. In this thesis we present thirteen objectives, techniques and tools for test case prioritization in HCSs. In particular, we define six objectives based on functional properties and seven objectives based on non–functional properties of the HCS under test. These objectives are evaluated using real data extracted from HCS Git repositories and bug tracking systems, which reinforces the validity of our conclusions. Regarding test case prioritization techniques, we present single–objective and multi-objective approaches based on evolutionary algorithms. Additionally, we report a comparison of 63 different combinations of up to three objectives to determine which combinations perform better in accelerating the detection of faults in an HCS. Furthermore, we propose an industry-strength HCS case study with more than 2 billions of configurations to be used as a realistic subject for further and reproducible validation of variability testing techniques. These contributions have been evaluated using extensive and rigorous experiments that reveal the efficacy and efficiency of our approach. Part of our contributions have been integrated into a tool called SmarTest for testing Drupal, a well-known web content management framework. *SmarTest* is a testing module that supports the analysis of the Drupal system to provide useful information to guide the testing. Also, it allows applying different prioritization testing techniques to reveals bugs faster in Drupal.

Resumen

Los sistemas software altamente configurables proporcionan una funcionalidad básica común y un conjunto de características opcionales para adaptar todas las variantes de un sistema de acuerdo a un conjunto determinado de requisitos. Sistemas operativos como Linux (con más de 37,000 paquetes) o herramientas de desarrollo como Eclipse (con más de 3,000 plugins) han sido presentados como ejemplos de sistemas altamente configurables. Las pruebas en sistemas software altamente configurables suponen un gran reto debido al elevado número de configuraciones que deben probarse. Por ejemplo, linux con más de 37,000 paquetes puede dar lugar a miles de millones de configuraciones diferentes. Esto hace que probar cada configuración individual de un sistema altamente configurable sea demasiado costoso. Para paliar este problema, se han propuesto numerosos trabajos de investigación para reducir el espacio de pruebas a un subconjunto razonable y representativo de configuraciones a probar. Sin embargo, incluso reduciendo el espacio de pruebas, el número de configurationes resultante puede seguir siendo demasiado grande y costoso para llevar a cabo las pruebas.

En esta tesis, se aborda el problema anteriormente mencionado utilizando priorización de casos de prueba para sistemas altamente configurables. Las técnicas de priorización de casos de prueba ordenan las pruebas de manera que se ejecuten primero aquellas que permitan maximizar un determinado objetivo de rendimiento, normalmente, detectar errores lo antes posible. La priorización de casos de prueba ayuda a mejorar la efectividad del proceso de pruebas permitiendo un feedback más rápido a los ingenieros del software y asegurando que las pruebas con mayor capacidad de detectar errores habrán sido ejecutadas si el proceso se parase por alguna circunstancia. La priorización de pruebas en sistemas software altamente configurables puede guiarse por diferentes objetivos funcionales y no funcionales, entendiéndose un objetivo como el criterio de ordenación elegido para acelerar la detección de errores. Los objetivos de priorización funcionales son aquellos basados en características funcionales del sistema y sus interacciones. Los objetivos no funcionales consideran información extra-funcional tales como las preferencias de usuario. En esta tesis se presentan un conjunto de objetivos, técnicas y herramientas para la priorización de casos de prueba en sistemas altamente configurables. En concreto, se definen 6 objetivos basados en propiedades funcionales y 7 objetivos basados en propiedades no funcionales del sistema para guiar el proceso de pruebas. Estos objetivos se evalúan utilizando datos reales extraídos de los repositiorios Git y sistemas de gestión de errores de sistemas altamente configurables, reforzándo así la validez de nuestras conclusiones. Con respecto a las técnicas de priorización de pruebas, se presentan propuestas basadas en un sólo objetivo y basadas en múltiples objetivos utilizando algoritmos evolutivos. Adicionalmente, se presenta una comparación de 63 combinaciones diferentes de 1, 2 y 3 objetivos con la intención de encontrar las mejores combinaciones para accelerar la detección de errores en un sistema altamente configurable. Además, proponemos un caso de estudio basado en un sistema real altamente configurable con más de 2,000 millones de configuraciones para evaluar y validar las técnicas de pruebas. Estas contribuciones han sido evaluadas realizando rigurosos experimentos que han revalado la eficacia y efficiancia de nuestra propuesta. Parte de nuestras contribuciones se han integrado en una herramienta denominada SmarTest para llevar acabo las pruebas en Drupal. Drupal es un sistema de gestión de contenidos web y framework ampliamente conocido. SmarTest es un módulo de pruebas para Drupal que permite analizar el sistema y proporcionar al ingeniero de calidad información relevante de Drupal que le permita guiar mejor el proceso de pruebas. Además, SmarTest también permite aplicar diferentes técnicas de priorización para detectar antes los errores en Drupal.

Contents

List of Figures

List of Tables

Part I

Preface

Introduction

"Which road do I take?" Alicia asked.

"Where do you want to go?" responded the Cheshire Cat.

Alice in Wonderland (1865), Lewis Caroll,

In this dissertation, we report on our work to develop a set of techniques, algorithms and tools to support test case prioritization in HCSs. In this chapter, we first introduce the topics that constitute the context of our research work in Section §1.1. Section §1.2 motivates this dissertation. The goals of this thesis are described in Section §1.3. We summarize our main contributions and publications in Section §1.4. Finally, in Section §1.5, we describe the structure of the dissertation.

1.1 Research context

In the next sections, we briefly present the main concepts that we will use throughout this dissertation. In Section §1.1.1, we present HCSs. Section §1.1.2 describes the testing of HCSs. Test case prioritization is introduced in Section §1.1.3.

1.1.1 Highly-configurable systems

Software development is progressively transitioning from the development of individual programs to the development of families of related programs. This leads to the design and implementation of software systems that share a common core set of capabilities, but have key differences, such as the hardware platform they require, the interfaces they expose, or the optional capabilities they provide to users. Frequently, significant reuse can be achieved by implementing a set of these systems as one integrated highly-configurable software system [22]. *Highly-configurable software systems*

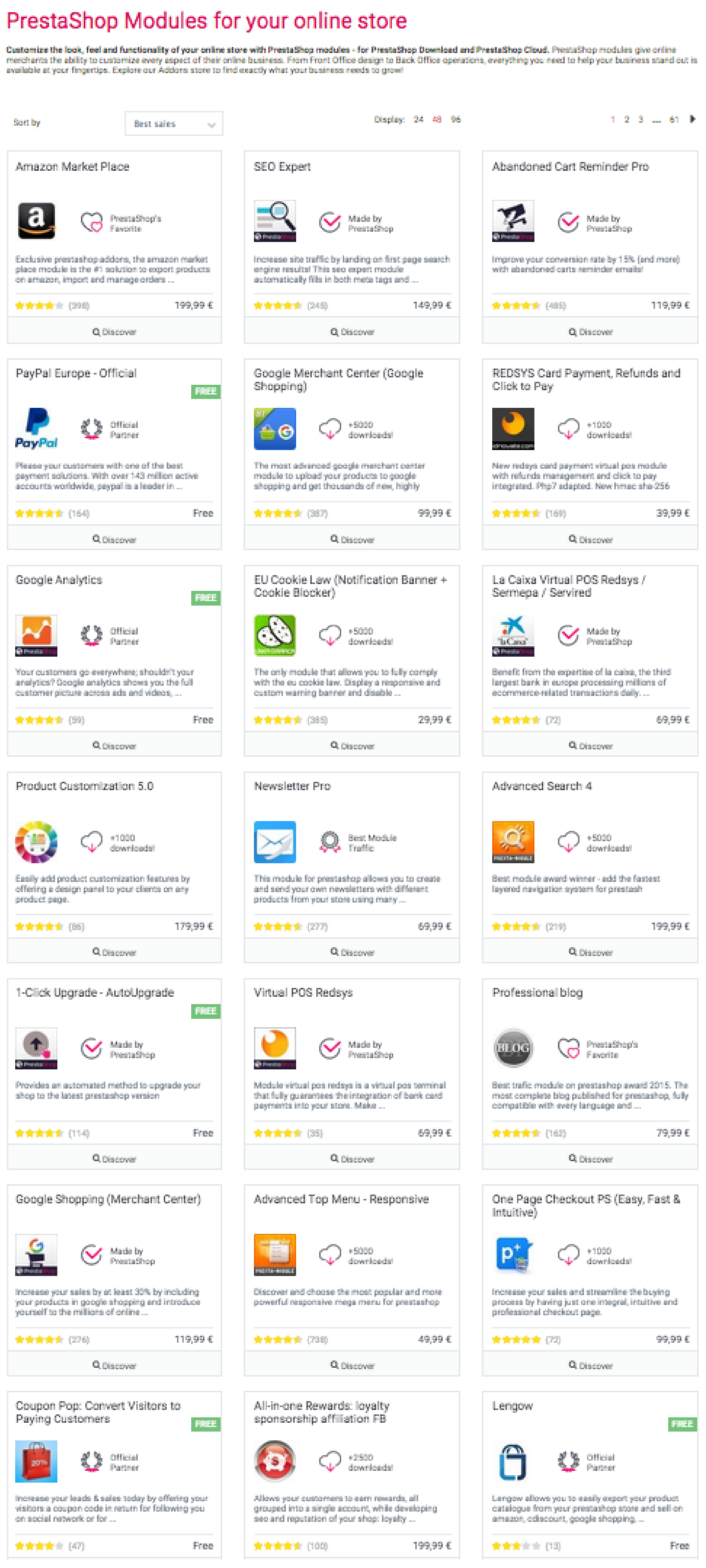
(HCSs) provide a common core functionality and a set of optional features to tailor variants of the system (a.k.a. configurations) according to a given set of requirements [144]. An example of HCS is seen in software product lines. Software product line engineering is about producing a set of related products that share commonalities and are differentiated by variabilities. Some cases of success of systems adapted to HCSs are those coming from Boeing, Philips and Nokia companies that were reported in [40]. Another examples reported as HCSs are operating systems as Linux [84, 127], development tools as Eclipse [68] or even cloud applications as the Amazon elastic compute service [50]. Figure §1.1 illustrates the module manager for the open source e-commerce platform Prestashop, a tool for the development of online shopping systems. This is another example of HCS, with more than 3,500 modules and visual templates powering more than 150,000 online stores worldwide.

HCSs are usually represented in terms of features. A *feature* depicts a choice to include a certain functionality in a system configuration [144]. 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 HCSs in terms of features and constraints among them (see Chapter §2 for a more detailed information) [14]. A configuration of an HCS is a valid composition of features satisfying all the constraints of the model. Figure §1.2 depicts a simplified example feature model inspired by the e-commerce industry. The model illustrates how features and relationships among them are used to specify the commonalities and variabilities of the on-line shopping system.

The development of HCSs provides clear benefits for IT industries facilitating the design and implementation of multiple software systems that share a common core set of capabilities reducing costs and time-to-market to companies. Nevertheless, these systems also present significant challenges such as their validation. The problem of validating a single configuration has been replaced with the much harder problem of validating the whole set of configurations that can be produced by all of the different possible combinations of features in HCSs.

1.1.2 Testing of highly-configurable software systems

Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely, that it does not do anything unintended [53]. In an ideal world, we would want to test every possible con-



PrestaShop Modules for your online store

Customize the look, feel and functionality of your online store with PrestaShop modules - for PrestaShop Download and PrestaShop Cloud. PrestaShop modules give online merchants the ability to customize every aspect of their online business. From Front Office design to Back Office operations, everything you need to help your business stand out is available at your fingertips. Explore our Addons store to find exactly what your business needs to grow!

Sort by: Best sales — Display: 24 48 96 — 1 2 3 ... 61

Amazon Market Place — PrestaShop's Favorite
Exclusive prestashop addons, the amazon market place module is the #1 solution to export products on amazon, import and manage orders ...
(398) 199,99 € — Discover

SEO Expert — Made by PrestaShop
Increase site traffic by landing on first page search engine results! This seo expert module automatically fills in both meta tags and ...
(245) 149,99 € — Discover

Abandoned Cart Reminder Pro — Made by PrestaShop
Improve your conversion rate by 15% (and more) with abandoned carts reminder emails!
(485) 119,99 € — Discover

PayPal Europe - Official — FREE — Official Partner
Please your customers with one of the best payment solutions. With over 143 million active accounts worldwide, paypal is a leader in ...
(164) Free — Discover

Google Merchant Center (Google Shopping) — +5000 downloads!
The most advanced google merchant center module to upload your products to google shopping and get thousands of new, highly
(387) 99,99 € — Discover

REDSYS Card Payment, Refunds and Click to Pay — +1000 downloads!
New redsys card payment virtual pos module with refunds management and click to pay integrated. Php7 adapted. New hmac sha-256
(169) 39,99 € — Discover

Google Analytics — FREE — Official Partner
Your customers go everywhere; shouldn't your analytics? Google analytics shows you the full customer picture across ads and videos, ...
(59) Free — Discover

EU Cookie Law (Notification Banner + Cookie Blocker) — +5000 downloads!
The only module that allows you to fully comply with the eu cookie law. Display a responsive and custom warning banner and disable ...
(385) 29,99 € — Discover

La Caixa Virtual POS Redsys / Sermepa / Servired — Made by PrestaShop
Benefit from the expertise of la caixa, the third largest bank in europe processing millions of ecommerce-related transactions daily. ...
(72) 69,99 € — Discover

Product Customization 5.0 — +1000 downloads!
Easily add product customization features by offering a design panel to your clients on any product page.
(86) 179,99 € — Discover

Newsletter Pro — Best Module Traffic
This module for prestashop allows you to create and send your own newsletters with different products from your store using many ...
(277) 69,99 € — Discover

Advanced Search 4 — +5000 downloads!
Best module award winner - add the fastest layered navigation system for prestash
(219) 199,99 € — Discover

1-Click Upgrade - AutoUpgrade — FREE — Made by PrestaShop
Provides an automated method to upgrade your shop to the latest prestashop version
(114) Free — Discover

Virtual POS Redsys — Made by PrestaShop
Module virtual pos redsys is a virtual pos terminal that fully guarantees the integration of bank card payments into your store. Make ...
(35) 69,99 € — Discover

Professional blog — PrestaShop's Favorite
Best trafic module on prestashop award 2015. The most complete blog published for prestashop, fully compatible with every language and ...
(162) 79,99 € — Discover

Google Shopping (Merchant Center) — Made by PrestaShop
Increase your sales by at least 30% by including your products in google shopping and introduce yourself to the millions of online ...
(276) 119,99 € — Discover

Advanced Top Menu - Responsive — +5000 downloads!
Discover and choose the most popular and more powerful responsive mega menu for prestashop
(738) 49,99 € — Discover

One Page Checkout PS (Easy, Fast & Intuitive) — +1000 downloads!
Increase your sales and streamline the buying process by having just one integral, intuitive and professional checkout page.
(72) 99,99 € — Discover

Coupon Pop: Convert Visitors to Paying Customers — FREE — Official Partner
Increase your leads & sales today by offering your visitors a coupon code in return for following you on social network or for ...
(47) Free

All-in-one Rewards: loyalty sponsorship affiliation FB — +2500 downloads!
Allows your customers to earn rewards, all grouped into a single account, while developing seo and reputation of your shop: loyalty ...
(100) 199,99 €

Lengow — FREE — Official Partner
Lengow allows you to easily export your product catalogue from your prestashop store and sell on amazon, cdiscount, google shopping, ...
(13) Free

Figure 1.1: Example illustrating some available modules for Prestashop e-commerce manager

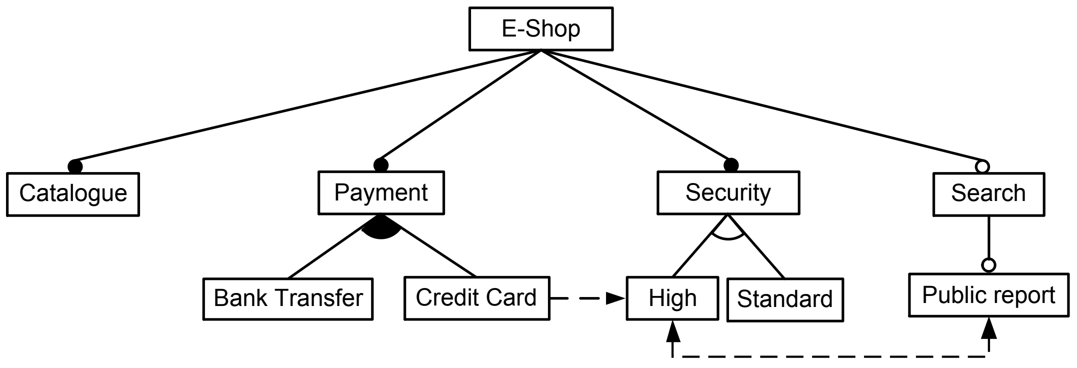


Figure 1.2: E-shop feature model

figuration of a program. In most cases, however, this simply is not possible. Even a seemingly simple application can have hundreds or thousands of possible input and output combinations. Creating test cases for all of these possibilities is impractical. Complete testing of a complex application such as an HCS would take too long and require too many resources to be economically feasible [53].

In this context, researchers have proposed different techniques to reduce the cost of testing in the presence of high configurability, including test case selection and test case prioritization techniques. *Test case selection* selects an appropriate subset of an existing test suite according to some coverage criteria [30, 58, 74, 88]. *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 [5, 82, 146]. Both test selection and test prioritization strategies are complementary and are often combined. Most of the approaches for HCSs testing use a model-based approach, that is, they take an input model (usually a feature model) representing the HCS and return a (ordered) subset of feature configurations to be tested, i.e. a test suite.

1.1.3 Test case prioritization

Testing each and every test case of a complex software application may require hours, days or even weeks [38, 59, 111]. This can be due to different reasons. For instance, in modern ecosystems the number of test cases can reach the figure of millions of tests, which requires a large amount of execution time. As an example, Eclipse has more than 40,000 test cases [66]. Moreover, the execution of tests is not always automated and sometimes the tests are computationally very expensive [94]. These aspects may be especially critical during regression testing when tests must be repeatedly executed after any relevant change is made to the software. In this context, time and budget constraints may hinder the complete execution of a test suite [102, 111, 150]. Also, even reducing the number of test cases, the resulting test suite may still be too large and expensive to run. As and example, the authors in [147] reported that after applying test selection to Cisco case study, the number of test cases was still large, more that 2000 test cases requiring 3-4 days to execute per configuration.

Test case prioritization techniques aim to identify the optimal ordering of test cases to maximize certain performance goals, such as accelerating the detection of faults. This idea can be applied to any type of tests, i.e. unit, integration and system testing. For example, we could accelerate the detection of faults testing first the most complex components, assuming that these are the most error-prone components. Also, we could test earlier those tests that revealed more faults in previous execution of the test suite or test first those ones covering more code faster.

Test case prioritization techniques help to improve the cost-effectiveness of testing in general and regression testing in particular. Also, they provide several benefits, such as earlier detection of faults and faster feedback to testers and developers enabling the early fix of faults, earlier evidence that quality goals were not met and value of ensuring that test cases that offer the greatest fault detection ability will have been executed if testing is halted.

1.2 Motivation

Although testing is a hard and time-consuming task in the software development process, in the case of HCSs it becomes more challenging due to the potentially huge number of configurations under test. This makes exhaustive testing of an HCS infeasible, that is, testing every single configuration is too expensive in general. For instance, Debian Wheezy, a well-known Linux distribution, provides more than 37,000 packages that can be combined with restrictions leading to billions of potential configurations, more than stars in the Milky Way galaxy.

In order to alleviate this problem, numerous contributions have been proposed to reduce the number of configurations to be tested while still having a good coverage. Most coverage approaches proposed are based on combinatorial interaction testing (CIT), where test cases are selected in a way that ensure that all combinations (t-way) of t features are tested. However, not much attention has been paid to the order in which the configurations are tested in HCSs.

Current literature for prioritization in HCS testing are scarce and most of them are based on weighted CIT, i.e., they generate combinatorial configurations using weights to prioritize them [59, 68, 82]. These approaches combine selection and prioritization during the test case generation. To the best of our knowledge, test case prioritization proposals for HCSs follow a single–objective perspective [5, 29, 42, 59, 82], that is, they either aim to maximize or minimize a single objective (e.g. feature coverage or suite size), except for the work proposed by Wang et al. [146] that employs the concept of multi-objective prioritization to combine different objectives into a single function by assigning weights to them 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 [81]. Thus, the potential benefits of optimizing multiple prioritization objectives simultaneously is a topic that remains unexplored.

Test case prioritization in HCSs can be driven by functional and non–functional objectives. The former are those based on the functional features of the system and their interactions, such as those prioritization objectives based on CIT [146] or test similarity [5, 59]. The latter consider extra-functional information such as user preferences [42, 68] or cost [146] to find the best ordering for test cases. Generally, current testing techniques for HCSs are based on functional properties extracted from the feature model representing the software system.

Another 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 [5, 58, 105, 149]. 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.

1.3 Thesis goals

We have analyzed the state of the art with regard to the issues described in Section §1.1, and we have identified a set of problems and challenges that constitute the goals of this dissertation described below and summarized in the following global goal:

Dissertation goal

Development of objectives, techniques and tools for test case prioritization in highly-configurable software systems.

This goal can be divided into several sub-objectives, namely:

1. Propose, evaluate and compare different prioritization objectives driven by functional and non–functional properties.
2. Develop single–objective and multi–objective test case prioritization algorithms.
3. Propose HCS realistic case studies to evaluate variability testing techniques.
4. Integrate the contributions of this dissertation into a real testing tool to be useful for the academic community as well as for the industrial community.

1.4 Contributions

In this section, we summarize the main contributions of our research work to address the aforementioned goal. These contributions have been published in journals, conferences and workshops.

1.4.1 Summary of contributions

This thesis delves into the study of testing of HCSs, and in particular, into the prioritization of test cases. The main goal of this work is to provide a set of objectives, techniques and tools to support and improve the effectiveness of testing HCSs. In the pursuit of this goal, we have made the following main contributions:

Functional prioritization objectives for HCSs

Problem statement: To date, test cases have been prioritized using weights such as user preferences or cost. To the best of our knowledge, there are no works analyzing how to exploit the variability information from feature models to guide the prioritization of test cases.

Contribution: We have proposed six different prioritization objectives based on common metrics of feature models. Three of these prioritization objectives are based on the complexity of the configurations, i.e. more complex configurations are given higher priority over less complex ones, i.e. they are tested first. Another prioritization objective is based on the degree of reusability of configurations features, i.e. configurations including the more reused features are given priority during tests. We also propose another objective based on the pairwise coverage, giving priority to those configurations that cover a higher number of pairs of features. Finally, we propose another prioritization objective based on the so-called dissimilarity among configurations, i.e. the more different configurations are tested first. The results showed that different orderings of the same HCS test suite may lead to significant differences in the acceleration of fault detection. The main results of this contribution have been presented in the Seventh IEEE International Conference on Software Testing, Verification, and Validation [114] and part of them was submitted to the Journal on Systems and Software [**?**].

Non–functional prioritization objectives for HCSs

Problem statement: Most of the prioritization papers in HCS are based on typical non–functional objectives such as user preferences. In our opinion, these pieces or work do not take advantage of all available HCS information that can help to define other kind of prioritization objectives to guide the testing.

Contribution: We reported on extensive non–functional data extracted from the Drupal Git repository and the Drupal issue tracking system. With this information we propose seven new prioritization objectives based on non-functional properties such as code size, cyclomatic complexity, number of tests, number of reported installations, number of developers, number of changes and number of faults found on each feature of the system. Also, we have performed a correlation study to explore the correlations between these seven non–functional properties and the fault propensity of software features. Among other results, we found positive correlations relating the number of bugs in Drupal features to their size, cyclomatic complexity, number of changes and fault history. The main results of this contribution have been presented in the Eighth International Workshop on Variability Modelling of Software-Intensive Systems [115] and in the Software and Systems Modeling journal [118].

Multi-objective test case prioritization

Problem statement: To the best of our knowledge, there are few studies in the literature about multi-objective test case prioritization in HCSs. In addition, current approaches are driven mostly by a single objective or by a combination of 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. Thus, the potential benefits of optimizing multiple prioritization objectives simultaneously, both functional and non–functional, is a topic that remains unexplored.

Contribution: We have developed a multi-objective test case prioritization proposal based on the NSGA-II evolutionary algorithm. Additionally, we present seven novel objective functions based on both functional and non–functional properties of the HCS under test. Furthermore, we have conducted a comparison of the effectiveness of 63 different combinations of these seven objectives in accelerating the detection of faults. Results suggested that the combination of multiple prioritization objectives, where at least one is non–functional, typically results in faster fault detection. The main results of this contribution was submitted to the Journal on Systems and Software and a preliminary work was presented in the National Workshop Jornadas de Ingeniería del Software y de Bases de Datos [116].

Industry-strength case studies to evaluate variability testing techniques

Problem statement: The lack of realistic HCSs with available code and fault reports to evaluate testing techniques weakens the conclusions of papers in the literature. Most of works use artificial variability models and simulated faults to evaluate their techniques.

Contribution: We have proposed an industry-strength case study to evaluate variability testing techniques based on Drupal, a highly modular web content management framework. Drupal provides detailed fault reports and its modules includes automated test cases. The high number of the Drupal community members together with its extensive documentation have also been strengths to choose this framework. We have modeled the Drupal variability and extracted useful information such as the number of bugs and changes in modules to carry out the evaluation. In addition to the Drupal case study, we have also presented another HCS case study based on the e-commerce platform Prestashop, a tool for the development of online shopping systems. The main results of these contributions have been published in the Eighth International Workshop on Variability Modelling of Software-Intensive Systems [115], in the Software and Systems Modeling journal [118] and in the 29th ACM/IEEE International Conference on Automated Software Engineering [125].

Integrate the results of this dissertation into an industry-strength tool

Problem statement: The majority of the research breakthroughs have been exploited within the limits of the academic realm instead of being used as a technological path towards industry.

Contribution: We have integrated part of our testing techniques developed in this dissertation on a tool, called SmarTest, for the well-known open-source Drupal web content management framework [18]. SmarTest is a testing module that supports the analysis of the Drupal system to provide useful information to guide the testing. Also, it allows applying different single-objective prioritization testing techniques to reveals bugs faster in Drupal, such as prioritization objectives based on the number of changes made in the code or based on the tests that failed in last executions. The main results of these contributions have been presented in the National Conference DrupalCamp-Spain 2015 [119] and in the International Conference DrupalConEurope 2015 [117].

1.4.2 Publications in chronological order

In this section, we present a list of the publications derived from our research work chronological ordered.

- **[2012]**. During our first year of work, we focused on the introduction and study of the state of the art of automated testing on variability-intensive systems. In particular, we continued working on the metamorphic testing approach carried out by ISA research group. We published our first paper on an automated test data generator for SAT solvers [112].
  - **JISBD'12**. Ana B. Sánchez and Sergio Segura. *Automated testing on the analysis of variability-intensive artifacts: An exploratory study with SAT Solvers*. XVII Jornadas de Ingeniería del Software y de Bases de Datos. Almería, Spain. 2012.
- **[2013]**. We started to study the state of the art of testing techniques with emphasis on the prioritization of test cases. As a result, we presented a paper about the test case prioritization approaches proposed in the literature as well as the trends and challenges identified [113]. Furthermore, we started to work in two papers on test case prioritization techniques for HCSs.
  - **Novática'13**. Ana B. Sánchez and Sergio Segura and Antonio Ruiz-Cortés.

Priorización de casos de prueba. Avances y retos. Novática: Revista de la Asociación de Técnicos de Informática, pág. 27-32. 2013.

- **[2014]**. We presented our first contributions in the context of test case prioritization for HCSs. We define new test case prioritization objectives based on functional properties of HCSs [114]. We introduced an industry-strength case study to evaluate variability testing techniques with actual non-functional data such as fault history and change history of previous versions of the system under test [115]. This work was selected among the best papers at the VaMoS Workshop, and we were invited to extend it in an Special Issue for the Software and System Modeling Journal. We also took a step further in this field proposing a preliminary work towards multi-objective test case generation for HCSs [116]. Finally, we also proposed an experience report on automated variability analysis and testing of an e-commerce site [125].

  – **ICST'14**. Ana B. Sánchez and Sergio Segura and Antonio Ruiz-Cortés. *A Comparison of Test Case Prioritization Criteria for Software Product Lines*. Seventh IEEE International Conference on Software Testing, Verification, and Validation, pages 41-50, Cleveland, Ohio, USA. 2014.

  – **VAMOS'14**. Ana B. Sánchez and Sergio Segura and Antonio Ruiz-Cortés. *The Drupal Framework: A Case Study to Evaluate Variability Testing Techniques*. Eighth International Workshop on Variability Modelling of Software-Intensive Systems. Nice, France. 2014.

  – **JISBD'14**. Ana B. Sánchez and Sergio Segura and Antonio Ruiz-Cortés. *Towards Multi-Objective Test Case generation for Variability-Intensive Systems*. XIX Jornadas de Ingeniería del Software y de Bases de Datos. Cádiz, Spain. 2014.

  – **ASE'14**. Sergio Segura and Ana B. Sánchez and Antonio Ruiz-Cortés. *Automated Variability Analysis and Testing of an E-commerce Site: An Experience Report*. 29th ACM/IEEE International Conference on Automated Software Engineering. Sweden. 2014. CORE: A.

- **[2015]**. We extended the Drupal case study by extracting more information about the components of the system and performed a correlation study to investigate how non-functional properties could be used to predict defects. This work counted on the collaboration of the professor Jose A. Parejo from the ISA group and was published in the journal of Software Systems and Modeling [118]. We also presented an article about automated metamorphic testing of variability analysis tools. This work is a continuation of a research started by our research group in recent years and it includes part of the study presented in JISBD'12. This was a join work with the professor Amador Durán from the ISA group, the professor Daniel Le Berre from the Université d'Artois and the researcher Emmanuel Lonca from the Centre de Recherche en Informatique de Lens, France. The article was published in the journal of Software Testing, Verification and Reliability [126]. In addition to the previous articles, we integrated part of our results into the Drupal web content management [18]. This work resulted in a new module (SmarTest) for testing Drupal that was positively received by the Community. Evidence of this was the presentation of our project in the National Conference DrupalCamp-Spain [119], and the later invitation to present our project in the International Conference DrupalConEurope [117].

- **SoSyM'15**. Ana B. Sánchez and Sergio Segura and Jose A. Parejo and Antonio Ruiz-Cortés. *Variability Testing in the Wild: The Drupal Case Study*. Software and Systems Modeling Journal. 2015. Q2.
- **STVR'15**. Sergio Segura and Amador Durán and Ana B. Sánchez and Daniel Le Berre and Emmanuel Lonca and Antonio Ruiz-Cortés. *Automated metamorphic testing of variability analysis tools*. Software Testing, Verification and Reliability Journal. 2015. Q1.
- **DrupalCampSpain'15**. Ana B. Sánchez and Sergio Segura and Antonio Ruiz-Cortés. *SmarTest: Proposal for accelerating the detection of faults in Drupal*. DrupalCampSpain. Cádiz, Spain. 2015.
- **DrupalConEurope'15**. Ana B. Sánchez and Sergio Segura and Antonio Ruiz-Cortés. *SmarTest: Accelerating the detection of faults in Drupal*. DrupalConEurope. Barcelona, Spain. 2015.

• **[2016]**. At the beginning of this year, a new survey on metamorphic testing has been accepted in the IEEE Transactions on Software Engineering Journal (TSE) in collaboration with the professor Gordon Fraser from the University of Sheffield, United Kingdom. We have also submitted to the Journal of Systems and Software an article that proposes a multi-objective test case prioritization in HCSs and evaluates the approach using the Drupal case study. This is a join work with the professor Jose A. Parejo and the professors Roberto E. Lopez-Herrejon and Alexander Egyed from the Johannes Kepler University, Austria. This article has been submitted in December 2015 and it is in second round of revision.

- **TSE'16**. Sergio Segura and Gordon Fraser and Ana B. Sánchez and Antonio

Ruiz-Cortés. *A survey on Metamorphic Testing*. IEEE Transactions on Software Engineering Journal. Q1.

– **JSS'16**. Ana B. Sánchez and Jose A. Parejo and Sergio Segura and Antonio Ruiz-Cortés and Roberto E. Lopez-Herrejon and Alexander Egyed. *Multi-Objective Test Case Prioritization in Highly-Configurable Systems: A Case Study*. Journal of Systems and Software. **Submitted** (second round of revision). Q1.

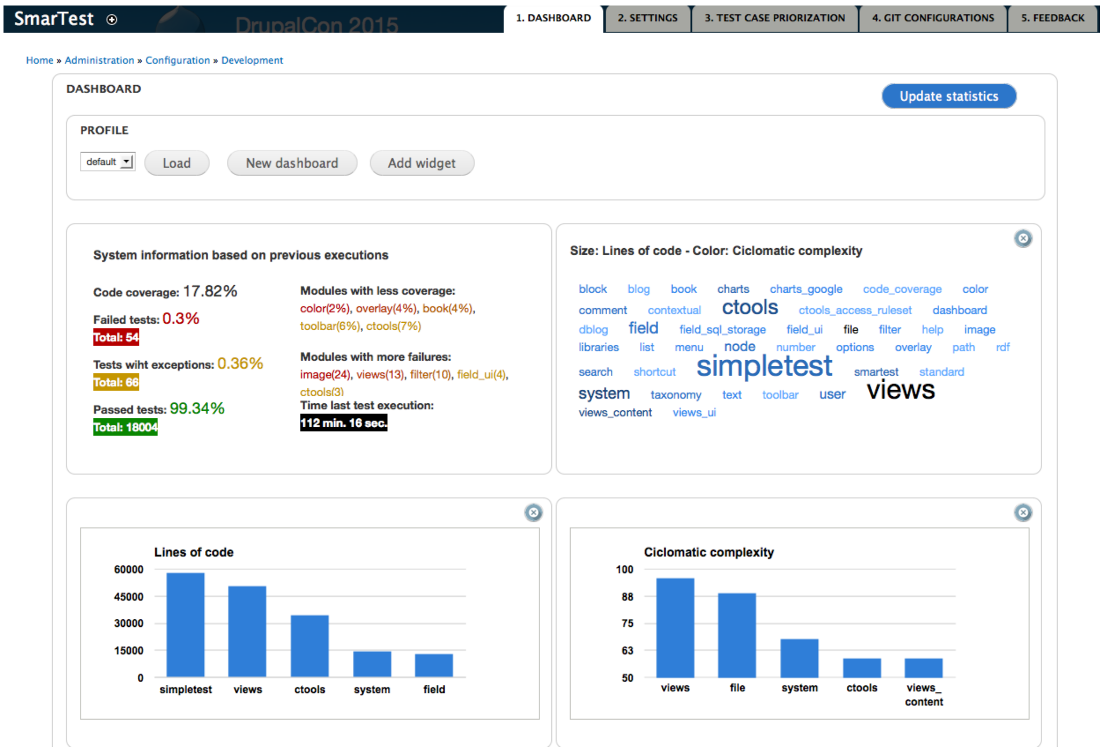
1.4.3 Developed tool

Part of the results of this thesis have been integrated into a module for the open-source Drupal web content management framework [18], that we have coined as SmarTest. *SmarTest* [117, 119] is an open-source module for improving the process of testing in Drupal [18]. Drupal is a highly modular open source web content management framework implemented in PHP [18, 140]. It can be used to build a variety of web sites including internet portals or e-commerce applications. SmarTest extends the functionality of SimpleTest, the default testing module of Drupal, by integrating our single-objective test case prioritization techniques, e.g. prioritization based on the changes (Git commits) found in the modules. SmarTest also provides the Drupal developer with a dashboard with statistics about the Drupal system in real time (tests that failed in last executions, percentage of code covered by tests, cyclomatic complexity per module, and so on), providing faster feedback to the testers and reducing debugging efforts. Images in Figure §1.3 show some screenshots of SmarTest. Figure 1.3(a) illustrates part of the dashboard developed in SmarTest to provide information to testers about the status of the system. Figure 1.3(b) displays the prioritized testing in progress for SmarTest. A more detailed description of the tool is presented in Appendix §A and in the SmarTest's web page[1].

1.4.4 Thesis context

This thesis has been developed in the context of the Applied Software Engineering (Ingeniería del Software Aplicada - ISA) research group of the University of Seville, and more specifically on the context of Automated Testing and Software Product Lines areas, as a holder of a pre-doctoral research scholarship from the THEOS (Tecnologías Habilitadoras para EcOsistemas Software) Andalusian R&D&I project. Besides THEOS project, there is a number of research projects that have made this dissertation possible:

[1]http://www.isa.us.es/smartest/index.html


SmarTest
DrupalCon 2015
1. DASHBOARD
2. SETTINGS
3. TEST CASE PRIORIZATION
4. GIT CONFIGURATIONS
5. FEEDBACK
Home » Administration » Configuration » Development
DASHBOARD
Update statistics
PROFILE
default
Load
New dashboard
Add widget
Size: Lines of code - Color: Ciclomatic complexity
block blog book charts charts_google code_coverage color comment contextual ctools ctools_access_ruleset dashboard dblog field field_sql_storage field_ui file filter help image libraries list menu node number options overlay path rdf search shortcut simpletest smartest standard system taxonomy text toolbar user views views_content views_ui
Lines of code
60000
45000
30000
15000
0
simpletest
views
ctools
system
field
Ciclomatic complexity
100
88
75
63
50
views
file
system
ctools
views_
content

System information based on previous executions

Code coverage: 17.82%

Failed tests: 0.3%
Total: 54

Tests wiht exceptions: 0.36%
Total: 66

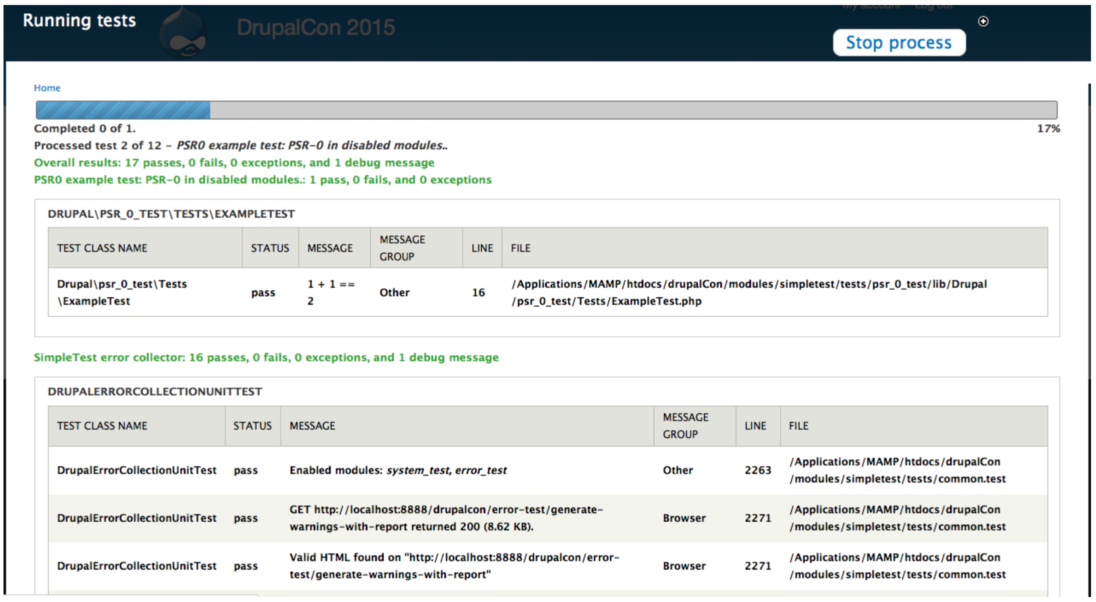
Passed tests: 99.34%
Total: 18004

Modules with less coverage:
color(2%), overlay(4%), book(4%), toolbar(6%), ctools(7%)

Modules with more failures:
image(24), views(13), filter(10), field_ui(4), ctools(3)

Time last test execution:
112 min. 16 sec.

(a) SmarTest's dashboard view



Completed 0 of 1. 17%
Processed test 2 of 12 – *PSR0 example test: PSR–0 in disabled modules..*
Overall results: 17 passes, 0 fails, 0 exceptions, and 1 debug message
PSR0 example test: PSR–0 in disabled modules.: 1 pass, 0 fails, and 0 exceptions

DRUPAL\PSR_0_TEST\TESTS\EXAMPLETEST

TEST CLASS NAME	STATUS	MESSAGE	MESSAGE GROUP	LINE	FILE
Drupal\psr_0_test\Tests \ExampleTest	pass	1 + 1 == 2	Other	16	/Applications/MAMP/htdocs/drupalCon/modules/simpletest/tests/psr_0_test/lib/Drupal /psr_0_test/Tests/ExampleTest.php

SimpleTest error collector: 16 passes, 0 fails, 0 exceptions, and 1 debug message

DRUPALERRORCOLLECTIONUNITTEST

TEST CLASS NAME	STATUS	MESSAGE	MESSAGE GROUP	LINE	FILE
DrupalErrorCollectionUnitTest	pass	Enabled modules: *system_test, error_test*	Other	2263	/Applications/MAMP/htdocs/drupalCon /modules/simpletest/tests/common.test
DrupalErrorCollectionUnitTest	pass	GET http://localhost:8888/drupalcon/error–test/generate–warnings–with–report returned 200 (8.62 KB).	Browser	2271	/Applications/MAMP/htdocs/drupalCon /modules/simpletest/tests/common.test
DrupalErrorCollectionUnitTest	pass	Valid HTML found on "http://localhost:8888/drupalcon/error–test/generate–warnings–with–report"	Browser	2271	/Applications/MAMP/htdocs/drupalCon /modules/simpletest/tests/common.test

(b) SmarTest's prioritization testing view

Figure 1.3: SmarTest tool for Drupal

COPAS (eCosystems for Optimized Process As a Service) project, TAPAS (Tecnologías Avanzadas para Procesos como Servicios) project and SETI (reSearching on intElligent Tools for Internet of services) project.

Additionally, the PhD candidate has completed one research stay, has attended a research summer school and has shared a fruitful collaboration with some international researchers:

- 8th International Summer School on Training And Research On Testing (TAROT'12). Métabief, France. July 2 - 6, 2012.
- A three-month research stay in the University of Nebraska-Lincoln, United States, under the supervision of the professor Myra B. Cohen. October 2014 - January 2015.
- International collaborations with the distinguished researchers Gordon Fraser, Roberto López Herrejón and Daniel Le Berre, with whom we have some articles.

1.5 Structure of this dissertation

This document is structured as follows:

Part I. Preface. It comprises this introduction chapter.

Part II. Background information. In this part, we provide the reader with a deep understanding of the research context in which our work has been developed.

Part III. Contributions. This part is the core of our dissertation since we present the contributions performed throughout this thesis.

Part IV. Final remarks. In this part, we report our main conclusions and our plans for future research.

Part V. Appendices. In appendix §A, we present the SmarTest tool which includes part of our research contributions in the field of testing HCSs.

Part II

Background information

Feature models

Nature is an endless combination and repetition of very few laws.
She hums the old well-known air through innumerable variations.

Sir Ralph Waldo Emerson. Essays, Lectures and Orations (1851),

Feature models are recognized in the literature to be one of the most important contributions to software product line engineering. They were introduced 26 years ago as a way to model variability in software product lines and later applied to other highly-configurable systems. In this chapter, we survey the most common notations for feature modeling providing some examples. In Section §2.1, we describe feature models. Section §2.2 presents the classical notation of feature models also referred to as basic feature models. Cardinality-based feature models are presented in Section §2.3. Extended feature models with attributes are introduced in Section §2.4. Finally, we summarize the main points of the chapter in Section §2.5.

2.1 Introduction

Software development is progressively transitioning from the production of individual programs to the production of families of related programs. This introduces a new dimension in the development process referred to as configuration. Configuration is the process of binding the optional features of a system to realizations in order to produce a specific software system, i.e., a member of the family [22]. This provides the production of multiple software systems that share a common set of mandatory requirements, but also have optional capabilities. Frequently, significant reuse can be achieved by implementing a set of these systems as one integrated highly-configurable software system. Configuration must be correctly managed at all levels. For instance, requirement engineers must be aware of which requirements are mandatory and which ones are optional. Similarly, design engineers must know what features are incompatible and which ones depend on each other. Also, testers must know which are the configurations that can be derived from the highly-configurable system in order to design their testing plans. In this context, feature models are one of the most common artifacts for variability management in highly-configurable systems [69]. A feature model provides a compact representation of all the configurations of an HCS in terms of features where a feature is any domain abstraction relevant for the stakeholder [138]. More specifically, a feature model is a tree-like structure and consists of:

- Nodes representing features.
- Relationships between a parent feature and its child features.
- Cross-tree constraints that are typically inclusion or exclusion statements of the form: "If feature *A* is included, then feature *B* must also be included (or excluded)".

Feature models were first introduced as a part of the Feature-Oriented Domain Analysis method (FODA) by Kang et al. in 1990 [69]. Since then, many extensions to the original proposal have been presented and a consensus about a feature model notation has not been reached yet. The goal of this chapter is to provide an overview of the feature modeling notations that we will refer to throughout the rest of this dissertation.

2.2 Basic feature models

We classify as basic feature models those using simple relationships between features. There are two notations that we have classified as basic feature models: FODA feature models and feature-RSEB feature models. Next, we detail them.

In 1990, [69] Kang et al. proposed the feature model notation called Feature-Oriented Domain Analysis method (FODA). In this approach, three different kind of relationships between features were defined:

- **Mandatory**. A child feature has a mandatory relationship with its parent when the child is included in all configurations in which its parent feature appears. Mandatory relationships are generally modeled using a filled circle as shown in

Figure 2.1(a). For instance, according to the feature model of Figure §2.3, it is mandatory to include support for *Payment* in an online electronic shop.

- **Optional**. A child feature has an optional relationship with its parent when the child can be optionally included in all configurations in which its parent feature appears. Optional relationships are generally represented using a empty circle as shown in Figure 2.1(b). For instance, support for *Search* feature is optional in the feature model of Figure §2.3.

- **Alternative**. A set of child features have an alternative relationship with their parent when only one feature of the children can be selected when its parent feature is part of the configuration. Figure 2.1(c) depicts the usual visual representation for this relationship. As an example, according to the feature model of Figure §2.3, an online shop may use a *High* or an *Standard* security policy but not both in the same configuration.

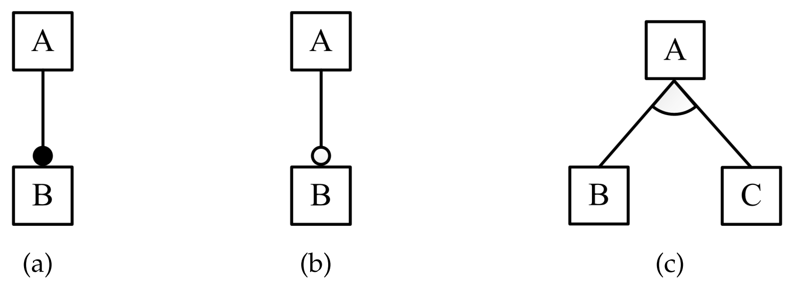


Figure 2.1: Feature relationships: a) Mandatory; b) Optional; c) Alternative

Notice that a child feature can only appear in a configuration if its parent feature does. The root feature is a part of all the configurations within the highly-configurable system.

Additionally, in 1998, Criss et al. [55] presented an extension of FODA feature models that is usually referred as Feature-RSEB. RSEB is an use-case driven systematic reuse process in which variability is explicitly modeled by means of variation points and variants. The authors of this approach proposed extending the FODA feature models with a new relationship between features. This is:

- **Or**. A set of child features are said to have an or relation with their parent when one or more of them can be included in the configurations in which its parent feature appears. Figure §2.2 depicts the usual visual representation used for this type of relationship. As an example, according to the feature model of Figure

§2.5, an online shop can provide payment support for *Bank Transfer*, *Credit Card* or the combination of both at the same time.

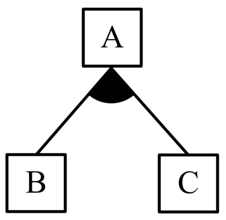


Figure 2.2: Or relationship

In order to clarify the concepts above explained about basic feature models we show some examples. Figure §2.3 presents a simplified feature model representing the variability of a highly-configurable system of online shopping systems. Their configurations must include mandatory features such as *Catalogue* management or *Payment* support and may include optional features such as *Search*. Furthermore, shopping system must include and specific security policy (*High* or *Standard*) and may include one or two different payment modules. Finally, cross-tree constraints indicate that payment with *Credit Card* requires a *High* security policy and *Public Report* search cannot be included in shopping systems with *High* security policy.

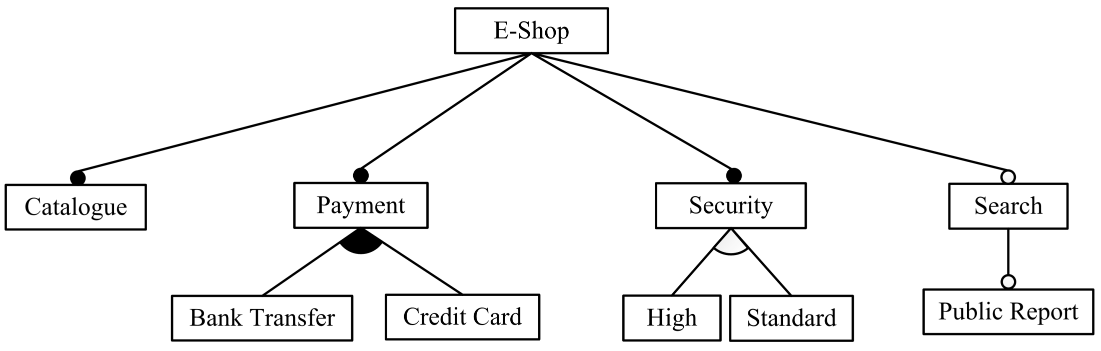


Figure 2.3: E-shop feature model (without cross-tree constraints)

In addition to the parental relationships between features, two kinds of cross-tree constraints between features are allowed in FODA. These are:

- **Requires**. If a feature *A* requires a feature *B*, the inclusion of *A* in a configuration implies the inclusion of B in such configuration. Requires constraints are commonly modeled using unidirectional arrows as shown in Figure 2.4(a). For instance, according to the feature model of Figure §2.5, online shops including *Credit Card* payment require support for *High* security policy.

- **Excludes**. If a feature *A* excludes a feature *B*, both features cannot be part of the same configuration. These constraints are visually represented using bidirectional arrows as shown in Figure 2.4(b). As an example, the feature model of Figure §2.5 rules out the possibility of offering support for *High* security policy and *Public Report* in a search in the same configuration.

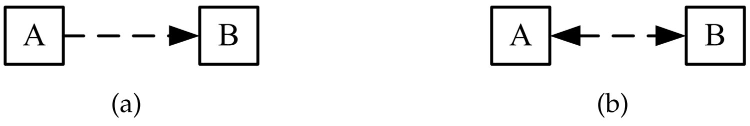


Figure 2.4: Cross-tree constraints: a) Requires; b) Excludes

An extended version of the previous example is shown in Figure §2.5. It includes cross-tree constraints. The requires cross-tree constraint implies that any shopping system including *Credit Card* payment must also include *High* security support. In a similar way, the excludes cross-tree constraint removes the possibility of providing support for *High* security and *Public Report* search in the same configuration.

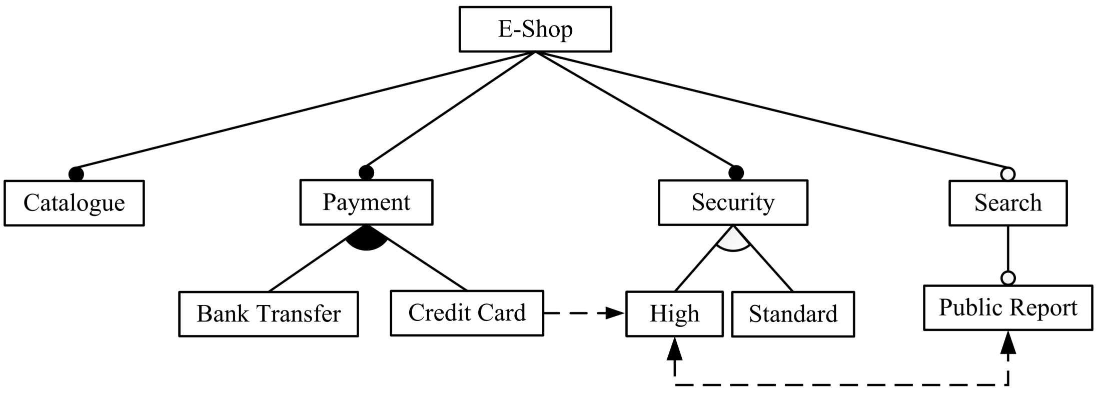


Figure 2.5: E-shop feature model

Figure §2.6 depicts a feature model inspired by the smart home industry. The model illustrates how features are used to specify and build software for smart homes. The software loaded in the smart home is determined by the features that it supports. According to the model, all smart homes must include support for lighting and one or more control systems (i.e. cell phone or control panel or both of them). Furthermore, the smart homes may optionally include support for anti-theft alarm, movie players, contents and Internet among other optional features. Also, some features like *Internet* provide different kind of support (e.g. *3G*, *Ethernet*, *Wifi-n* or *Wifi-b* internet connections or more than one at the same time). According to the Figure §2.6, the smart home systems including *Anti-theft Alarm* must also include *Control Panel* system, and those smart homes supporting *Video on Demand* contents must support *Internet* connection too.

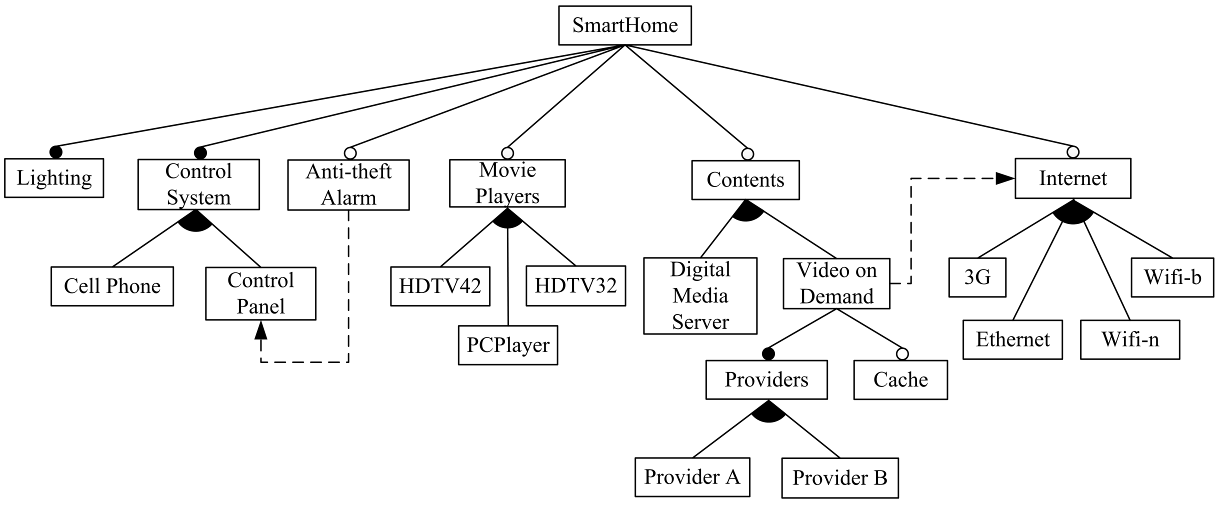


Figure 2.6: Smart Home feature model

2.3 Cardinality-based feature models

Some authors proposed extending FODA feature models with cardinalities (also known as multiplicities) [23, 24, 107, 108]. The main motivation to extend feature models with cardinalities was that some cases could not be modeled using *alternative* and *or* relationships.

Riebisch et al. [107, 108] proposed using *mandatory* and *optional* relationships as in the original FODA proposal and replacing *alternative* and *or* relationships with a new relationship:

- **Set**. A *set* of child features are said to have a set relationship if a number of features can be included in a configuration when their parent feature is included. This number depends on the cardinality. The *cardinality* is an interval, $< n..n' >$, with n as lower bound and n' as upper bound limiting the number of child features that can be part of a configuration. For instance, in the *set* relationship of Figure §2.7, the number of child features that can be included in a configuration is limited to 1. Thus, an *alternative* relationship in the original FODA proposal is equivalent to a *set* relationship with cardinalities $< 1..1 >$. Likewise, an *or* relationship is equivalent to a multiplicity relationship with cardinalities $< 1..N >$ being N the number of features in the relationship.

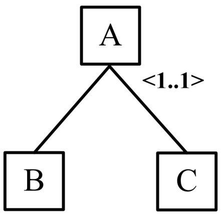


Figure 2.7: Set relationship with cardinality

Later, Czarnecki et al. [23, 24] proposed using the *mandatory*, *optional* and *set* relationships previously detailed. Additionally, they also introduced a new relationship:

- **Feature cardinality**. A *feature cardinality* is a sequence of intervals of the form $[n..n']$ with n as lower bound and n' as upper bound. These intervals restrict the number of instances of the feature that can be part of a configuration. For instance, according to the feature cardinality showed in Figure §2.8, the number of instances of the feature B that can be included in a configuration is restricted to 1. Notice that this relationship may be used as a generalization of the original *mandatory* and *optional* relationships defined in FODA. A *feature cardinality* of $[0..1]$ would be equivalent to an *optional* relationship meanwhile a *feature cardinality* of $[1..1]$ would be equivalent to a *mandatory* relationship.

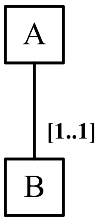


Figure 2.8: Feature cardinality

As an example, Figure §2.9 depicts a cardinality-based version of the basic feature model presented in Figure §2.6. The model was built by replacing the *alternative* and *or* relationships of the original model with *set* relationships. *Cardinalities* were adjusted conveniently to make the model represents the same configurations than the original model. In particular, *set* relationships with *cardinality* $< 1..N >$ (being N the number of children) were used instead of *or* relationships.

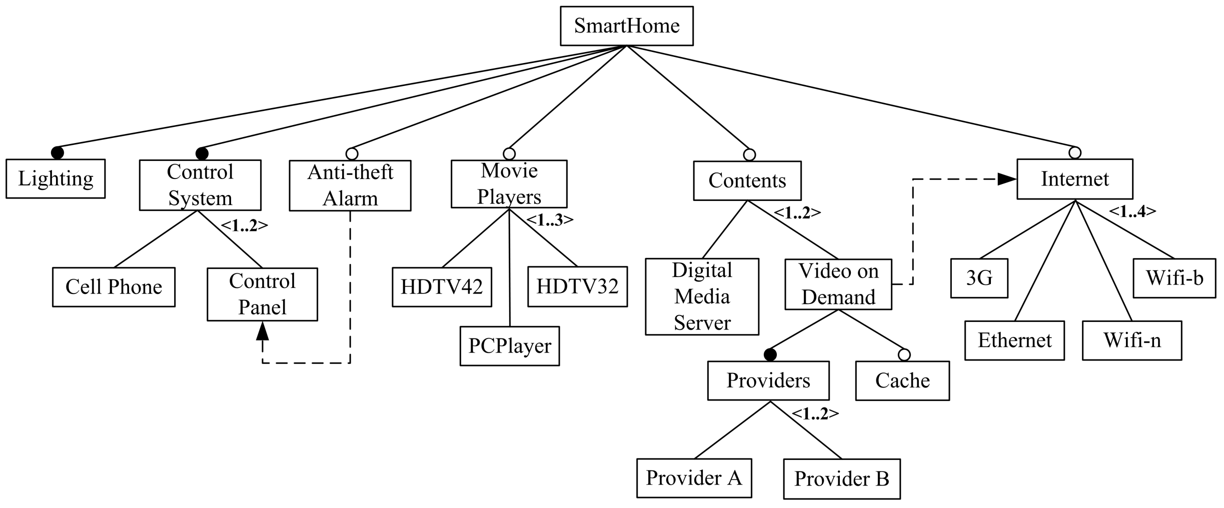


Figure 2.9: Cardinality-based Smart Home feature model

2.4 Attributed feature models

Basic feature models are mainly used to manage functional commonality and variability in highly-configurable software systems. However, sometimes it is necessary to extend feature models to add more information to them. To this aim, several works propose adding attributes to the features of feature models [12, 13, 24, 45]. These types of feature models in which extra information is included by means of attributes are usually referred in the literature as attributed, extended or advanced feature models [10, 11, 13].

For instance, Figure §2.10 depicts a partial attributed feature model using the notation proposed in [13]. As illustrated in the figure, an attribute mainly consists of:

- **Attribute name**. Name or id of an attribute. For instance, the cost attribute of the feature *3G* in Figure §2.10.
- **Attribute domain**. It specifies the space of possible values for an attribute. Every attribute has an associated domain. It is possible to have discrete domains (e.g. integers, booleans, enumerated values) or continuous domains (e.g. real). For instance, the domain of *Cost* attribute is Real values in Figure §2.10.
- **Attribute value**. The value of an attribute. This could be an specific value within the domain or an expression depending on the value of other attributes of the same or other features. A default value for the cases in which the feature is not

selected could also be included. For instance, the cost of selecting the feature *3G* in Figure §2.10 is a basic attribute defined by $cost = 50.8$.

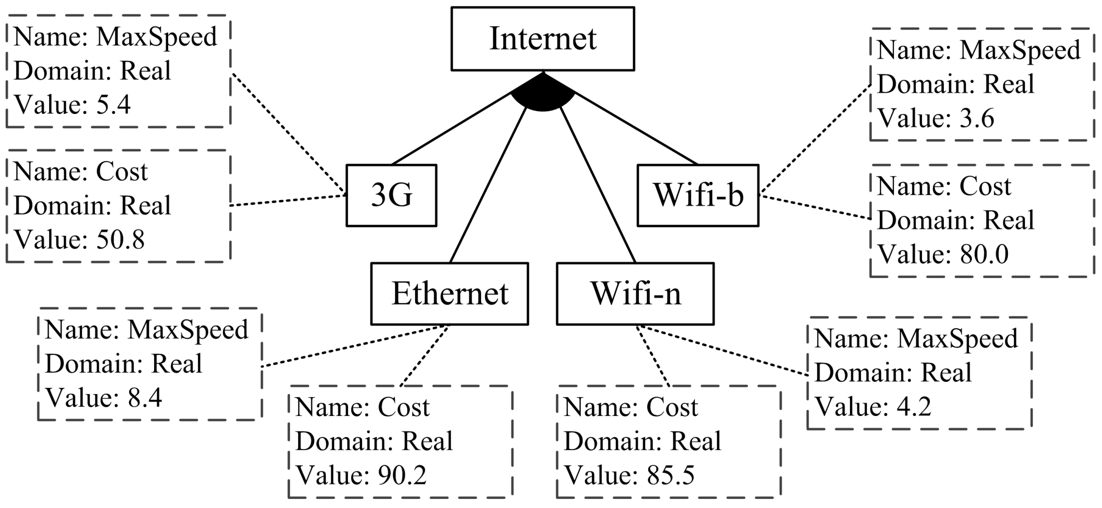


Figure 2.10: Attributed features in a feature model

2.5 Summary

In this chapter, we have summarized the most common feature modeling notations found in the literature. Two main groups of feature model notations are used: basic feature models and cardinality-based feature models. The main difference is that cardinality-based feature models provide support for more complex relationships that the former one. In particular, we have presented the concepts of basic feature models and attributed feature models. As a possible complement to both of them, a third notation, attributed feature models, proposes adding extra functional information to the models by means of feature attributes. Some examples have been also described to guide the comprehension of the different notations. For a more extensive and rigorous survey of feature modeling languages we refer the reader to [121].

Test case prioritization

Action expresses priorities.

Mahatma Gandhi,

The concept of test case prioritization was introduced by Rothermel et al. in 1999. Test case prioritization techniques schedule test cases in an execution order according to some criterion. In this chapter, we describe and classify the main test case prioritization objectives and approaches identified in this research area. In Section §3.1, we introduce the test case prioritization concept. Section §3.2 presents a classification and description of test case prioritization objectives and approaches found in the literature. Part of the most frequently cited prioritization proposals is described in Section §3.3. In Section §3.4, we introduce the concept and some works of multi-objective test case prioritization. The most relevant metrics to evaluate test case prioritization techniques have been summarized in Section §3.5. Finally, we describe the main points of the chapter in Section §3.6.

3.1 Introduction

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 [5, 37, 78, 110, 111, 150]. The approach was introduced by Rothermel et al. in 1999 [110]. Since then, many authors have explored the prioritization of test cases as an alternative to improve the performance of testing. Different goals can be identified in the literature for prioritizing test cases, although, in practice, only one prioritization goal is applied, the acceleration of detection of faults. We will focus exclusively on that goal. For its application, test cases are ordered according to their ability to detect faults. However, that information is typically not known until all test cases are executed. In order to overcome the difficulty of knowing which tests reveal faults, prioritization approaches depend on representative objectives related with the propensity to faults. For example, we could accelerate the detection of faults based on the complexity of the components under test, testing first the most complex ones assuming that these are the most error-prone. If faults are detected, these could be fixed as soon as possible reducing the total time needed for testing and debugging. Furthermore, if the execution of tests is prematurely halted at some arbitrary point we would know that the most critical components have already been tested.

The goal of this chapter is to review the state of the art in the context of the test case prioritization approaches. In particular, we study and classify the test case prioritization objectives and approaches found in the literature. We also introduce the test case prioritization approaches based on multiple objectives and the techniques used for that optimization. Then, we describe metrics used to evaluate the prioritization techniques in this research area.

3.2 Classification of test case prioritization approaches

We propose to classify the proposals in the test case prioritization field based on the following points:

- **Objective**. This is the order criterion used to accelerate the detection of faults. Different objectives may be proposed. For instance, in order to accelerate the detection of faults, testers could order the execution of test cases according to the number of faults detected by the test cases in previous executions of the suite, or according to the complexity of the components under test.
- **Approach**. This is the method used to implement the prioritization objective. Given a prioritization objective, we can found different approaches to implement it. For instance, to order the test cases based on the software complexity, we could take as a reference the developer's knowledge or the use of metrics measuring the complexity (e.g. cyclomatic complexity).

Table §3.1 shows a summary of prioritization objectives and approaches identified in the literature. In the following, these prioritization objectives and approaches are described in detail.

Objective	**Approaches**
Based on code coverage	[17, 20, 31, 44, 64, 71, 76, 82, 105, 111, 146]
Based on cost	[44, 48, 62, 86, 134, 146]
Based on fault history	[44, 62, 71, 129, 134]
Based on similarity of tests	[5, 59, 76, 82, 129]
Based on customer's requirements or goals	[42, 72, 133, 135]
Based on capability to detect faults	[48, 105, 146]
Based on software complexity	[72, 133, 135]
Based on component usage	[29, 48]
Based on requirement volatility	[133, 135]
Based on fault severity	[148]
Based on change history	[128]
Based on execution history	[97]
Based on number of tests	[146]
Based on fault proneness	[135]
Based on component coupling	[136]
Based on memory consumption	[82]
Based on footprint	[82]
Based on downloads statistics	[68]

Table 3.1: Classification of prioritization objectives and approaches

3.3 Prioritization objectives

In this section, we present some of the prioritization objectives and approaches most frequently cited in the literature. In order to facilitate their understanding, we will refer to the example shown in Table §3.2, which presents actual information on four modules obtained from the open source e-commerce solution Prestashop 1.5 [46]. For each module, we collected the number of bugs found in Prestashop 1.5 obtained from the issue tracking system of Prestashop[1] and the number of changes made from its github project[2]. This information was collected from the period of 1 May 2013 to 30 June 2013.

	PayPal	**BlockCart**	**ProductsCategory**	**FavoriteProducts**
Number of faults	27	37	15	3
Number of changes	13	16	1	3

Table 3.2: Number of faults and changes of some Prestashop 1.5 modules

3.3.1 Fault history

This objective is used to accelerate the detection of faults in general or critical faults in particular. The test cases are ordered according to the fault history of previous versions of the system, executing first those test cases that validate the components that have been more error-proneness in previous versions [19, 109, 150].

As an example, consider that we want to test a new version of Prestashop system following the fault history objective. Based on Table §3.2, we would run first the test cases of *Blockcart* since this has been the most error proneness module in previous versions (37 faults), following by the module *Paypal* (27 faults), *ProductsCategory* (15 faults) and *FavoriteProducts* (3 faults).

In [71], the authors assign binary priorities to test cases based on whether they revealed faults (1) or not (0) in earlier versions of the software. In [134], the tests are reordered considering their capacity to detect failures (using failure history of previous versions) and also, considering the cost of test setup required each time a configuration is modified. Simons et al. [129] cluster and prioritize the test cases by using test information available from previous versions. Epitropakis et al. [44] study multiple

[1]http://forge.prestashop.com/secure/ Dashboard.jspa
[2]https://github.com/PrestaShop

prioritization objectives such as the average percentage of past fault coverage based on previous works showing that tests that detected faults in the past may tend to be more effective than those that did not.

In [62], the authors propose to accelerate the detection of critical faults from the information about the test costs and the number and severity of faults detected in the latest regression testing. Then, a genetic algorithm is used to find an order with the greatest rate of critical fault detected per the test cost.

3.3.2 Change history

In this objective, test cases are executed according to the number of changes made in the components involved in them in previous versions.

The objective is based on the fact that changes made in components of a system, often may add a new fault in the code [19, 109, 150]. For instance, given the changes made in the modules of Table §3.2, we would execute before the test cases associated to the module *BlockCart* (16 changes), following by the tests of *Paypal* (13 changes), *FavoriteProducts* (3 changes) and *ProductsCategory* (1 change).

Sherriff et al. [128] propose a methodology for prioritizing regression test cases by gathering software change records. Furthermore, this approach identifies clusters of files that historically tend to change together.

3.3.3 Execution history

Test cases are ordered using test execution history collected from previous versions. The data collected can include the number of test cases in prior execution or the number of detected faults in similar prior execution (objective based on fault history).

Based on that objective, the test cases of a configuration would be executed according to the execution data of a similar prior configuration. Thus, the test cases associated to a configuration whose similar prior configuration had more number of test cases involved and more number of detected faults would be executed first.

In [97] a framework is proposed to prioritize test cases for black box testing on new configurations using the test execution history collected (number of tests and faults detected) from similar prior configurations and the Ant Colony Optimization.

3.3.4 Number of tests

This objective tries to maximize the number of prioritized test cases of a system.

In practice, considering limited budget (i.e., available time and resources), it is usually not feasible to execute all the possible test cases for testing the configurations, thus it requires an strategy to prioritize the given test cases. This objective measures the prioritization for the given test cases achieved within the given available test resources. As an example, if we have two possible test case ordering, we would choose for testing that ordering involving the highest number of test cases. This objective is often combined with other prioritization objectives.

In [146], the authors propose a search-based multi-objective test prioritization technique including the maximization of number of prioritized test cases with limited execution cost.

3.3.5 Capability to detect faults

This objective is similar to the objective based on fault history. It measures the fault detection capability achieved by a test suite.

Usually, fault detection capability refers to the probability of a test case to detect faults. More specifically, the execution of a test case can be defined as a success if it can detect faults given test resources (e.g. time) and as a fail if it does not detect any fault. For instance, if a test case is normally executed 1000 times per week and if it executes successfully for 800 times, its capacity to detect faults can be calculated as $800/1000 = 0.8$, i.e. 80%.

In [146], it is maximized the fault detection capability using the successful rate of execution of test cases, understanding the success of a test case as the detection of faults in a given time. Ferrer et al. [48] proposed a prioritization based on an error model that reflects distribution of error probabilities of software components. Components with a high probability of revealing an error have higher priority to be tested than components with a low probability. Qu et al. [105] applied a prioritization method that measures the percentage of faults found by a given test suite to guide the testing.

3.3.6 Fault severity

This objective tries to accelerate the detection of critical faults. Test cases are ordered based on the severity of the faults found by them in previous executions.

As an example, consider the data in Table §3.2. Suppose that while the module *Paypal* detects 27 faults, all of them have a severity level of minor and that the module *ProductCategory* detecting just 15 faults, all of them are critical faults. We would execute before the test cases associated to *ProductCategory* although it detects a lower number of faults.

In [148], a method is proposed to prioritize the test cases based on the fault severity. In order to weigh the degree of importance of each test case, they enumerate all faults which can be found by a test case, accumulate fault severities of those faults and reorder test cases according to that information.

3.3.7 Customer's requirements or goals

In this objective, the customer's requirements are used to assign priorities to the system components [19, 109, 150].

As an example, suppose that the customer of an online shopping system considers as relevant parts of the system (specified in the requirements) those ones related with the module *BlockCart*. Thus, this module receives a customer's priority of 10 out of 10, followed by the module *ProductsCategory* with a priority of 8, the module *FavoriteProducts* with priority of 4 and *Paypal* with priority of 3. These priorities will set the execution order of the test cases.

In [72], [133] and [135] the authors propose to accelerate the detection of faults by prioritizing the test cases taking into account the requirements priority assigned by the customer. The higher the value, the higher the priority of the test case related to this requirement to be executed.

Ensan et al. [42] suggest a prioritization method for product line test cases which is based on the preferences of the domain stakeholders' goals and objectives. Code metrics could also be used (e.g. cyclomatic complexity) to obtain quantitative complexity measures.

3.3.8 Software complexity

This objective orders the execution of test cases according to the complexity of software development. This information will be provided by the developers of the program based on their experience during the implementation phase.

The objective is based on the assumption that the most difficult components to develop will be more error-prone. Thus, the test cases with a higher complexity are executed earlier [19, 109, 150].

In [133], [72] and [135] the authors propose to prioritize the test cases by assigning weights ([1-10]) to measure the complexity of the software (or requirement) that the test cases validate. The higher the weight, the higher the execution priority of the associated test case.

3.3.9 Similarity of tests

Some studies show that similar test cases (those running parts of common code) are redundant from the point of view of the discovery of new bugs. However, the test cases that differ more between them are more likely to detect unexpected behavior [19, 59, 109, 150]. Hence, the most dissimilar test cases will be executed first.

In [76], a proposal for test case prioritization is presented. This approach is based on the similarity of test cases according to the code they cover. In [129], the authors prioritize test cases taking into account their similarities to cluster them according to their ability to detect faults in previous versions. Henard et al. [59] also propose the prioritization based on similarities to generate software product line test cases. In [5], it presents a similarity-based prioritization to be applied on these products of a software product line before they are generated. The proposed approach does not guarantee to find more errors than sampling approaches, but it aims at increasing interaction coverage of an product line under test as fast as possible over time. Also, authors in [82] select the products of an SPL to prioritize based on how dissimilar they are when compared to all other products, and assigned them priority weights based on their rank values.

3.3.10 Component coupling

This objective is based on the system components dependencies to prioritize the test cases.

This kind of objective is often reflected in proposals based on models (UML models, graphs, state models, etc.), that take advance of their structure for representing system components and dependencies between them [19, 136]. Thus, we will execute earlier those test cases with higher coupling among their components, since this may be a sign of occurrence of faults.

Tahat et al. [136] presents a prioritization approach using models describing system behaviors through a set of states and transitions between states. In order to prioritize the test cases, the authors assign a high priority to those ones that execute the transitions modified (added or deleted) in the model and a low priority to the test cases that are not running any modified transition.

3.3.11 Component usage

This objective prioritizes the test cases of a system based on the use of its components.

For instance, we can find works that base their test case prioritization on an usage model that represents the distribution of components in terms of usage scenarios. Thus, those components with a high occurrence have higher priority to be tested than those with a low occurrence.

Ferrer et al. [48] proposed to prioritize test cases according to the importance of the components related with the test cases. This importance is defined according to an usage model that reflects the usage distribution of all components. In [29], the authors presented an approach to prioritize the products of a software product line to test according to the probability of their execution traces representing their usages.

3.3.12 Code coverage

This objective pursues to execute as much code as possible sooner. Thus, test cases are ordered according to the code coverage achieved in previous executions, running first the test cases with higher coverage.

As an example, assuming fictitious coverage percentages for modules in Table §3.2, we could define the following execution order: test cases of *Paypal* (45% of code coverage), *BlockCart* (25%), *ProductsCategory* (20%) and *FavoriteProducts* (10%). Often, different test cases exercise common parts of the code. In theses cases, we can assign priorities taking into account not only the total code coverage percentage but also the percentage of new code not covered by previous test cases.

In [76], the authors describe a test case prioritization approach based on additional coverage. Thus, they first select the test case covering the largest percentage of code, then, they select the test case covering the largest percentage of code not covered by the first one, and so on.

Rothermel et al. [111] performed several proposals for prioritization grouped into the following categories: a) proposals ordering tests based on total code coverage, b) proposals ordering tests based on code coverage of components that have not been covered previously, c) proposals ordering tests based on the estimation of the ability to reveal bugs in the code that cover. Furthermore, in [71] it is presented a prioritization technique that gives a higher priority to those test cases that cover functions that were rarely covered in previous test sessions.

Bryce et al. [17] developed prioritization criteria for event-driven software based on parameter-value interaction coverage, count–based coverage and frequency–based coverage. The first group gives priority to test cases with a large number of parameter values interactions, the second one prioritizes the test cases based on counts of the number of windows, actions or parameter values that they cover, and the last one gives higher priority to test cases that cover windows that are perceived to be important to the event-driven software from a testing perspective.

Do et al. [31] prioritize test cases according to four techniques: block-total, block-addtl, method-total and method-addtl. Thus, they first prioritize test cases by the total number of code blocks they cover by sorting them in terms of that number. Then, they prioritize test cases in terms of those numbers of additional blocks they cover by greedily selecting the test case that covers the most as–yet–uncovered blocks until all blocks are covered. The third and fourth techniques, method-total and method-addtl, are the same as corresponding block level techniques except that they rely on coverage measured in terms of methods.

In [44], the authors present a test case prioritization criterion based on statement coverage, i.e., one has to at least execute the faulty statement in order to detect it. Here, statement coverage is used as the surrogate for fault detection capability. Also, they propose use the information about the difference of the statement coverage between two consecutive versions. In a regression testing scenario, one may conjecture that new regression faults are likely to originate from the changed parts of the source code in the version under test. Therefore, the coverage of the changed parts, obtained with diff, is a rational candidate for prioritization.

In [64], the authors propose a family of coverage–based adaptive random test case prioritization techniques using different level of coverage information such as the total number of statements covered and its additional alternative one, total number of functions covered and its additional alternative one and finally, the total number of branches covered and its additional alternative one.

Wang et al. [146] introduced a test prioritization technique considering feature pairwise coverage, i. e., try to cover as much pairs of features as possible. They chose this technique based on domain knowledge and history data about faults since a higher percentage of detected faults are mainly due to the interactions between test functionalities that can be represented as features in a feature model. Similarly, Chen et al. [20] used a hybrid approach to build prioritized pairwise interaction test suites.

Herrejon et al. [82] presented a genetic algorithm for the generation of prioritized pairwise testing suites for software product lines. This approach adds priority weights to the products that cover each pair of features and then uses these weights to compute the total weight of all the pairs of features. Thus, it selects first the test cases that covers weighted pairwise configurations with higher weights. In [105], the author use combinatorial interaction testing techniques to model and generate prioritized configuration samples for use in regression testing. The technique uses the interaction importance of individual factors and values to determine the final configuration order.

3.3.13 Requirement volatility

In this objective, test cases are executed in an order that take into account the changeability of the requirements associated to them. This is based on the idea that changes in a requirement can be considered as a risk to introduce faults in the code.

For instance, those test cases involved in requirements that have changed will be given a higher priority to be tested than those with a low volatility.

Srikanth et al. [133, 135], proposed a test case prioritization approach that involves analyzing and assigning values to each requirement using factors such as the requirements volatility. Requirements volatility is equal to the number of times a requirement has been changed in the development cycle normalized to a range of 0 to 10.

3.3.14 Fault proneness

In this objective, test cases are ordered for execution according to the fault proneness of the requirements associated to them.

This objective uses the requirements of the development team to identify the requirements that have had customer-reported failures in the previous software release. As the system evolves into several versions, the developers can use the data collected from prior versions to identify requirements that are likely to be error prone.

In [135], the authors present an approach that uses the information about fault proneness of the system requirements to prioritize its test cases.

3.3.15 Memory consumption or footprint or download statistics

Test cases will be executed according to a non-functional property such as the memory consumption, the footprint or the number of downloads of a system.

This objective applies reliable estimates of measurable non-functional properties to be used as weights to the configurations to be tested. These weights indicate the order of execution.

Herrejon et al. [82] propose a priority assignment scheme based on criteria as main memory consumption or footprint of a system. The measured values are used to order the products to be tested in a software product line. Similarly, Johansen et al. [68] propose combinatorial testing with weights as a way to select important feature interactions to cover them first. In particular, they use a real software project with its download statistics as weights.

3.3.16 Cost

This objective aims minimizing the cost associated to the tests. When cost is a relevant factor, this type of objective and the approaches implementing it are an effective solution. The cost of testing is related to the resources required to configure, execute and validate the test cases: time, human effort, cost of hardware, equipment, etc. Thus, test cases involving lower cost are running first.

In [86], the authors present a prioritization proposal based on the cost of testing. They define the cost of a test case as the total time required to run the test case and validate its output by comparison with the expected output. In [62], the authors propose to order test cases according to the cost of testing in terms of the run-time obtained from the latest regressing testing performed. Similarity, Epitropakis et al. [44] propose a test case prioritization technique using the execution cost criterion. They measure the execution cost with a tool that provides a precise measurement of the computational effort required to execute each test case.

Ferrer et al. [48] present a prioritization based on risk models that take into account error costs. Components with a high risk have higher weights than components with a low risk to be tested. Hema et al [134] also proposed a prioritization problem aiming to maximize fault detection while minimizing the install and build cost within a given time budget.

In [146], the authors consider multiple cost measures to prioritize the test cases. In particular, their work try to minimize the execution cost defined as execution time of test cases and cost of setting up correct resources for executing an specific test case.

3.4 Multi-objective test case prioritization

Most of the existing research works on test case prioritization methods are based on single prioritization objective. This neglects the potential benefits of combining multiple objectives to guide the detection of faults. Test case prioritization approaches that use several objectives simultaneously for ordering test cases are called multi-objective test case prioritization. Thus, the multi-objective perspective enables software developers to analyze the trade-offs between their objectives (sometimes conflicting) such as the case that, for instance, test cases are prioritized reducing the cost of testing while trying to accelerate the detection of faults.

Within the multi-objective test case prioritization techniques, we can found two types of approaches: those that combine several objectives into a single function by assigning them weights proportional to their relevance (e.g. fitness function), and others that use separate objective functions (equally relevance) with pure multi-objective optimization algorithms that results in many different solutions. We present some instances of these works in Table §3.3 as we will explain later.

Since the automation of test case prioritization process is limited by constrained such as the size and complexity of software, and the basic fact that in general, test data optimization is an undecidable problem, the use of search based optimization techniques has been grown in recent years [2, 90]. The term of search based software engineering involves the application of search techniques to software engineering problems. Optimizing test case prioritization seeks to order test cases so that test goals are achieved earlier in the sequence of test case application. Software engineering problems are typically multi-objective problems. The objectives that have to be met are often competing and somewhat contradictory. For example, in project planning, seeking earliest completion time at the cheapest overall cost will lead to some conflict of objectives. However, there does not necessarily exist a simple tradeoff between the two, making it desirable to find solutions that optimize both [2, 78, 90].

Most common examples of search based algorithms are the well-known hill climbing algorithm (an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution) and the genetic algorithms (algorithms that represent a class of adaptive search techniques based on the processes of natural genetic selection according to Darwinian theory of biological evolution) [78].

Table §3.3 shows some examples of test case multi-objective prioritization approaches, the combined prioritization objectives and the testing techniques used. We also extend the description of these proposals below.

Objectives	**Techniques**	**Approach**
Code coverage + Cost	Genetic algorithm	[145]
Code coverage + Cost + Capability to detect faults + Number of tests	Search algorithms	[146]
Code coverage metrics	Genetic algorithm	[4]
Code coverage metrics + Fault history	Evolutionary algorithms	[44]
Code coverage + Customer's requirements + Cost	Genetic algorithm	[63]

Table 3.3: Examples of multi-objective test case prioritization approaches

Walcoot et al. [145] presented a prioritization technique that uses a genetic algorithm to prioritize a regression test suite considering both testing time and the percentage of code coverage. In [146], it is proposed a multi-objective test case prioritization including minimization of execution cost, maximization of number of prioritized test cases, feature pairwise coverage and fault detection capability. They use three search algorithms (i.e. (1+1) evolutionary algorithm, alternating variable method and random search). In [4], the authors proposed a test case prioritization strategy using a genetic algorithm with multi-criteria fitness function. The fitness function is a weighted sum of the maximum coverage of the test case for conditions, multiple condition and statements in each test case.

The authors in [44] study a three-objective formulation of test case prioritization considering statement coverage, the difference of the statement coverage between two consecutive versions and fault history coverage. Also, they take into account the execution cost of each test case. Their study includes new implementations of different multi-objective evolutionary algorithms, the non-dominated sorting genetic algorithm II (NSGA-II) and the two archive evolutionary algorithm (TAEA). The work in [63] proposes a multi-objective technique to prioritize test cases using 3 objectives: code coverage, requirement relevance and execution cost. They applied the NSGA-II genetic algorithm.

3.5 Evaluation metrics

Various metrics have been proposed to evaluate the effectiveness of the test case prioritization approaches. The most commonly used metric is the well-known Average Percentage of Faults Detected (APFD) [19].

The metric APFD evaluates how quick faults are detected by a test suite [39, 111, 134]. APFD 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:

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

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. The closest the value is to 1, the fastest is the suite at detecting faults. For example, consider a test suite of 4 test cases (TC1-TC4) and 5 faults (F1-F5) detected by those test cases, as shown in Table §3.4. Consider two orderings of these test cases, ordering O1: TC1, TC2, TC3, TC4 and ordering O2: TC3, TC2, TC4, TC1. According to the previous APFD equation, ordering O1 produces an APFD of 58%: $1 - \frac{1+1+2+3+4}{4\times5} + \frac{1}{2\times4} = 0.58$ and ordering O2 an APFD of 78%: $1 - \frac{1+1+1+1+3}{4\times5} + \frac{1}{2\times4} = 0.78$, being O2 much faster detecting faults than O1.

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

Table 3.4: Test suite and faults exposed

The limitation of using APFD metric is that this metric assumes that all faults have equal severity and test cases have equal costs which does not meet the practice. Therefore, an improved metric called APFDc [36] was proposed for taking into account the fault severity and test cost. The value of APFDc can be computed according to the following Equation:

$$APFD_C = \frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i}$$

where n is the number of test cases and t_i represents the cost of test case i, m is the number of revealed faults and f_i represents the severity of fault i. TF_i is the position of the first test case which reveals the fault i in the ordered test cases sequence. This formula can also be used to compute the APFD value if both fault severity and test case costs are identical.

There are other metrics in the literature to evaluate the prioritization such as ASFD (Average Severity of Faults Detected) [132], TPFD (Total Percentage of Faults Detected) [72], NAPFD (Normalized APFD) [104] y CE (Coverage Effectiveness) [70].

3.6 Summary

In this chapter we have presented and classified the objectives and main proposals for test case prioritization. Furthermore, we have shown some of the works proposed in the field of multi-objective test case prioritization as well as the most used techniques in this field. The order in which test cases are executed is more relevant as the complexity and the size of software applications increase. Choosing the right order allows us to achieve our goals before, minimizing the effort of testing and debugging.

TESTING HIGHLY-CONFIGURABLE SYSTEMS

Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger Wybe Dijkstra, computer scientist, 1972 Turing Award,

Testing highly-configurable software systems is a difficult task due to the high number of possible combinations of features which often lead to a combinatorial explosion of possible configurations that need to be tested. This makes exhaustive testing of HCSs unfeasible. However, different testing approaches have been proposed to lead with this problem. In this chapter, we introduce the testing on HCSs and present the body of literature on this research area. Section §4.1 describes the general idea of testing HCSs. The most applied techniques to test HCSs are presented in Section §4.2 and Section §4.3. Finally, we summarize the chapter in Section §4.4.

4.1 INTRODUCTION

As the development of highly-configurable software systems is becoming a very common practice, there is an increasing need of efficient HCSs testing techniques. Companies such as Toshiba, Nokia and Boeing have been reporting success cases for the adoption of HCSs, in the form of SPLs, into their respective practice [40]. Thus, in recent years, there has been a growing interest by the academic and industrial communities to propose and evaluate new methods and tools to test the HCSs [41, 83, 146]. However, the testing of these systems is a complex and costly task since the variety of configurations derived from a highly-configurable software system is huge. As an example, Debian Wheezy, a well-known Linux distribution, has more than 37,000 packages that can be combined (with restrictions) to form millions of different configurations [1]. This makes exhaustive testing of HCSs unfeasible, 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.

In this context, there have been many attempts to reduce the space of testing through test selection techniques. Test case selection approaches choose a subset of test cases according to some coverage criteria [30, 58, 88]. These techniques have taken a step forward to make HCS testing affordable. However, the number of test cases derived from selection could still be high and expensive to run. This may be especially costly during regression testing when tests must be repeatedly executed after any relevant change is made to the HCS. In this context, the order in which configurations are tested is commonly assumed to be irrelevant. As a result, it could be the case that the most promising test cases (e.g. those detecting more faults) are run in last place forcing the tester to wait for hours or even days before starting the correction of faults. In a worse scenario, testing resources could be exhausted before running the whole test suite remaining faults undetected.

To take advance of the order in which test cases are executed arise the test case prioritization techniques. 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 [5, 82, 146]. Both, test selection and test prioritization strategies are complementary and are often combined.

In this chapter, we introduce the challenges found on the area of HCS testing and the solutions proposed reviewing the most salient works in this research area.

4.2 Test case selection approaches for HCSs

As mentioned previously, test case selection is one of the main strategies adopted for testing highly-configurable software systems. Test case selection reduces the test space by selecting an effective and manageable subset of configurations to be tested [30, 58, 74, 85, 88]. The goal is to achieve a high coverage of feature interactions with as small a number of configurations as possible. Figure §4.1 graphically depicts the test case selection process for HCSs. The process starts with all the test cases of an HCS. Then, the test cases are selected based on a given coverage criterion, resulting in a smaller but representative subset of test cases to be tested.

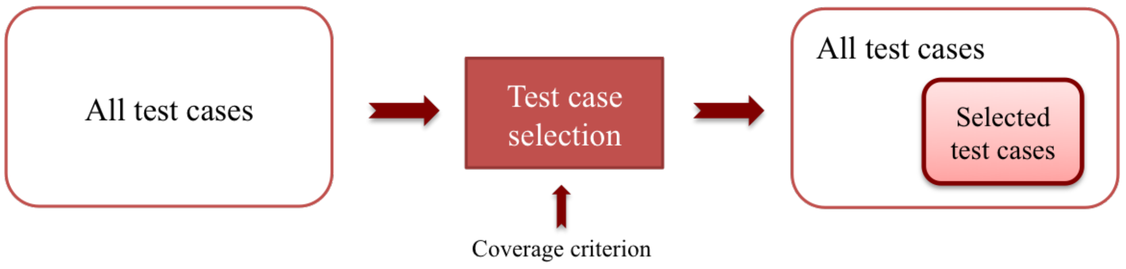


Figure 4.1: Test case selection

Within this context, there has been a stark and recent interest in the area of software product lines testing as evidenced by several systematic studies [26, 32, 41, 74, 75]. Among the finding of these studies, Combinatorial Interaction Testing (CIT) was identified as the leading selection approach for testing. Other authors have proposed using grammar-based and search-based techniques to reduce the number of test cases while maintaining a high fault detection capability. We introduce these works in the following subsections.

4.2.1 Combinatorial interaction testing

Combinatorial interaction testing (CIT) is a recognized software testing technique introduced by Cohen [21]. The effectiveness of CIT is based on the observation that most of the defects are expected to be caused by an interaction of few features. In these approaches test cases are selected in a way that guarantee that all combinations of t features are tested (i.e. t-wise). Roberto et al. conducted a systematic mapping study to delve into more detail about this subject [80]. This study identified over forty different approaches that rely on diverse techniques, such as genetic and greedy domains of different characteristics. This study also revealed that the large majority of approaches focus only on computing the samples of configurations based purely on variability models (e.g. feature models).

Additionally, most of the approaches found focus on pairwise testing (i.e. 2-wise), this is, test cases must cover all possible combinations of pairs of features based on the observation that most faults originate from a single feature or by the interaction of two features [101]. For instance, Table §4.1 shows the set of configurations obtained when applying pairwise testing to the model in Figure §2.3. The test suite is reduced from 18 (total number of configurations of the feature model) to 7 in the pairwise suite. Pérez et al. [73] define a strategy for testing SPL products using pairwise as coverage criteria, in the sense that all the pairs of features must be included and tested in at least one product.

ID	Test case
TC1	E-Shop,Catalogue,Payment,Security,Bank Transfer,High
TC2	E-Shop,Catalogue,Payment,Security,Credit Card,Standard,Search,Public Report
TC3	E-Shop,Catalogue,Payment,Security,Bank Transfer,Standard
TC4	E-Shop,Catalogue,Payment,Security,Credit Card,High,Search
TC5	E-Shop,Catalogue,Payment,Security,Bank Transfer,High,Search,Public Report
TC6	E-Shop,Catalogue,Payment,Security,Bank Transfer,Credit Card,Standard
TC7	E-Shop,Catalogue,Payment,Security,Credit Card,Standard

Table 4.1: E-Shop pairwise test suite

Moreover, only few combinatorial techniques take into account higher coverage strengths (i.e. $t>3$). A recent 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 [59]. They used an evolutionary algorithm with a similarity based objective function (see Section §3.3.9) to generate samples of test cases. Marijan et al. [88] presented a framework for automated pairwise testing of SPL, with an objective to generate the minimal set of test configurations that are valid and cover all pairwise feature interactions. Perrouin proposes transforming the feature models into Alloy declarative programs, in order to select valid configurations, with respect to the initial model [100]. Johansen generates test configurations with 1-3-way coverage from large feature models using greedy algorithm to enforce all pairs in a set of configurations [65].

4.2.2 Other techniques

Other authors have proposed using grammar-based and search-based techniques to reduce the number of test cases while maintaining a high fault detection capability [9, 35, 43]. Bagheri et al. attempted to reduce the number of required tests for testing a software product line using eight coverage criteria based on the transformation of feature models into formal context-free grammars. Closest to the previous work, Cohen et al. [35] proposed to map OVM product line representation models onto a relational model for defining the cumulative coverage criteria based on whose combination with combinatorial interaction testing methods suitable tests are generated. In [43], the authors presented a search-based testing approach based on the evolutionary Genetic Algorithms to automatically generate test suites for software product lines. They exploited the exploration capabilities of Genetic Algorithms to search the configuration space of a feature model and to find the subset that provides suitable error and feature coverage.

Later, Henard et al. [58] presented a genetic algorithm to handle multiple conflicting objectives in test optimization for SPLs. They define three optimization objectives: maximizing the pairwise coverage, minimizing the number of products selected and minimizing the overall test suite cost. However they assign weights to these objectives and this may make impossible reaching some parts of the Pareto front when dealing with convex fronts. Sayyad et al. [120] demonstrated that search-based optimization can be used to find products that optimize multiple objectives simultaneously, in particular, five objectives in their experiments. Roberto et al. proposed an approach for computing the exact Pareto front for pairwise coverage of two objective functions, maximization of coverage and minimization of test suite size [79]. Another really recent work is the proposed by Henard et al. [60] combining a multi-objective search-based SPL feature selection algorithm and constraint solving.

4.3 Test case prioritization approaches for HCSs

Test case prioritization identifies the efficient ordering of test cases to accelerate the detection of faults as we explained in Chapter §3. There are very few works about prioritization in HCS testing. Most of them are based on weighted combinatorial interaction testing, that is, generating combinatorial configurations using weights to prioritize them. For instance, Johansen et al. attached arbitrary weights to configurations to reflect market relevance and compute the covering arrays using a greedy approach [68]. This approach was formalized by Herrejon et al. who also proposed a parallel genetic algorithm that achieves better performance in a larger number of case studies [82]. Hernard et at. [59] presented a prioritization approach based on similarities to generate software product line test cases while maximizing the t-wise coverage. All these approaches combine sampling and prioritization during the test case generation.

Some other works do not use combinatorial interaction testing for sampling their test cases. Devroey et al. proposed a model-based testing approach to prioritize SPL testing [29]. Their approach relies on a feature model, a feature transition system and a usage model with the probabilities of executing relevant transitions. This approach combines concepts stemming from statistical testing with SPL sampling to extract products of interest according to the probability of their execution traces. In [42], the authors proposed a prioritization method for product line test cases based on the preferences of the domain stakeholders' goals. Al-Hajjaji et al. [5] proposed a similarity-based prioritization to be applied on the products of an SPL before they are generated. All these test case prioritization approaches do not depend on specific sampling strategies but they are independent and can be combined with any sampling algorithm.

All the previous approaches employ single objective algorithms to prioritize the test cases of their HCSs. To the best of our knowledge, only one study have been conducted on employing multi-objective prioritization of HCS testing. Wang et al. [146], propose a search-based multi-objective test prioritization technique including the minimization of execution cost (i.e. cost of allocate test resources), and the maximization of number of prioritized test cases, feature pairwise coverage and fault detection capability. However, they combine the objectives into a single function by assigning them weights proportional to their relative importance. Among the shortcomings is 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.

4.4 Summary

In this chapter, we have discussed the challenges and the main approaches for testing highly-configurable software systems. Two main strategies have been studied in this research area: test case selection and test case prioritization. However, very few works have been proposed on test case prioritization of HCSs to date.

Part III

Contributions

Functional test case prioritization criteria for HCSs

If debugging is the process of removing bugs,
then programming must be the process of putting them in.

Edsger Dijkstra, computer scientist,

As mentioned in previous chapters, HCS testing is a challenging task. In that sense, numerous contributions have been proposed to reduce the number of configurations to be tested, however, not much attention has been paid to the order of execution of tests. In this chapter, we explore the applicability of test case prioritization techniques to HCS testing. We propose five different prioritization criteria based on common metrics of feature models and we compare their effectiveness in increasing the rate of early fault detection, i.e. a measure of how quickly faults are detected. The results show that different orderings of the same HCS suite may lead to significant differences in the rate of early fault detection. They also show that our approach may contribute to accelerate the detection of faults of HCS test suites based on combinatorial testing. In Section §5.1, we describe the context of testing of this proposal. Section §5.2 presents some background about the analysis of feature models. In Section §5.3 we propose five functional prioritization criteria for HCSs. The evaluation of our approach is described in Section §5.4. Section §5.5 presents the threats to validity of our work. Finally, we summarize our approach and conclusions in Section §5.6.

5.1 Introduction

HCSs are often represented through feature models (see Chapter §2). The automated analysis of feature models deals with the computer-aided extraction of information from feature models. These analyses allow studying properties of the HCS such as consistency, variability degree, complexity, etc. In the last two decades, many operations, techniques and tools for the analysis of feature models have been presented [14].

In this chapter, HCS testing consists in deriving a set of configurations from an HCS and testing each configuration. An *HCS test case* can be defined as a configuration of the HCS to be tested, i.e. a set of features. The high number of feature combinations in HCSs may lead to thousands or even millions of different configurations, e.g. the e-shop model available in the SPLOT repository has 290 features and represents more than 1 billion of configurations [92]. This makes exhaustive testing of an HCS infeasible. In this context, there have been many attempts to reduce the space of testing through feature-based test selection [73, 85, 98, 100] as we described in Chapter §4. Test selection techniques have taken a step forward to make HCS testing affordable. However, the number of test cases derived from selection could still be high and expensive to run. *Test case prioritization techniques* schedule test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goal [19, 78, 110, 111]. Test case prioritization techniques have been extensively studied as a complement for test case selection techniques in demanding testing scenarios [42, 104].

In this chapter, we present a test case prioritization approach for HCSs. In particular, we explore the applicability of scheduling the execution order of HCS test cases as a way to reduce the effort of testing and to improve their effectiveness. To show the feasibility of our approach, we propose five different prioritization criteria intended to maximize the rate of early fault detection of the HCS suite, i.e. detect faults as fast as possible. Three of these criteria are based on the complexity of the configurations. Hence, more complex configurations are assumed to be more error-prone and therefore are given higher priority over less complex ones, i.e. they are tested first. Another prioritization criterion is based on the degree of reusability of configurations features. In this case, configurations including the more reused features are given priority during tests. This enables the early detection of high-risk faults that affect to a high portion of the configurations. Finally, we propose another criterion based on the so-called dissimilarity among configurations, i.e. a measure of how different two configurations are. This criterion is based on the assumption that the more different two configurations have higher feature coverage and fault detection rate. The proposed prioritization criteria are based on common metrics of feature models extensively studied in the literature. This allowed us to leverage the knowledge and tools for the analysis of feature models making our approach fully automated. Also, this makes our prioritization criteria complementary to the numerous approaches for feature-based test case selection.

For the evaluation of our approach, we developed a prototype implementation of the five prioritization criteria using the SPLAR tool [91]. We selected a number of realistic and randomly generated feature models and generated both random and pairwise-based test suites. Then, we used our fault generator based on the work of Bagheri et al. [9, 43] to seed the features with faults. Finally, we reordered the suite according to the five criteria and we measured how fast the faults were detected by each ordering. The results show that different orderings of the same HCS suite may lead to significant differences in the rate of fault detection. More importantly, the proposed criteria accelerated the detection of faults of both random and pairwise-based HCS test suites in all cases. These results support the applicability and potential benefits of test case prioritization techniques in the context of HCS. We trust that our work will be the first of a number of contributions studying new prioritization goals and criteria as well as new comparisons and evaluations.

5.2 Preliminaries

5.2.1 Automated analysis of feature models

Highly-Configurable Software Systems are often graphically represented using feature models (see Chapter §2). The analysis of feature models consists on examining their properties. This is performed in terms of analysis operations. Among others, these operations allow finding out whether a feature model is void (i.e. it represents no configurations) whether it contains errors (e.g. dead features) or what is the number of possible feature combinations in an HCS. Catalogues with up to 30 different analysis operations on feature models have been reported in the literature [14]. Some tools supporting the analysis of feature models are AHEAD Tool Suite [3], FaMa Framework [141] and SPLAR [91]. Next, we introduce some of the operations that will be mentioned throughout this chapter:

All configurations: This operation takes a feature model as input and returns all the configurations represented by the model. For the model in Figure §2.5 of Chapter §2, this operation returns the list of configurations shown in Table §5.1.

Commonality: This operation takes a feature model and a feature as inputs and returns the commonality of the feature in the HCS represented by the model. Commonality is a metric that indicates the reuse ratio of a feature in an HCS, this is, the percentage of

ID	Configurations
C1	E-Shop,Catalogue,Payment,Bank Transfer,Security,High
C2	E-Shop,Catalogue,Payment,Bank Transfer,Security,Standard
C3	E-Shop,Catalogue,Payment,Credit Card,Security,High
C4	E-Shop,Catalogue,Payment,Bank Transfer,Credit Card,Security,High
C5	E-Shop,Catalogue,Payment,Bank Transfer,Security,High,Search
C6	E-Shop,Catalogue,Payment,Bank Transfer,Security,Standard,Search
C7	E-Shop,Catalogue,Payment,Bank Transfer,Security,Standard,Search,Public Report
C8	E-Shop,Catalogue,Payment,Credit Card,Security,High,Search
C9	E-Shop,Catalogue,Payment,Credit Card,Bank Transfer,Security,High,Search

Table 5.1: E-Shop configurations

configurations that include the feature. This operation is calculated as follows:

$$Comm(f,fm) = \frac{filter(fm,f)}{\#configurations(fm)} \tag{5.1}$$

$\#configurations(fm)$ returns the number of configurations of an input feature model, *fm*, and $filter(fm,f)$ returns the number of configurations in *fm* that contain the feature f. The result of this operation is in the domain [0,1]. As an example, consider the model in Figure §2.5 and the feature *Credit Card*. The commonality of this feature is calculated as follows:

$Comm(f,fm) = \frac{filter(fm,Credit\,Card)}{\#configurations(fm)} = \frac{4}{9} = 0.45$

The feature *Credit Card* is therefore included in 45% of the configurations. A more generic definition of this operation for software product lines is presented in [14].

Cross-Tree-Constraints Ratio (CTCR): This operation takes a feature model as input and returns the ratio of the number of features in the cross-tree constraints (repeated features counted once) to the total number of features in the model [8, 14, 93]. This metric is usually expressed as a percentage value. This operation is calculated as follows:

$$CTCR(fm) = \frac{\#constraintsfeatures(fm)}{\#features(fm)} \tag{5.2}$$

$\#constraintsfeatures(fm)$ is the number of features involved in the cross-tree constraints and $\#features(fm)$ is the total number of features of the model fm. The result of this operation is in the domain [0,1]. For instance, the CTCR of the model in Figure §2.5 is $3/10 = 0.3$ (30%).

Coefficient of Connectivity-Density (CoC): In graph theory, this metric represents how well the graph elements are connected. Bagheri et al. [8] defined the CoC of a feature model as the ratio of the number of edges (any connection between two features, including constraints) over the number of features in a feature model. This is calculated as follows:

$$CoC(fm) = \frac{\#edges(fm)}{\#features(fm)} \tag{5.3}$$

$\#edges(fm)$ denotes the number of parent-child connections plus the number of cross-tree constraints of an input model *fm* and $\#features(fm)$ is the number of total features in the model fm. For instance, the model in Figure §2.5 has 11 edges (9 parent-child connections plus 2 constraints) and 10 features, i.e. $CoC(fm) = 11/10 = 1.1$.

Cyclomatic Complexity (CC): The cyclomatic complexity of a feature model can be described as the number of distinct cycles that can be found in the model [8]. Since a feature model is a tree, cycles can only be created by cross-tree constraints. Hence, the cyclomatic complexity of a feature model is equal to the number of cross-tree constraints of the model. In Figure §2.5, $cc(fm) = 2$.

Variability Coverage (VC): The variability coverage of a feature model is the number of variation points of the model [43]. A variation point is any feature that provides different variants to create a configuration. Thus, the variation points of a feature model are the optional features plus all non-leaf features with one or more non-mandatory subfeatures. In Figure §2.5, $vc(fm) = 5$ because features *E-Shop*, *Payment*, *Security*, *Search* and *Public Report* are variation points.

5.3 Test case prioritization criteria for highly-configurable software systems

In this section, we propose the application of test case prioritization techniques in the context of Highly-Configurable Software System. As a part of the proposal, we define and compare five test case prioritization criteria to maximize the rate of early fault detection of a test suite. This goal aims to achieve a sequence of test cases to be run in a way that faults are detected as soon as possible. This enables faster feedback about the system under test and lets developers begin correcting faults earlier. Hence, it could provide faster evidence that quality objectives were not met and the assurance that those tests with greatest fault detection ability will have been executed if testing is halted [37].

Figure §5.1 depicts a rough overview of the general HCSs testing process and how our prioritization approach fits on it. First, the variability model (usually a feature model) is inspected and the set of configurations to be tested is selected. The selection could be done either manually (e.g. selecting the configuration portfolio of the company) or automatically (e.g. using t-wise). Once the suite is selected, specific test cases should be designed for each configuration under test. Then, the set of configurations to be tested could be prioritized according to multiple criteria determining the execution order of the test cases. Some of the criteria may need analyzing the feature model or using feedback from previous test executions during regression testing. A point to remark is that prioritization does not require creating new test cases, just reordering the existing ones. As a result, prioritization could be re-executed as many times as needed with different criteria. For instance, during the initial stages of development configurations could be reordered to maximize feature coverage and during regression testing they could be reordered to detect critical faults as soon as possible, e.g. those causing failures in a higher number of configurations. The prioritization criteria proposed are presented in the next sections.

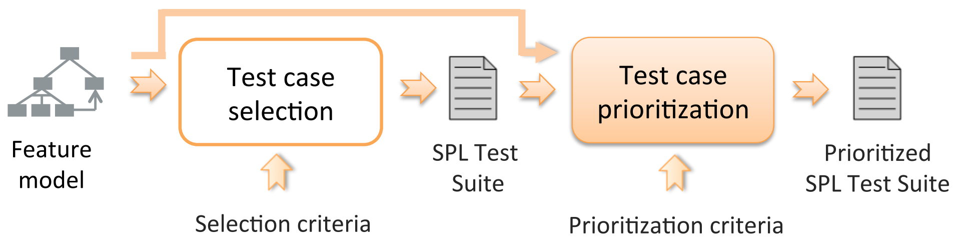


Figure 5.1: Overview of the highly-configurable software system testing process

5.3.1 CTCR prioritization criterion

This criterion is based on the Cross-Tree-Constraints Ratio (CTCR) defined in Section §5.2.1. The CTCR metric has been used to calculate the complexity of feature models and it is correlated with the possibility and ease of change in a model when modifications are necessary [8]. This metric inspired us to define the CTCR prioritization criterion as a way to identify the more complex configurations in terms of the degree of involvement in the constraints of their features. We hypothesize that this criterion can reduce testing effort while retaining a good fault detection rate by testing earlier the more complex configurations in terms of constraints.

Given a configuration c and a feature model fm, we define the CTCR criterion as follows:

$$CTCR(c, fm) = \frac{\#constraintsfeatures(c, fm)}{\#features(c)} \tag{5.4}$$

$\#constraintsfeatures(c, fm)$ denotes the number of distinct features in c involved in constraints and $\#features(c)$ is the number of total features in configuration c. This formula returns a value that indicates the complexity of configuration c in terms of features involved in constraints.

As an example, the CTCR prioritization value of the configurations C4 and C6 presented in Section §5.2.1 is calculated as follows:

$CTCR(C4, fm) = 2/7 = 0.29$

$CTCR(C6, fm) = 0$

In C4, the *Credit Card* and *High* Security features share an *include* constraint and also *High* Security feature has an exclude constraint with *Public Report*. However, the features in C6 do not involve any constraints. Thus, configuration C4 will be tested earlier than configuration C6 according to the CTCR values i.e., $CTCR(C4, fm) > CTCR(C6, fm)$.

5.3.2 CoC prioritization criterion

The Coefficient of Connectivity-Density (CoC) metric, presented in Section §5.2.1, was proposed to calculate the complexity of a feature model in terms of the number of edges and constraints of the model [8]. We propose to adapt this metric for HCS configurations and use it as a test case prioritization criterion. The goal is to measure the complexity of configurations according to their CoC and give higher priority to those ones with higher complexity.

Given a configuration c and a feature model fm, we define the CoC of a configuration as shown below:

$$CoC(c, fm) = \frac{\#edges(c, fm)}{\#features(c)} \tag{5.5}$$

$\#edges(c, fm)$ denotes the number of edges (parent-child connections plus cross-tree constraints) among the features in configuration c. This formula returns a value that indicates the complexity of c based on the CoC metric.

As an example, the *CoC* value of the configurations C7 and C9 presented in Section §5.2.1 is calculated as follows:

$CoC(C7, fm) = 8/8 = 1$

$CoC(C9, fm) = 9/8 = 1.13$

In C7, the *E-Shop* feature is connected with edges to four features (*Catalogue*, *Payment*, *Security* and *Search*). Also, *Payment* is connected to the *Bank Transfer* feature, *Security* to the *Standard* Security feature, *Search* to *Public Report* and *Public Report* has an exclude constraint with *High* Security. Note that the exclude constraint is considered because it is being fulfilled by this configuration since it includes *Public Report* feature and not *High* Security feature. Thus, C9 has higher priority than C7 and therefore it would be tested first.

5.3.3 VC&CC prioritization criterion

In [43], the authors presented a genetic algorithm for the generation of SPL products with an acceptable tradeoff between fault coverage and feature coverage. As part of their algorithm, they proposed a fitness function to measure the ability of a product to exercise features and reveal faults, i.e. the higher the value of the function, the better the product. This function is presented below:

$$VC\&CC(p, fm) = \sqrt{vc(p, fm)^2 + cc(p, fm)^2} \tag{5.6}$$

$vc(p, fm)$ calculates the variability coverage of a product p of the model fm and $cc(p, fm)$ represents the cyclomatic complexity of p (c.f. Section §5.2.1).

Since this function has been successfully applied to SPL test case selection, we propose to explore its applicability for test case prioritization in a more general perspective, HCSs. According to this criterion, those configurations in a HCS with higher values for the function are assumed to be more effective in revealing faults and will be tested first.

As an example, the VC&CC value of the configurations C3 and C6 presented in Section §5.2.1 is calculated as follows:

$$VC\&CC(C3, fm) = \sqrt{3^2 + 2^2} = \sqrt{9 + 4} = 3.6$$

$$VC\&CC(C6, fm) = \sqrt{4^2 + 0^2} = \sqrt{16 + 0} = 4$$

In configuration C3, *E-Shop*, *Payment* and *Security* features are variation points. Also, C3 presents a require constraint with *Credit Card* and *High* Security features and an exclude constraint between *High* Security and *Public Report*. According to this criterion, configuration C6 would be tested earlier than configuration C3, $VC\&CC(C6) > VC\&CC(C3)$.

5.3.4 Commonality prioritization criterion

We define a commonality-based prioritization criterion that calculates the degree of reusability of configurations features. That is, the features that have higher commonality and the configurations that contain them will be given priority to be tested. This enables the early detection of faults in highly reused features that affect to a high portion of the configurations providing faster feedback and letting software engineers begin correcting critical faults earlier.

Given a configuration c and a feature model fm, we define the Commonality criterion as follow:

$$Comm(c, fm) = \sum_{i=1}^{\#features(c)} (Comm(fi))/\#features(c) \tag{5.7}$$

fi denotes a feature of configuration c. The range of this measure is [0,1]. Roughly speaking, the priority of a configuration is calculated by summing up the commonality of its features. The sum is then normalized according to the number of configuration features.

As an example, the Commonality values of the configuration C1 and C2 presented in Section §5.2.1 are calculated as follows:

$Comm(C1, fm) = (Comm(EShop) + Comm(Catalogue) + Comm(Payment)$
$+Comm(Bank\ Transfer) + Comm(High) + Comm(Security))/6 =$
$((9+9+9+7+9+6)/9)/6 = 0.91$

$Comm(C2, fm) = ((9+9+9+7+9+3)/9)/6 = 0.85$

Based on these results, C1 would appear before than C2 in the prioritized list of configurations and would be tested first.

5.3.5 Dissimilarity prioritization criterion

A (dis)similarity measure is used for comparing similarity (diversity) between a pair of test cases. Hemmati et al. [57] and Henard et al. [59] investigated ways to select an affordable subset with maximum fault detection rate by maximizing diversity among test cases using the dissimilarity measure. The results obtained in those papers suggested 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.

In this context, we propose to prioritize the test cases based on this dissimilarity metric, testing the most different configurations first, assuring a higher feature coverage and a higher fault detection rate. In order to measure the diversity between two configurations, we use the Jaccard distance that compare similarity of sample sets [137]. Jaccard distance is defined as the size of the intersection divided by the size of the union of the sample sets. In our context, each set represents a configuration containing a set of features. Thus, we choose first the two more dissimilar configurations (i.e. the configurations with the highest distance between them) and we add them to a list. Then, we continue adding the configurations with the highest distance between them until all configurations have been added to the list. The resulting list of configurations represents the order of configurations to be tested.

Given two configurations c_a and c_b, we define the Dissimilarity formula as follows:

$$Dissimilarity(c_a, c_b) = 1 - \frac{|c_a \bigcap c_b|}{|c_a \bigcup c_b|} \tag{5.8}$$

c_a and c_b represent different set of features (i.e. configurations). The resulting distance varies between 0 and 1, where 0 denotes that the configurations c_a and c_b are the same and 1 indicates that c_a and c_b share no features.

The dissimilarity values of the configurations C1, C7 and C8 presented in Section

§5.2.1 are calculated as follows:

$$Dissimilarity(C1,C7) = 1 - 5/9 = 0.44$$

$$Dissimilarity(C7,C8) = 1 - 5/10 = 0.5$$

For example, regarding to the distance between C1 and C7, they have 5 features in common (*E-Shop, Catalogue, Payment, Bank Transfer and Security*) out of 9 total features (the previous five plus *High, Standard, Search, Public Report*). C7 and C8 present greater distance between them than C7 and C1. Thus, C7 and C8 would be tested earlier than C1.

5.4 Evaluation

In this section, we present two experiments to answer the following research questions:

RQ1: Is the order in which HCS configurations are tested relevant?
RQ2: Are the prioritization criteria presented in Section §5.3 effective at improving the rate of early fault detection of HCS test suites?
RQ3: Can our prioritization approach improve the rate of early fault detection of current test selection techniques based on combinatorial testing?

We begin by describing our experimental settings and then we explain the experimental results.

5.4.1 Experimental settings

In order to assess our approach, we developed a prototype implementation for each prioritization criterion. Our prototype takes an HCS test suite and a feature model as inputs and generates an ordered set of test cases according to the prioritization criterion selected. We used the SPLAR tool [91] for the analysis of feature models. All the performed experiments were implemented using Java 1.6. We ran our tests on a Linux CentOS release 6.3 machine equipped with an Intel Xeon X5560@2.8Ghz microprocessor and 4 GB of RAM memory.

Models

For our experiments we selected 7 feature models of various sizes from the SPLOT repository [92]. Also, we generated 8 random models with up to 500 features using the BeTTy online feature model generator [124]. Table §5.2 lists the characteristics of the models. For each model, the name, the number of features and configurations and the CTCR are presented.

Name	Features	Configurations	CTCR	Faults
Web portal	43	2120800	25%	4
Video player	71	$4,5 \cdot 10^{13}$	0%	4
Car selection	72	$3 \cdot 10^{8}$	31%	4
Model transf.	88	$1 \cdot 10^{12}$	0%	8
Fm test	168	$1,9 \cdot 10^{24}$	28%	16
Printers	172	$1,14 \cdot 10^{27}$	0%	16
Electronic shop	290	$4,52 \cdot 10^{49}$	11%	28
Random1	300	$7,65 \cdot 10^{39}$	8%	28
Random2	300	$1,65 \cdot 10^{32}$	5%	28
Random3	350	$7,41 \cdot 10^{37}$	10%	32
Random4	400	$3,06 \cdot 10^{44}$	10%	40
Random5	450	$3,80 \cdot 10^{54}$	0%	44
Random6	450	$1,03 \cdot 10^{48}$	5%	44
Random7	500	$4,97 \cdot 10^{36}$	5%	48
Random8	500	$2,21 \cdot 10^{58}$	5%	48

Table 5.2: Feature models used in our experiments

Fault generator

To measure the effectiveness of our proposal, we evaluated the ability of our test case prioritization criteria to detect faults in the HCS under test. For this purpose, we implemented a fault generator for feature models. This generator is based on the fault simulator presented by Bagueri et al. which has been used in several works to evaluate the fault detection rate of HCS test suites [9, 43]. Our fault generator simulates faults in n-tuples of features with $n \in [1,4]$ (where n is a natural integer giving the number of features present in the n-tuple). Faults in n-tuples simulate interaction faults which require the presence of a set of features to be revealed. Studies show that faults caused by the interaction of between 2 and 4 features are frequent in practice [25]. The number of faults seeded on each model is equal to $m/10$ being m the number of model features. This information is detailed in the last column of Table §5.2. Each type of fault was introduced in the same proportion, i.e. 25% in single features, 25% in 2-tuples of features, 25% in 3-tuples of features and 25% in 4-tuples of features. Hence, our generator receives a feature model as input and returns a random list of faulty feature sets as output. For instance, for the model in Figure §2.5 the faults seeded could be given as follows: {{High}{Credit Card,Search}} representing a fault in the feature *High* Security and another fault caused by the interaction between the *Credit Card* and *Search* features.

5.4.2 Experiment 1. Prioritizing HCS test suites

In order to answer *RQ1* and *RQ2*, we checked the impact on the rate of early fault detection of the prioritization criteria defined in Section §5.3. The experimental setup and the results are next reported.

Experimental setup. For each model presented in Table §5.2, we performed several steps. First, we used our fault generator to simulate faults in the HCSs obtaining as a result a list of faulty features sets. Then, we randomly generated a test suite using SPLAR. The suite was composed of between 100 and 500 configurations depending on the size of the model. This step simulates the manual tests selection of an HCS engineer who could choose the configurations to be tested following multiple criteria: cost, release plan, users requests, marketing strategy, etc. For each fault in the model, we made sure that there was at least one configuration in the suite detecting it, i.e. the suite detected 100% of the faults. Once generated, the suite was ordered according to the prioritization criteria defined in Section §5.3 resulting in six total test suites, one random suite and five prioritized suites. Finally, we measured how fast the faults were detected by each suite calculating their APFD values (see Section §3.5 of Chapter §3). For the random suite the APFD was calculated as the average of 10 random orderings to avoid the effects of chance.

Experimental results. Table §5.3 depicts the size of the test suites and the APFD values obtained by the random and the five prioritized suites for each model. The best value on each row is highlighted in boldface. Also, main values are shown in the final row. As illustrated, there are significant differences on the APFD average values ranging from the 74.9% of the Commonality-ordered suite to the 96.5% reached by the VC&CC-ordered suite. These differences are even more noticeable in individual cases. For the model "Random3", for instance, the difference between the random and VC&CC-ordered suite is 33.3 points, i.e. from 63.9% to 97.2%. The best average results were obtained by the VC&CC criterion with 96.5%, followed by CoC (90.6%), Dissimilarity (87.4%), CTC (84.5%), random criterion (77.4%) and Commonality (74.9%). Interestingly, the APFD values obtained by the random suites in the models of lower size were remarkably high, e.g. $APFD(Videoplayer) = 92.0\%$. We found that this was due to the low number of faults seeded and to the size of the models that made the faults easily detectable using just a few tests. In the eight largest models, however, the difference between the random suite APFDs (63.9%-77.5%) and the ones of the prioritized suites (91.5%-98.2%) was noticeable. This suggests that our approach is especially helpful in large test spaces. Finally, we may remark that all the proposed prioritization criteria except Commonality improved the main value obtained by the random ordering. In fact, for all models at least several of the prioritized suites improved the random APFD values.

Figure §5.2 shows the percentage of detected faults versus the fraction of the test suite used for the model "Random3". Roughly speaking, the graphs show how the APFD value evolves as the test suite is exercised. It is noteworthy that the VC&CC-ordered suite, for instance, detected all the faults (32) by using just 15% of the suite, i.e. 75 test cases out of 500 with the highest priority. Another example is the Dissimilarity-ordered suite that detected all the faults by using only the 45% of the suite. The random suite, however, required using 95% of the test cases to detect exactly the same faults. This behaviour was also observed in the rest of the models under study. In real scenarios, with a higher number of faults and time-consuming executions, this acceleration in the detection of faults could imply important saving in terms of debugging efforts.

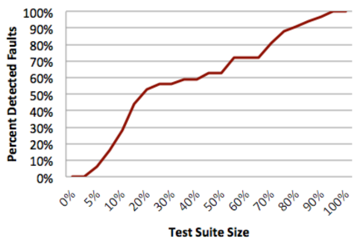
The results obtained answer positively to *RQ1* and *RQ2*. Regarding *RQ1*, the results show that the order in which tests are run is definitely relevant and can have a clear impact on the early fault detection rate of an HCS suite. Regarding *RQ2*, the results suggest that the presented ordering criteria, especially VC&CC, CoC and Dissimilarity could be effective at improving the rate of early fault detection of HCS test suites.

5.4.3 Experiment 2. Prioritization + combinatorial testing

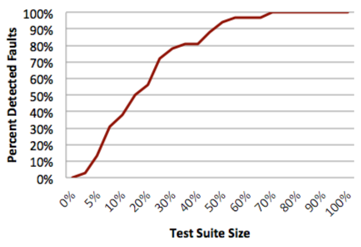
In order to answer *RQ3*, we checked whether our prioritization criteria could be used to increase the rate of early fault detection of test suites based on combinatorial selection. The experimental setup and results are next reported.

FM	**Suite size**	**APFD**					
		Random	**CoC**	**CTC**	**Comm**	**VC&CC**	**Diss**
Web p.	100	81.5	95.8	94.3	60.0	**99.0**	92.3
Car s.	100	83.6	96.5	94.8	92.5	**97.8**	90.5
Video p.	100	92.0	**98.8**	93.0	76.0	**98.8**	97.3
Model t.	100	83.3	94.8	75.5	79.9	**95.4**	91.5
Fm test	300	85.2	87.3	83.4	84.3	**94.3**	93.6
Printers	300	92.9	94.1	**98.4**	93.9	96.3	96.5
E. shop	300	90.2	93.4	92.1	87.6	96.3	**97.0**
Random1	300	64.7	92.6	84.2	60.1	**97.9**	89.1
Random2	300	73.1	87.9	78.6	77.5	**98.2**	80.4
Random3	500	63.9	79.6	70.7	80.7	**97.2**	85.2
Random4	500	69.7	93.9	92.2	53.8	**97.8**	65.8
Random5	500	77.5	**91.6**	78.4	73.7	91.5	89.1
Random6	500	69.5	83.5	73.6	43.8	**95.1**	88.8
Random7	500	66.3	90.4	81.6	72.6	**95.7**	87.0
Random8	500	67.3	79.3	76.3	86.3	**95.9**	67.0
Average		**77.4**	**90.6**	**84.5**	**74.9**	**96.5**	**87.4**

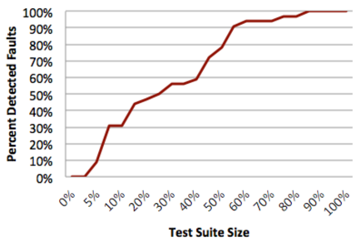
Table 5.3: APFD for random and prioritized suites



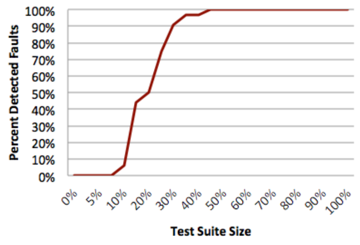
(a) Random APFD



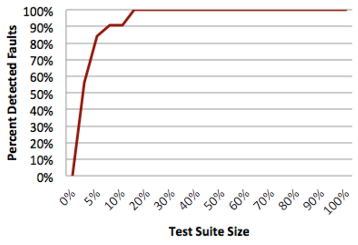
(b) CoC APFD



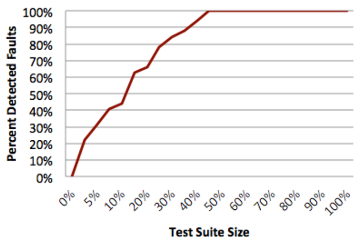
(c) CTC APFD



(d) Commonality APFD



(e) VC&CC APFD



(f) Dissimilarity APFD

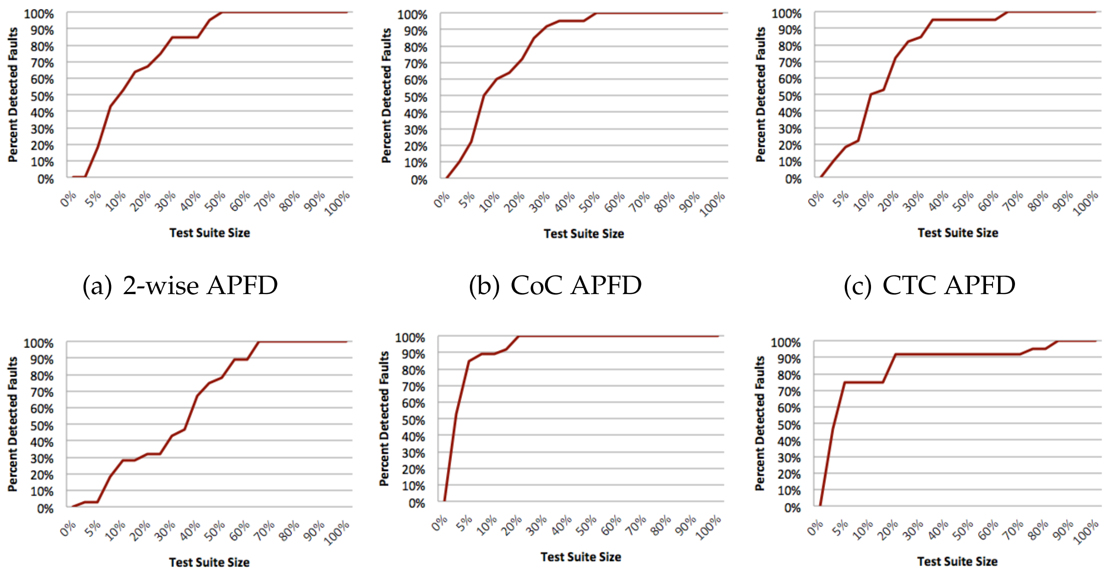
Figure 5.2: APFD metrics for Random3 model

Experimental setup. The experimental procedure was similar to the one used in Experiment 1. Feature models were seeded with the same faults used in our previous experiment. Then, for each model, we generated a 2-wise test suite using the SPLCAT tool presented by Johansen et al. [67]. As a result, we obtained a list of configurations covering all the possible pairs of features on each model. Then, we prioritized the list of configurations according to our five prioritization criteria and we calculated the APFD (see Section §3.5) of the resulting six suites, 2-wise and five prioritized suites. It is noteworthy that SPLCAT uses an implicit prioritization criterion placing first in the list those configurations that covers the most uncovered pairs of features. This tends to place those configurations with more features at the top of the list getting fast feature coverage. This approach therefore is likely to increase the fault detection rate and thus it is considered as an extra prioritization approach in our comparison.

Experimental results. The results of this experiment are presented in Table §5.4. For each model, the size of the pairwise test suite, the number of faults detected out of the total number of seeded faults and the APFD values of each ordering are presented. Note that the generated pairwise suites did not detect all the faults seeded on each model. As illustrated, the APFD average values ranged from 55.0% to 90.7%. As expected, the APFD average value of the pairwise suite (85.0%) was higher than the one of the random suite (77.4%) in Experiment 1. This was due to implicit prioritization criterion used by the SPLCAT tool that places at the top of the list those configurations containing a higher number of uncovered pairs of features, usually the largest configurations. As in the previous experiment, the best APFD average results were obtained by the VC&CC criterion with 90.7%, followed by CoC (88.0%) and Dissimilarity (86.9%). The pairwise suite got the fourth best APFD average value (85.0%). The CTC and Commonality prioritization criteria did not get to improve the results of the original suite. In terms of individual values, CoC got to improve the pairwise APFD values in 13 out of 15 models, VC&CC in 12 out of 15 models and Dissimilarity in 11 out of 15. As in our previous experiment, there was not a single model in which the pairwise suite obtained a higher APFD value than all the rest prioritized suites.

Figure §5.3 shows the percentage of detected faults versus the fraction of suite used for the feature model "Random3". In this example, it is remarkable that VC&CC-ordered suite detected all the faults with just 20% of the suite (i.e. 27 tests out of 135). Also, the CoC-ordered and the pairwise suites detected the same faults with only 50% of the suite. However, the CoC-ordered suite detected more faults earlier, i.e. the curve of CoC is slightly steeper than the 2-wise curve. A similar behaviour was observed in the rest of the models under study.

In response to *RQ3*, our results show that our prioritization criteria can be helpful to increase the rate of early fault detection of the current combinatorial testing techniques.



(a) 2-wise APFD (b) CoC APFD (c) CTC APFD

(d) Commonality APFD (e) VC&CC APFD (f) Dissimilarity APFD

Figure 5.3: APFD metrics for Random3 model

FM	**Suite size**	**Detected faults**	**APFD**					
			2wise	**CoC**	**CTC**	**Comm**	**VC&CC**	**Diss**
Web p.	19	3/4	90.3	93.9	90.4	46.5	**97.4**	**97.4**
Car s.	24	4/4	81.2	**90.6**	**90.6**	51.0	80.2	71.9
Video p.	18	4/4	76.4	**93.1**	76.4	22.2	83.3	83.3
Model t.	28	7/8	78.8	**88.0**	78.8	49.2	85.5	80.9
FM test	43	15/16	85.0	68.8	55.3	53.6	**93.9**	75.4
Printers	129	13/16	96.2	**97.1**	96.2	58.0	89.5	96.8
E shop	24	24/28	81.7	82.6	79.7	55.6	78.8	**85.2**
Random1	124	23/28	81.1	83.4	83.6	61.7	**96.6**	92.2
Random2	105	25/28	86.2	91.6	90.1	52.3	**94.8**	90.0
Random3	135	28/32	84.7	87.4	84.7	68.8	**97.1**	89.9
Random4	178	34/40	86.9	88.8	87.8	62.9	**95.7**	90.7
Random5	126	38/44	86.9	**94.9**	86.9	61.3	94.6	80.6
Random6	157	39/44	85.1	86.1	82.0	48.7	**92.4**	88.3
Random7	253	37/48	84.9	84.2	79.2	63.1	85.8	**91.8**
Random8	216	40/48	89.2	90.4	86.1	70.8	**95.1**	88.3
Average			**85.0**	**88.0**	**83.2**	**55.0**	**90.7**	**86.9**

Table 5.4: APDF for 2-wise and prioritized suites

5.5 Threats to Validity

In this section we discuss some of the potential threats to the validity of our studies. In order to avoid any bias in the implementation and make our work reproducible, we used a number of validated and publicly available tools. In particular, we used SPLAR [91] for the analysis of feature models, BeTTy [124] for the generation of random feature models, SPLCAT [67] for pairwise test selection and also we implemented a fault generator based on the work of Bagheri et al. [43]. Due to the lack of real HCSs with available test cases, we evaluated our approach by simulating faults in a number of HCSs represented by published and randomly generated models of different sizes. This may be a threat to our conclusions. However, we may remark that the evaluation of testing approaches using feature models is extensively used in the literature [59, 65, 67]. The use of a fault generator also implies several threats. The type, number and distribution of generated faults could not be the one found in real code. We may emphasize, however, that our generator is based on the fault simulator presented by Bagheri et al. [43] which has been validated in the evaluation of several SPL test case selection approaches [9, 43]. Furthermore, we remark that the characteristics and distribution of faults have a limited impact in our work since we are not interested in how many faults are detected but how fast they are revealed by different orderings of the same test suite.

5.6 Summary

In this chapter, we have presented a test case prioritization approach for HCSs. In particular, we have proposed five prioritization criteria to schedule test execution in an order that attempt to accelerate the detection of faults providing faster feedback and reducing debugging efforts. These prioritization criteria are based on standard techniques and metrics for the analysis of feature models and therefore are fully automated. The evaluation results showed that there are significant differences in the rate of early fault detection provided by different prioritization criteria. Also, the results showed that some of the criteria proposed may contribute to accelerate the detection of faults of both random and pairwise-based HCS test suites. This suggests that our work could be a nice complement for current techniques for test case selection. To the best of our knowledge, this approach has been the first considering not only *which* HCS configurations should be tested but *how* they should be tested. The main conclusion of this work is that the order in which HCS test cases are run does matter.

The main results described in this chapter were presented in the Seventh IEEE International Conference on Software Testing, Verification, and Validation [114]. Our test case prioritization tool together with the feature models and the seeded faults used in our evaluation are available at `www.isa.us.es/~isaweb/anabsanchez/material.zip`.

Non–functional test case prioritization criteria for HCSs

Testing is questioning a system in order to evaluate it.

James M. Bach, software tester, author, trainer and consultant,

Variability testing techniques search for effective and manageable test suites that lead to the rapid detection of faults in systems with high configurability. Evaluating the effectiveness of these techniques in realistic settings is a must, but challenging due to the lack of HCSs with available code, automated tests and fault reports. In this chapter, we propose using the Drupal framework as a case study to evaluate variability testing techniques. First, we represent the framework variability using a feature model. Then, we report on extensive non–functional data extracted from the Drupal Git repository and the Drupal issue tracking system. Among other results, we identified 3,392 faults in single features and 160 faults triggered by the interaction of up to 4 features in Drupal v7.23. We also found positive correlations relating the number of bugs in Drupal features to their size, cyclomatic complexity, number of changes and fault history. To show the feasibility of our work, we evaluated the effectiveness of non–functional data for test case prioritization in Drupal. Results show that non–functional attributes are effective at accelerating the detection of faults, outperforming related prioritization criteria as test case similarity. Section §6.1 introduces the research problem. The Drupal framework is described in Section §6.2. Section §6.3 presents the Drupal feature model. A complete catalogue of non–functional feature attributes is depicted in Section §6.4. The number and types of faults detected in Drupal are presented in Section §6.5. Section §6.6 presents a correlation study exploring the relation among the reported non–functional attributes and the fault propensity of features. The results of using non–functional attributes to accelerate the detection of faults in Drupal are described in Section §6.7. We identify a number of basic requirements to apply our approach to other HCSs in Section §6.8. Section §6.9 presents the threats to validity of our work. Finally, we summarize the conclusions of this chapter in Section §6.10.

6.1 Introduction

As we explained in previous chapters, the testing of highly-configurable software systems is extremely challenging due to the potentially huge number of configurations under test. To address this problem, researchers have proposed various techniques to reduce the cost of testing in the presence of variability, including test case selection and test case prioritization techniques. *Test case selection* approaches select an appropriate subset of the existing test suite according to some coverage criteria (see Section §4.2). *Test case prioritization* approaches schedule test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goal, e.g. accelerate the detection of faults (see Section §4.3).

The number of works on highly-configurable testing is growing rapidly and thus the number of experimental evaluations. However, it is hard to find real HCSs with available code, test cases, detailed fault reports and good documentation that enable reproducible experiments [123, 127]. As a result, authors often evaluate their testing approaches using synthetic feature models, faults and non–functional attributes, which introduces threats to validity and weaken their conclusions. A related problem is the lack of information about the distribution of faults in HCSs, e.g. number and types of faults, fault severity, etc. This may be an obstacle for the design of new testing techniques since researchers are not fully aware of the type of faults that they are looking for.

In the search for real HCSs some authors have explored the domain of open source operating systems [15, 49]. However, these works mainly focus on the variability modelling perspective and thus ignore relevant data for testers such as the number of test cases or the distribution of faults. Also, repositories such as SPLOT [92, 131] and SPL2GO [130] provide catalogues of variability models and source code. However, they do not include information about the faults found in the programs and it is up to the user to inspect the code searching for test cases.

In order to find a real HCS with available code, we followed the steps of previous authors and looked into the open source community. In particular, we found the open source Drupal framework [18] to be a motivating HCS. Drupal is a modular web content management framework written in PHP [18, 140]. Drupal provides detailed fault reports including fault description, fault severity, type, status and so on. Also, most of the modules of the framework include a number of automated test cases. The high number of the Drupal community members together with its extensive documentation have also been strengths to choose this framework. Drupal is maintained and developed by a community of more than 630,000 users and developers.

In this chapter, we propose using the Drupal framework as a motivating case study to evaluate variability testing techniques. In particular, the following contributions are presented.

1. We map some of the main Drupal modules to features and represent the framework variability using a feature model. The resulting model has 48 features, 21 cross–tree constraints and represents more than 2,000 millions of different Drupal configurations.

2. We report on extensive non–functional data extracted from the Drupal Git repository. For each feature under study, we report its size, number of changes (during two years), cyclomatic complexity, number of test cases, number of test assertions, number of developers and number of reported installations. To the best of our knowledge, the Drupal feature model together with these non–functional attributes represents the largest attributed feature model published so far.

3. We present the number of faults reported on the Drupal features under study during a period of two years, extracted from the Drupal issue tracking system. Faults are classified according to their severity and the feature(s) that trigger it. Among other results, we identified 3,392 faults in Drupal v7.23, 160 of them caused by the interaction of up to 4 different features.

4. We replicated the study of faults in two consecutive Drupal versions, v7.22 and v7.23, to enable fault–history test validations, i.e. evaluate how the bugs detected in Drupal v7.22 could drive the search for faults in Drupal v7.23.

5. We present a correlation study exploring the relation among the non–functional attributes of Drupal features and their fault propensity. The results revealed statistically significant correlations relating the number of bugs in features to the size and cyclomatic complexity of its code, number of changes and number of faults in previous versions of the framework.

6. We present an experimental evaluation on the use of non–functional data for test case prioritization. The results show that non–functional attributes effectively accelerate the detection of faults of combinatorial test suites, outperforming related functional prioritization criteria such as variability coverage [43, 114] and similarity [5, 59, 114].

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

6.2 The Drupal framework

Drupal is a highly modular open source web content management framework implemented in PHP [18, 140]. It can be used to build a variety of web sites including internet portals, e-commerce applications and online newspapers [140]. Drupal is composed of a set of modules. A *module* is a collection of functions that provide certain functionality to the system. Installed modules in Drupal can be enabled or disabled. An enabled module is activated to be used by the Drupal system. A disabled module is deactivated and adds no functionality to the framework.

The modules can be classified into core modules and additional modules [18, 140]. *Core modules* are approved by the core developers of the Drupal community and are included by default in the basic installation of Drupal framework. They are responsible for providing the basic functionality that is used to support other parts of the system. The Drupal core includes code that allows the system to bootstrap when it receives a request, a library of common functions frequently used with Drupal, and modules that provide basic functionality like user management and templating. In turn, core modules can be divided into core compulsory and core optional modules. *Core compulsory modules* are those that must be always enabled while *core optional modules* are those that can be either enabled or disabled. Additional modules can be classified into contributed modules and custom modules and can be optionally installed and enabled. *Contributed modules* are developed by the Drupal community and shared under the same GNU Public License (GPL) as Drupal. *Custom modules* are those created by external contributors. Figure §6.1 depicts some popular core and additional Drupal modules.

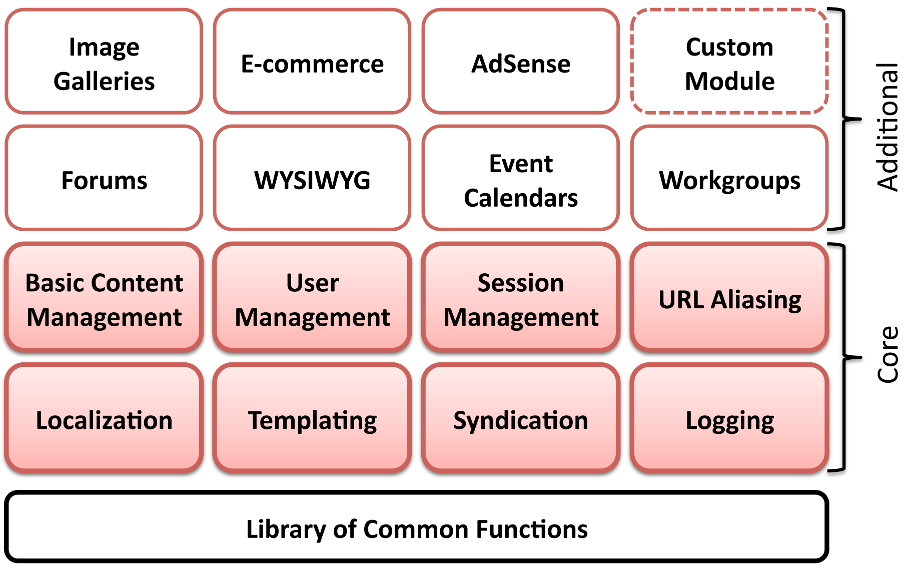


Figure 6.1: Several Drupal core and additional modules. Taken from [140]

6.2.1 Module structure

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

A Drupal *.info* file is a plain text file that describes the basic information required for Drupal to recognize the module. The name of this file must match the name of the module. This file contains a set of so-called directives. A *directive* is a property $name = value$. Some directives can use an array–like syntax to declare multiple values properties, $name[] = value$. Any line that begins with a semicolon (';') is treated as a comment. For instance, Listing 6.1 describes a fragment of the views.info file included in the *Views* module of Drupal v7.23:

Listing 6.1: Fragment of the file views.info

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

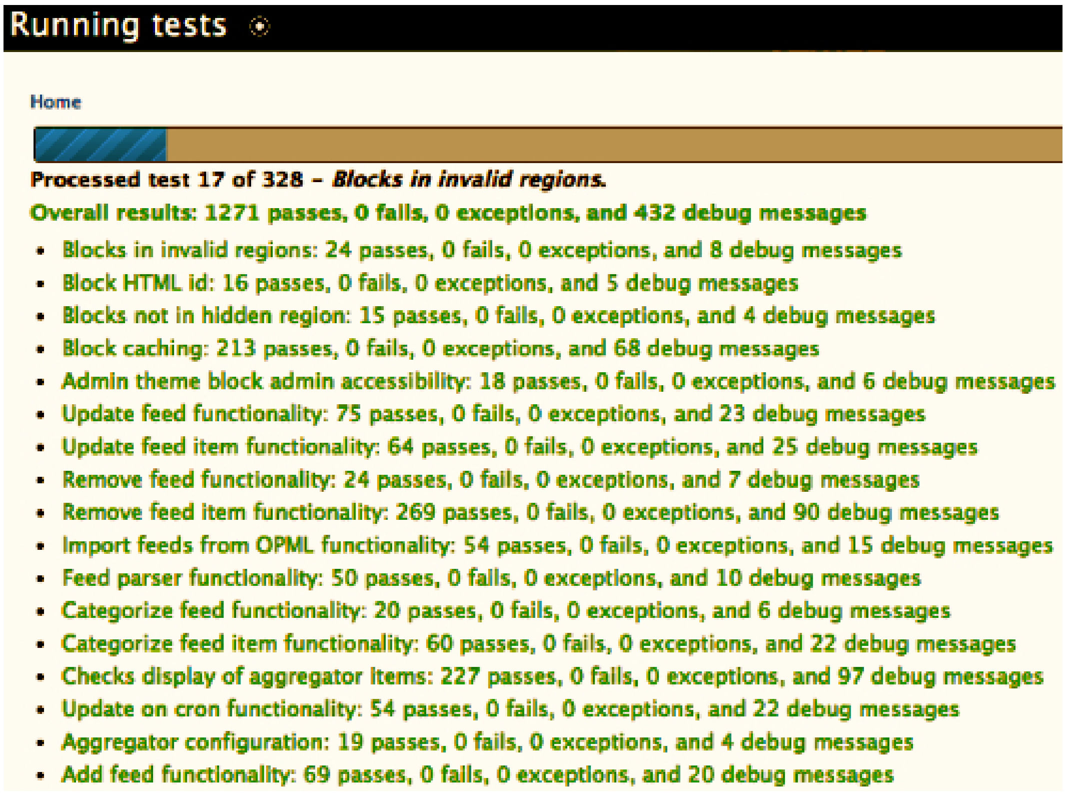
```
stylesheets[all][] = css/views.css

dependencies[] = ctools
; Handlers
files[] = handlers/views_handler_area.inc
files[] = handlers/views_handler_area_result.inc
files[] = handlers/views_handler_area_text.inc
... more
; Information added by drupal.org on 2014-05-20
version = "7.x-3.8"
core = "7.x"
project = "views"
```

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

6.2.2 Module tests

Drupal modules can optionally include a test directory with the test cases associated to the module. Drupal defines a test case as a class composed of functions (i.e. tests). These tests are performed through assertions, a group of methods that check for a condition and return a Boolean. If it is TRUE, the test passes, if FALSE, the test fails. There exist three types of tests in Drupal, unit, integration and upgrade tests. *Unit tests* are methods that test an isolated piece of functionality of a module, such as functions or methods. *Integration tests* test how different components (i.e. functionality) work together. These tests may involve any module of the Drupal framework. Integration tests usually simulate user interactions with the graphical user interface through HTTP messages. According to the Drupal documentation[1], these are the most common tests in Drupal. *Upgrade tests* are used to detect faults caused by the upgrade to a newer version of the framework, e.g. from Drupal v6.1 to v7.1. In order to work with tests in Drupal, it is necessary to enable the SimpleTest module. This module is a testing framework moved into core in Drupal v7. SimpleTest automatically runs the test cases of all the installed modules. Figure §6.2 shows a snapshot of SimpleTest while running the tests of Drupal v7.23 modules.


Running tests

Home

Processed test 17 of 328 – *Blocks in invalid regions.*

Overall results: 1271 passes, 0 fails, 0 exceptions, and 432 debug messages

- Blocks in invalid regions: 24 passes, 0 fails, 0 exceptions, and 8 debug messages
- Block HTML id: 16 passes, 0 fails, 0 exceptions, and 5 debug messages
- Blocks not in hidden region: 15 passes, 0 fails, 0 exceptions, and 4 debug messages
- Block caching: 213 passes, 0 fails, 0 exceptions, and 68 debug messages
- Admin theme block admin accessibility: 18 passes, 0 fails, 0 exceptions, and 6 debug messages
- Update feed functionality: 75 passes, 0 fails, 0 exceptions, and 23 debug messages
- Update feed item functionality: 64 passes, 0 fails, 0 exceptions, and 25 debug messages
- Remove feed functionality: 24 passes, 0 fails, 0 exceptions, and 7 debug messages
- Remove feed item functionality: 269 passes, 0 fails, 0 exceptions, and 90 debug messages
- Import feeds from OPML functionality: 54 passes, 0 fails, 0 exceptions, and 15 debug messages
- Feed parser functionality: 50 passes, 0 fails, 0 exceptions, and 10 debug messages
- Categorize feed functionality: 20 passes, 0 fails, 0 exceptions, and 6 debug messages
- Categorize feed item functionality: 60 passes, 0 fails, 0 exceptions, and 22 debug messages
- Checks display of aggregator items: 227 passes, 0 fails, 0 exceptions, and 97 debug messages
- Update on cron functionality: 54 passes, 0 fails, 0 exceptions, and 22 debug messages
- Aggregator configuration: 19 passes, 0 fails, 0 exceptions, and 4 debug messages
- Add feed functionality: 69 passes, 0 fails, 0 exceptions, and 20 debug messages

Figure 6.2: Running Drupal tests

6.3 The Drupal feature model

In this section, we describe the process followed to model Drupal v7.23 variability using a feature model, depicted in Figure §6.3. Feature models are the *de–facto* standard for software variability modelling [14, 69]. We selected this notation for its simplicity and its broad adoption in the field of variability testing.

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

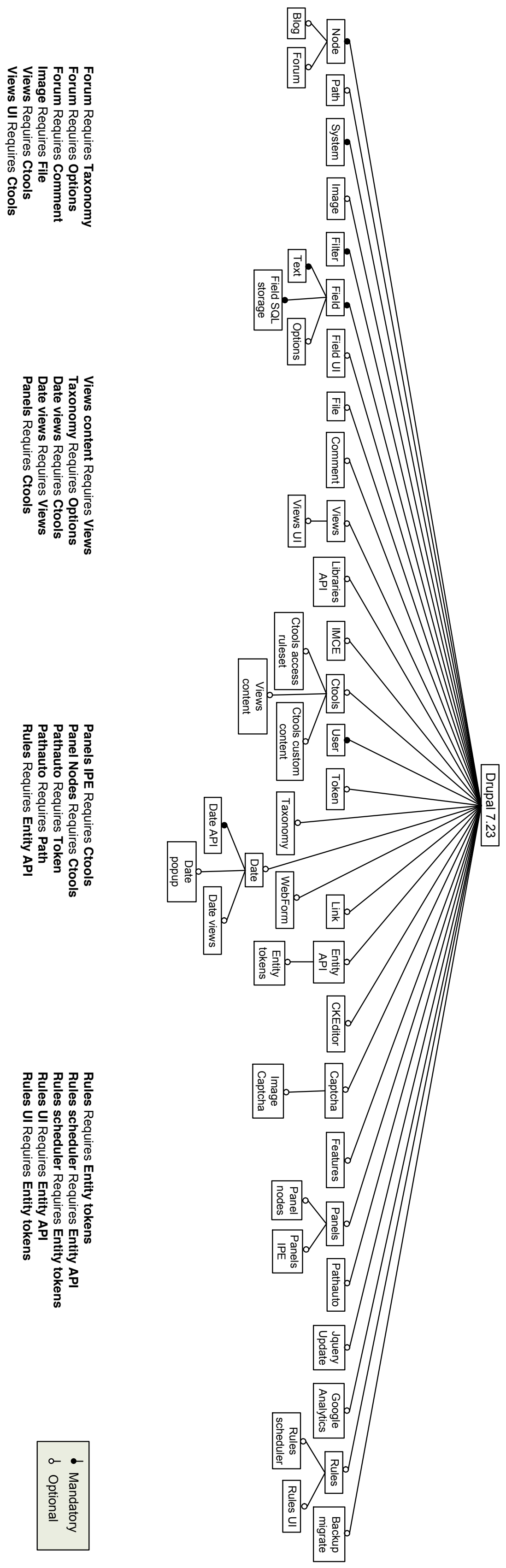


Figure 6.3: Drupal feature model

6.3.1 Feature tree

According to the Drupal documentation, each module that is installed and enabled adds a new *feature* to the framework [140] (chapter 1, page 3). Thus, we propose modelling Drupal modules as features of the feature model. Figure §6.3 shows the Drupal features that were considered in our study, 48 in total including the root feature. In particular, among the 44 core modules of Drupal, we first selected the Drupal core modules that must be always enabled (i.e. core compulsory modules), 7 in total, e.g. *Node*. In Figure §6.3, these features appear with a mandatory relation with the features root and *Field*. These features are included in all Drupal configurations. A *Drupal configuration* is a valid combination of features installed and enabled. Then, we selected 40 modules within the most installed Drupal core optional modules (e.g. *Path*) and additional modules (e.g. *Google Analytics*) ensuring that all dependencies were self-contained, i.e. all dependencies points at modules also included in our study. Most of these modules can be optionally installed and enabled and thus were modelled as optional features in the feature model. Exceptionally, the additional module *Date API* has a mandatory relation with its parent feature *Date*.

Submodules were mapped to subfeatures. Drupal submodules are those included in the directory of other modules. They provide extra functionality to its parent module and they have no meaning without it. As an example, the feature *Date* presents several subfeatures such as *Date API*, *Date popup* and *Date views*. Exceptionally, the submodules of *Node*, *Blog* and *Forum*, appear in separate module folders, however, the description of the modules in the Drupal documentation indicates that these modules are specializations of *Node* [140]. With respect to the set relationships *or* and *alternative*, typically found in feature models, none of them were identified among the features considered in Figure §6.3.

The selected modules are depicted in Table §B.1 in Appendix §B. In total, we considered 16 Drupal core modules and 31 additional modules obtaining a feature model with 48 features, i.e. root feature + 8 mandatory features + 39 optional features.

6.3.2 Cross–tree constraints

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

Cross–tree constraints were identified by manually inspecting the dependencies directive in the *.info* file of each module. For each dependency, we created a requires constraint in the feature model, 42 in total. For instance, consider the *views.info* file depicted in Listing 6.1. The file indicates that *Views* depends on the *Ctools* module, i.e. dependencies[] = ctools. Thus, we established a requires constraint between modules *Views* and *Ctools*. We may remark that 21 out of the 42 cross–tree constraints identified were redundant when considered together with the feature relationships in the tree. For instance, the constraint *Forum* requires *Field* is unnecessary since *Field* is a core feature included in all the Drupal configurations. Similarly, the constraint *Date popup* requires *Date API* can be omitted since *Date API* has a mandatory relationship with their parent feature *Date*. We manually identified and removed all redundant cross–tree constraints. This makes a total of 21 requires cross–tree constraints shown in Figure §6.3. No excludes constraints were identified among the modules. Interestingly, we found that all modules in the same version of Drupal are expected to work fine together. If a Drupal module has incompatibilities with others, it is reported as a bug that must be fixed. As an example, consider the bug for Drupal 6 titled "Incompatible modules"[2].

As a sanity check, we confirmed the constraints identified using the Javascript InfoVis Toolkit (JIT) Drupal module, which shows a graphical representation of the modules and their relationships [33]. Figure §6.4 depicts a fragment of the dependency graph provided by the JIT module showing a dependency (i.e. directed edge) between *Views* (source node) and *Ctools* (target node). Therefore, we confirm the requires cross–tree constraint found in the *views.info* file presented in Listing 6.1.

The ratio of features involved in cross–tree constraints to the total number of features in the model (CTCR) is 45.8%. This metric provides a rough idea of the complexity of the model and enables comparisons. Hence, for instance, Drupal is more complex (in terms of CTCR) than the models in the SPLOT repository (Avg. CTCR = 16.1%) [92, 131] and the models investigated by Bagheri et al [8] in their work about feature model complexity metrics (Avg. CTCR = 19.5%). Conversely, Drupal CTCR is less complex than the models reported by Berger et al. [15] in the context of operating systems (Avg. CTCR = 72.9%). This was expected since system software interacts with hardware in multiple ways and it is certainly more complex than Web applications. The Drupal feature model represents 2,090,833,920 configurations. In Section §6.10, we

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

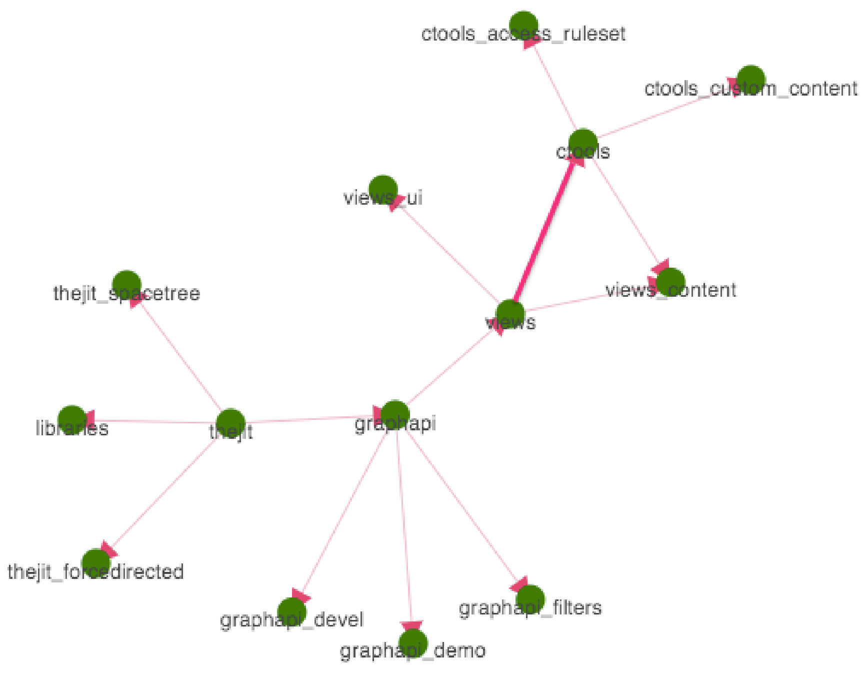


Figure 6.4: Dependency graph generated by the JIT module

provide the Drupal feature model in two different formats (SXFM and FaMa).

6.4 Non–functional attributes

In this section, we report a number of non–functional attributes of the features presented in Figure §6.3 extracted from the Drupal web site and the Drupal Git repository. These data are often used as good indicators of the fault propensity of a software application. Additionally, this may provide researchers and practitioners with helpful information about the characteristics of features in a real HCS. By default, the information was extracted from the features in Drupal v7.23. For the sake of readability, we used a tabular representation for feature attributes instead of including them in the feature model of Figure §6.3. Table §6.1 depicts the non–functional attributes collected for each Drupal feature, namely:

Feature size. This provides a rough idea of the complexity of each feature and its fault propensity [77, 89]. The size of a feature was calculated in terms of the number of Lines of Code (LoC). LoC were counted using the egrep and wc Linux commands on each one of the source files included in the module directory associated to each feature.

The command used is shown below. Blank lines and test files were excluded from the counting. The sizes range between 284 LoC (feature *Ctools custom content*) and 54,270 LoC (feature *Views*). It is noteworthy that subfeatures are significantly smaller than their respective parent features. The total size of the selected Drupal features is 336,025 LoC.

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

Cyclomatic Complexity (CC). This metric reflects the total number of independent logic paths used in a program and provides a quantitative measure of its complexity [103, 139]. We used the open source tool *phploc* [16] to compute the CC of the source code associated to each Drupal feature. Roughly speaking, the tool calculates the number of control flow statements (e.g. “if”, “while”) per lines of code [16]. The values for this metric ranged from 0.14 (feature *Path*) to 1.09 (feature *Entity token*). It is noteworthy that *Entity tokens*, the feature with highest CC, is one of the smallest features in terms of LoC (327).

Number of tests. Table §6.1 shows the total number of test cases and test assertions of each feature, obtained from the output of the *SimpleTest* module. In total, Drupal features include 352 test cases and 24,152 assertions. In features as *Ctools*, the number of test cases (7) and test assertions (121) is low considering that the size of the feature is over 17,000 LoC. It is also noteworthy that features such as *JQuery update*, with more than 50,000 LoC, have no test cases.

Number of reported installations. This depicts the number of times that a Drupal feature has been installed as reported by Drupal users. This data was extracted from the Drupal web site [18] and could be used as an indicator of the popularity or impact of a feature. Notice that the number of reported installations of parent features and their respective subfeatures is equal since they are always installed together, although they may be optionally enabled or disabled. Not surprisingly, the features with the highest number of reported installations are those included in the Drupal core (5,259,525 times) followed by *Views* (802,467 times) and *Ctools* (747,248 times), two of the most popular features in Drupal.

Number of developers. We collected the number of developers involved in the development of each Drupal feature. This could give us information about the scale and relevance of the feature as well as its propensity to faults related to the number of people working on it [89]. This information was obtained from the web site of each Drupal module as the number of committers involved [18]. The feature with the highest number of contributors is *Views* (178), followed by those included in the Drupal core (94) and *Ctools* (75).

Number of changes. Changes in the code are likely to introduce faults [54, 150]. Thus, the number of changes in a feature may be a good indicator of its error proneness and could help us to predict faults in the future. To obtain the number of changes made in each feature, we tracked the commits to the Drupal Git repository[3]. The search was narrowed by focusing on the changes performed during a period of two years, from May 1st 2012 to April 31st 2014. First, we cloned the entire Drupal v7.x repository. Then, we applied the console command showed below to get the number of commits by module, version and date. We collected the number of changes in Drupal v7.22 to check the correlation with the number of faults in Drupal v7.23 (see Section §6.6). As illustrated in Table §6.1, the number of changes ranged between 0 (feature *Blog*) and 90 (feature *Backup and migrate*). Interestingly, the eight features with the highest number of changes are optional. In total, we counted 557 changes in Drupal v7.22 during a two–years period.

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

6.5 Faults in Drupal

In this section, we present the number of faults reported in the Drupal features (i.e. modules) shown in Figure §6.3. The information was obtained from the issue tracking systems of Drupal[4] and related modules. In particular, we used the web–based search tool of the issue systems to filter the bug reports by severity, status, date, feature name and Drupal version. The search was narrowed by collecting the bugs reported in a period of two years, from May 1st 2012 to April 31st 2014. We collected the faults of two consecutive Drupal versions, v7.22 and v7.23, to achieve a better understanding of the evolution of a real system and to enable test validations based on fault–history (see Section §6.7).

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

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

Table 6.1: Non–functional feature attributes in Drupal

First, we filtered the faults by feature name (using the field "component"), framework version and the dates previously mentioned. Then, the search was refined to eliminate the faults not accepted by the Drupal community, those classified as *duplicated bugs*, *non reproducible bugs* and *bugs working as designed*. The latter are issues that have been considered not to be a bug because the reported behaviour was either an intentional part of the project, or the issue was caused by customizations manually applied by the user. A total of 3,401 faults matched the initial search for Drupal v7.22 and 3,392 faults for Drupal v7.23.

Second, we tried to identify those faults that were caused by the interaction of several features, i.e. integration faults. Our first approach was to follow the strategy presented by Artho et al. [7] in the context of operating systems. That is, for each feature, we searched for faults descriptions containing the keywords "break", "conflict" or "overwrite". However, the search did not return any result related to the interaction among features. We hypothesize that keywords related to potential bugs caused by the interaction among features may be domain–dependent. Thus, we followed a different approach. For each feature, we searched for bug reports that included the name (not case–sensitive) of some of the other features under study in its description or tags. As an example, consider the bug report depicted in Listing 6.2 extracted from the Drupal issue tracking system. The bug is associated to the module *Rules* and its description contains the name of the feature *Comment*. Thus, we considered it as a candidate integration fault between both features, *Rules* and *Comment*. In total, we detected 444 candidate integration faults in Drupal v7.22 and 434 in Drupal v7.23 following this approach. Interestingly, we found candidate faults containing the name of up to 9 different features.

Listing 6.2: Drupal bug report (http://www.drupal.org/node/1673836)

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

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

Next, we manually checked the bug reports of each candidate integration fault and discarded those that *i*) included the name of a feature but it actually made no reference to the feature (e.g. image), *ii*) included the name of Drupal features not considered in our study, and *iii*) included users and developers' comments suggesting that the fault was not caused by the interaction among features. We may remark that in many cases the description of the fault clearly suggested an integration bug, see as an example the last sentence in Listing 6.2: "*The only thing that gets back my pages without this fatal error is if I disable comments module*". The final counting revealed 160 integration faults in Drupal v7.23 and 170 in Drupal v7.22. In Drupal v7.23, we found that 3 out of 160 faults were caused by the interaction of 4 features, 25 were caused by the interaction of 3 features and 132 faults were triggered by the interaction between 2 features. It is noteworthy that 51 out of the 160 integration faults were caused by the interaction of *Views* with other features. *Views* enables the configuration of all the views of Drupal sites. Also, 31 integration faults were triggered by the interaction of *Ctools* with other features. *Ctools* provides a set of APIs and tools for modules to improve the developer experience. A complete list of the integration faults detected and the features involved on them is presented in Table §B.2 in Appendix §B. It is noteworthy that we did not find any specific keywords in the bugs' descriptions suggesting that they were caused by the interaction among features.

Table §6.2 summarizes the faults found in both versions of Drupal. For each feature, the total number of individual and integration faults in which it is involved are presented classifying them according to the reported severity level. We found that the bugs reported in additional Drupal modules (e.g. *Ctools*) do not discriminate among subversions of Drupal 7, i.e. they specify that the fault affects to Drupal v7.x. In those cases, we assumed that the bug equally affected to the two versions of Drupal under study, v7.22 and v7.23. This explains why the number of faults in most of the additional features is the same for both versions of the framework in Table §6.2. Notice, however, that in some cases the number of integration faults differs, e.g. feature *CKEditor*.

6.6 Correlation study

The information extracted from software repositories can be used, among other purposes, to drive the search for faults. In fact, multiple works in the field of software repository mining have explored the correlations between certain non–functional properties and metrics and the fault propensity of software components [7, 56, 61, 95,

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

Table 6.2: Faults in Drupal v7.22 and v7.23

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

Table 6.3: Spearman correlation results

99]. In this section, we investigate whether the non–functional attributes presented in Section §6.4 could be used to estimate the fault propensity of Drupal features.

The correlation study was performed using the R statistical environment [106] in two steps. First, we checked the normality of the data using the Shapiro-Wilk test concluding that the data do not follow a normal distribution. Second, we used the Spearman's rank order coefficient to assess the relationship between the variables. We selected Spearman's correlation because it does not assume normality, it is not very sensitive to outliers, and it measures the strength of association between two variables under a monotonic relationship[5], which fits well with the purpose of this study.

The results of the correlation study are presented in Table §6.3. For each correlation under study the following data are shown: identifier, measured variables, Spearman's correlation coefficient (ρ) and p-value. The Spearman's correlation coefficient takes a value in the range [-1,1]. A value of zero indicates that no association exists between the measured variables. Positive values of the coefficient reflect a positive relationship between the variables, i.e. as one variable decreases, the other variable also decreases and vice versa. Conversely, negatives values of the coefficient provide evidence of a negative correlation between the variables, i.e. the value of one variable increases as the value of the other decreases. Roughly speaking, the p–value represents the probability that the coefficient obtained could be the result of mere chance, assuming that the correlation does not exist. It is commonly assumed that p–values under 0.05 are enough evidence to conclude that the variables are actually correlated and that the

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

estimated ρ is a good indicator of the strength of such relationship, i.e. the result is statistically significant. The correlations that revealed statistically significant results are highlighted in grey in Table §6.3. By default, all correlations were investigated in Drupal v7.23 except several exceptions explicitly mentioned. The results of the correlation study are next discussed.

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

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

C3. Correlation between faults in consecutive versions of the framework. We investigated the correlation between the number of reported faults in Drupal features in two consecutive versions of the framework, v7.22 and v7.23. As mentioned in Section §6.5, the bugs reported in Drupal additional modules are not related to a specific subversion of the framework (they just indicate 7.x). Thus, in order to avoid biased results, we studied this correlation on the features of the Drupal core (16 out of 47) where the specific version of Drupal affected by the bugs is provided. We obtained a Spearman's correlation coefficient of 0.98 reflecting a very strong correlation between the number of faults in consecutive versions of Drupal features. This provides helpful information about the across–release fault propensity of features and it could certainly be useful to drive testing decisions when moving into a new version of the framework, e.g. test case prioritization.

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

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

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

C7-8. Correlation between faults and Cross–Tree Constraints (CTCs). In [43], Bagheri et al. studied several feature model metrics and suggested that those features involved in a higher number of CTCs are more error–prone. To explore this fact, we analyzed the features involved in CTCs in order to study the relation between them and their fault propensity. We obtained a correlation coefficient of 0.11 between the number of CTCs in which a feature is involved and the number of faults in that feature (C7). The correlation coefficient rose to 0.21 when considering integration faults only (C8). Therefore, we conclude that the correlation between feature involvement in CTCs and fault propensity is not confirmed in our study.

C9. Correlation between faults and Coefficient of Connectivity–Density (CoC). The CoC is a feature model complexity metric proposed by Bagheri et al. [8] and adapted by the authors for its use at the feature level [114]. The CoC measures the complexity of a feature as the number of tree edges and cross–tree constraints connected to it. We studied the correlation between the complexity of features in terms of CoC and the number of faults detected on them. The correlation study revealed a Spearman's coefficient of 0.28. Therefore, we cannot conclude that those features with a higher CoC are more error–prone. Nevertheless, we obtained a p–value very close to the threshold of statistical significance (0.051), therefore this relationship is a good candidate to be further investigated in future studies of different HCSs.

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

6.7 Evaluation

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

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

RQ2: *Are non–functional attributes more, less or equally effective than functional data at accelerating the detection of faults?* Some related works have proposed using functional information from the feature model to drive test case prioritization such as configuration similarity, commonality or variability coverage [43, 59, 114]. In this experiment, we measure whether non–functional attributes are more, less or equally effective than functional data at accelerating the detection of faults.

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

6.7.1 Experimental setup

The goal of this experiment is to use a combinatorial algorithm (e.g. ICPL) to generate a pair–wise suite for the Drupal feature model, and then, use different criteria based on functional and non–functional information to derive an ordering of test cases that allows detecting faults as soon as possible. The evaluation was performed in several steps. First, we seeded the feature model with the faults detected in Drupal v7.23, 3,392 in total. For this purpose, we created a list of faulty feature sets. Each set represents faults triggered by n features ($n \in [1,4]$). For instance, the list {{Node}{Ctools, User}} represents a fault in the feature *Node* and another fault caused by the interaction between the features *Ctools* and *User*. Second, we used the ICPL algorithm [65] to generate a pairwise suite for the Drupal feature model. We define a test case in this domain as a valid Drupal configuration, i.e. valid set of features. Among the whole testing space of Drupal (more than 2,000 millions of test cases), the ICPL algorithm returned 13 test cases that covered all the possible pairs of feature combinations. Then, we checked whether the pairwise suite detected the seeded faults. We considered that a test case detects a fault if the test case includes the feature(s) that trigger the bug. The pairwise test suite detected all the faults.

Next, we reordered the test cases in the pairwise suite according to different prioritization criteria. In order to answer RQ1, we prioritized the suite using the non–functional attributes that correlated well with the fault propensity, namely, the feature size, number of changes and number of faults in the previous version of the framework, i.e. Drupal v7.22. To answer RQ2, we reordered the test cases using two of the functional prioritization criteria that have provided better results in related evaluations, the similarity and VC&CC metrics. These prioritization criteria are fully described in Section §6.7.1. For each prioritized test suite, we measured how fast the faults of Drupal 7.23 were detected by each ordering calculating their Average Percentage of Faults Detected (APFD) values (see Section §3.5).

Finally, we repeated the whole process using the CASA algorithm [52] for the generation of the pairwise test suite for more heterogeneous results. Since CASA is not deterministic, we ran the algorithm 10 times for each prioritization criterion and calculated averages. The suites generated detected all the faults.

Prioritization criteria

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

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

$$PV_{Size}(t) = \sum_{i=1}^{|t|} loc(fi) \tag{6.1}$$

Fault–driven. This criterion measures the priority value of the test case as the sum of the number of bugs detected on its features. Inspired by the correlations described in Section §6.6, we propose counting the number of faults in Drupal v7.22 to effectively search for faults in Drupal v7.23. Let $faults(f,v)$ be the function returning the number of bugs in version v of feature f. The priority of t is calculated as follows:

$$PV_{Faults}(t) = \sum_{i=1}^{|t|} faults(fi, \text{``7.22''}) \tag{6.2}$$

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

$$PV_{Changes}(t) = \sum_{i=1}^{|t|} changes(fi, \text{``7.22''}) \tag{6.3}$$

Variability Coverage and Cyclomatic complexity (VC&CC). This criterion aims to get an acceptable trade off between fault coverage and feature coverage. It was presented by Bagheri et al. in the context of SPL test case selection [43] and later adapted by the authors for SPL test case prioritization [114]. In the previous Chapter §5, the work presented in [114], the authors compared several prioritization criteria for HCSs and found that VC&CC ranked first at accelerating the detection of faults. This motivated the selection of this prioritization criterion as the best of its breed. Given a feature model fm and a test case t, the VC&CC metric is calculated as follows:

$$PV_{VC\&CC}(t, fm) = \sqrt{vc(t, fm)^2 + cc(t, fm)^2} \tag{6.4}$$

where $vc(t, fm)$ calculates the variability coverage of a test case t for the feature model fm. The variability coverage of a test case is the number of variation points involved in it. A variation point is any feature that provides different variants to create a configuration. $cc(t, fm)$ represents the cyclomatic complexity of t. The cyclomatic complexity of a test case is calculated as the number of cross–tree constraints involved in it. We refer the reader to the previous Chapter §5 also presented in [114] for more details about this prioritization criterion.

Dissimilarity. The (dis)similarity metric has been proposed by several authors as an effective prioritization criterion to maximize the diversity of a test suite [59, 114]. Roughly speaking, this criterion gives a higher priority to those test cases with fewer features in common since they are likely to get a higher feature coverage than similar test cases. In Chapter §5, also presented in [114], the authors proposed to prioritize the test cases based on the dissimilarity metric using the Jaccard distance to measure the similarity among test cases. The Jaccard distance [137] is defined as the size of the intersection divided by the size of the union of the sample sets. In our context, each set represents a test case containing a set of features. The prioritized test suite is created by progressively adding the pairs of test cases with the highest Jaccard distance between them until all test cases are included. Given two test cases t_a and t_b, the distance between them is calculated as follows:

$$Dissimilarity(t_a, t_b) = 1 - \frac{|t_a \bigcap t_b|}{|t_a \bigcup t_b|} \tag{6.5}$$

The resulting distance varies between 0 and 1, where 0 denotes that the test cases

t_a and t_b are the same and 1 indicates that t_a and t_b share no features.

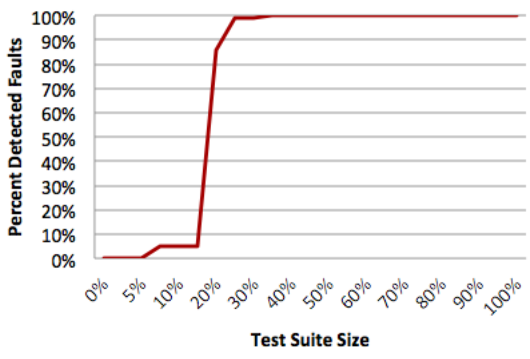
6.7.2 Experimental results

Table §6.4 depicts the results of the experiment. For each combinatorial testing algorithm, ICPL and CASA, the table shows the APFD values of the pairwise test suites prioritized according to the criteria presented in previous section. The first row shows the APFD of the suites when no prioritization is applied. The top three highest APFD values of each column are highlighted in bold. For ICPL, the prioritized suites based on non–functional attributes revealed the best results with fault–driven prioritization ahead (95.5%), followed by the size–driven (95.4%) and change–driven (95%) prioritization criteria. For CASA, the best APFD value was again obtained by the fault–driven prioritization criterion (93.4%), followed by the VC&CC (92.8%), size–driven (92.7%) and change–driven (91.8%) criteria. Overall, prioritization driven by non–functional attributes revealed the best rates of early fault detection followed by the functional prioritization criteria VC&CC and dissimilarity. Not surprisingly, all prioritization criteria accelerated the detection of faults of the unprioritized suites. It is noteworthy that the APFD values of the suites generated with CASA were lower than those of ICPL in all cases. We presume this is due to the internal ordering implemented as a part of the ICPL algorithm [65].

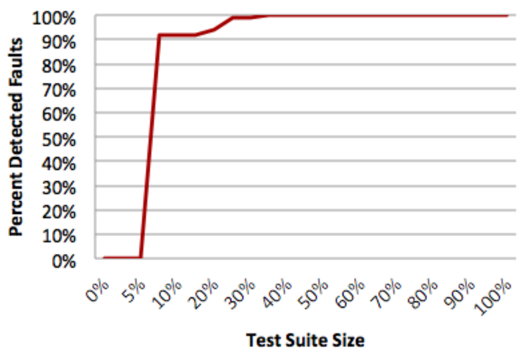
Prioritization criterion	**ICPL**	**CASA**
None	87.7	87.4
Size–driven	**95.4**	**92.7**
Fault–driven	**95.5**	**93.4**
Change–driven	**95.0**	91.8
VC&CC	93.5	**92.8**
Dissimilarity	92.7	87.9

Table 6.4: APFD values of the prioritized test suites

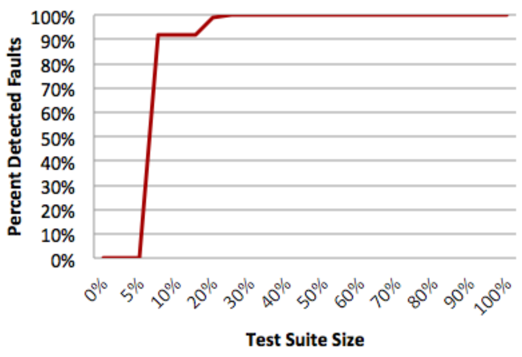
Figure §6.5 shows the percentage of detected faults versus the fraction of the ICPL prioritized test suites. Roughly speaking, the graphs show how the APFD value evolves as the test suite is exercised. Interestingly, all three non–functional prioritization criteria revealed 92% of the faults (3,120 out of 3,392) by exercising just 8% of their respective suites, i.e. 1 test case out of 13. In contrast, the original suite (unprioritized) detected just 5% of the faults (169 out of 3,392) with the same number of test cases. The fault-driven prioritized suite was the fastest suite in detecting all the faults (3,392) by using just 24% of the test cases (3 out of 13). All other orderings required 31% of the suite (4 out 13) to detect all the faults with the exception of the dissimilarity criterion, which required exercising 47% of the test cases (6 out of 13).



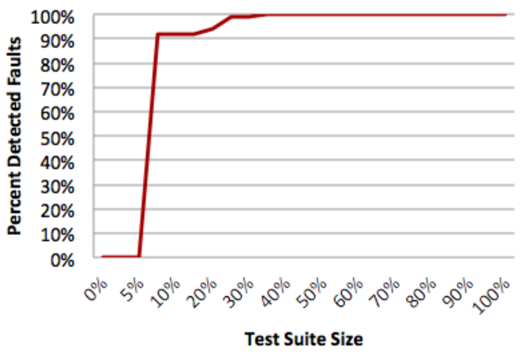
(a) No prioritization



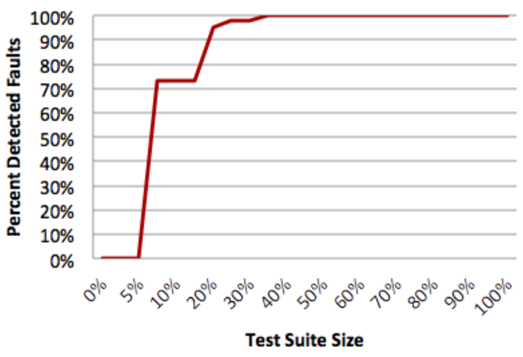
(b) Size–driven prioritization



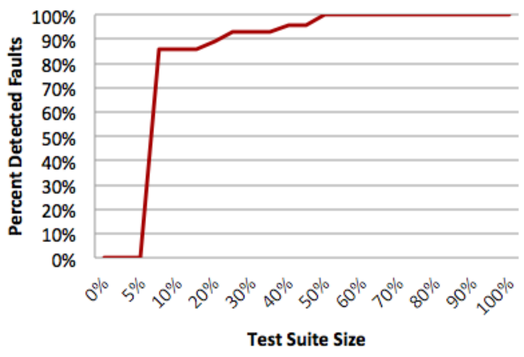
(c) Fault–driven prioritization



(d) Change–driven prioritization



(e) VC&CC



(f) Dissimilarity

Figure 6.5: Percentage of detected faults versus the fraction of the exercise suites

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

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

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

6.8 Applicability to other HCSs

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

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

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

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

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

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

6.9 Threats to validity

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

Internal validity. This refers to whether there is sufficient evidence to support the conclusions and the sources of bias that could compromise those conclusions. The re-engineering process could have influenced the final feature model of Drupal and therefore the evaluation results. To alleviate this threat, we followed a systematic approach and mapped Drupal modules to features. This is in line with the Drupal documentation, which defines an enabled module as a feature providing certain functionality to the system [140]. This also fits in the definition of feature given by Batory, who defines a feature as an increment in product functionality [10]. In turn, submodules were mapped to subfeatures since they provide extra functionality to its parent module and they have no meaning without it. Finally, we used the dependencies defined in the information file of each Drupal module to model CTCs.

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

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

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

The correlations and prioritization results reported are based on a single case study and thus cannot be generalized to other HCS. However, based on our experience, we believe that the described re–engineering process could be applicable to other open–source plug–in and module–based systems such as Wordpress or Prestashop, recently used in variability-related papers [96, 125]. We admit, however, that the described process could not be applicable to other domains with poorly documented variability. Despite this, our work does confirm the results of related works in software repository mining showing that can be found correlation between non–functional data and the fault propensity of software components. Similarly, our results show the efficacy of using non–functional attributes as driver for test case prioritization in HCSs.

6.10 Summary

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

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

Part of the results described in this chapter were presented in the 8th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS'14) [115]. An extension of that paper was published in the Special Issue of Software and System Modeling journal [118]. The Drupal feature model, non–functional attributes, source code of the evaluation and R scripts to reproduce the statistical analysis of the correlations are available at http://www.isa.us.es/anabsanchez-sosym14.

Multi–objective test case prioritization for HCSs

Quality is not an act, it is a habit.

Aristotle, philosopher and scientist (347 BC),

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 driven by a single objective which neglects the potential benefits of combining multiple criteria to guide the detection of faults. Second, evaluations are conducted using synthetic data rather than industry-strength case studies, which provides no information about the effectiveness of different prioritization objectives. In this chapter, 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 the combination of prioritization objectives, where at least one is non–functional, typically results in faster fault detection. In Section §7.1, we introduce the research problem. Section §7.2 defines the concepts of feature models and multi-objective evolutionary algorithms. Section §7.3 presents the Drupal case study used to perform this work. In Section §7.4 and Section §7.5 we respectively describe the overview and definition of our approach and the multi-objective optimization algorithm proposed. Section §7.6 defines seven objective functions for HCSs based on functional and non-functional goals. The evaluation of our approach is described in Section §7.7. Section §7.8 presents the threats to validity of our work. Finally, we summarize our conclusions in Section §7.9.

7.1 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 [22, 144]. Testing HCSs is extremely challenging due to the potentially huge number of configurations under test. 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 [30, 58, 88]. *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 [5, 82, 146]. 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 [146], configuration dissimilarity [5, 59, 114] or feature model complexity metrics [114, 118]. Non–functional prioritization objectives consider extra–functional information such as user preferences [42, 68], cost [146], memory consumption [82] or execution probability [29] to find the best ordering for test cases. In the previous chapter, work published in [118], 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 [5, 29, 42, 59, 68, 82, 118], 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 [146] 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 [81]. 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 [5, 58, 105, 149]. 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 chapter, 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 (see Chapter §6 or cite [118] for more information). Results reveal that non–functional properties such as the number of changes in the code accelerate the detection of faults more effectively than functional properties extracted from the feature model. Results also suggest that the combination of prioritization objectives, where at least one is non–functional, usually results in faults being detected faster. In summary, we present the following contributions.

1. Seven novel prioritization objective functions for HCSs: four functions based on functional information extracted from the feature model, and three functions based on non–functional data typically available in source version control and issue tracking repositories.
2. A novel multi–objective evolutionary algorithm for the generation of prioritized test suites for HCSs.
3. A comparison of the effectiveness of 63 different combinations of up to three

objective functions in accelerating the detection of faults in Drupal. To the best of our knowledge, this is the first work on multi-objective test case prioritization in HCSs using industry-strength data.

7.2 Background

7.2.1 Feature Models

As we described in Chapter §2, a *feature model* defines all the possible configurations of a system or family of related systems [14, 69]. A feature model is visually represented as a tree–like structure in which nodes represent features, and edges denote the relationships among them. Recall that a *feature* can be defined as any increment in the functionality of the system [142]. A *configuration* of the system is composed of a set of features satisfying all the constraints of the model. Figure §7.1 shows a feature model describing a simplified family of mobile phones.

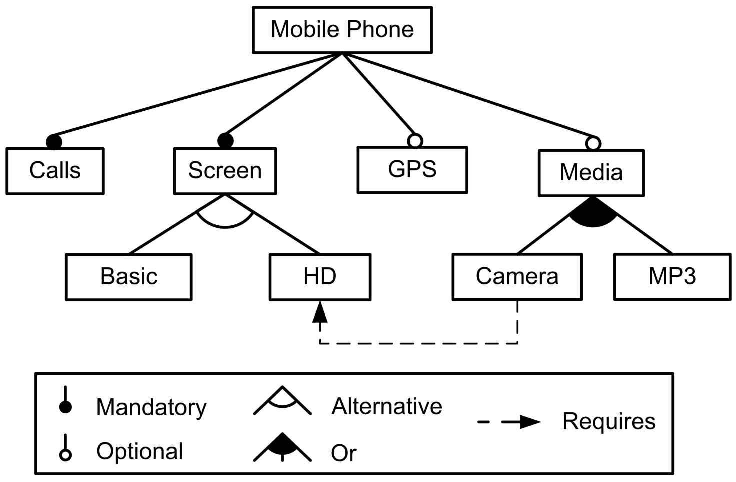


Figure 7.1: Mobile phone feature model

Feature models can be extended with additional information by means of feature attributes, these are called *attributed or extended feature models* [14]. 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 §7.1 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 Figure §7.1.

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

Table 7.1: Mobile phone feature attributes

7.2.2 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 Figure §7.2. 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

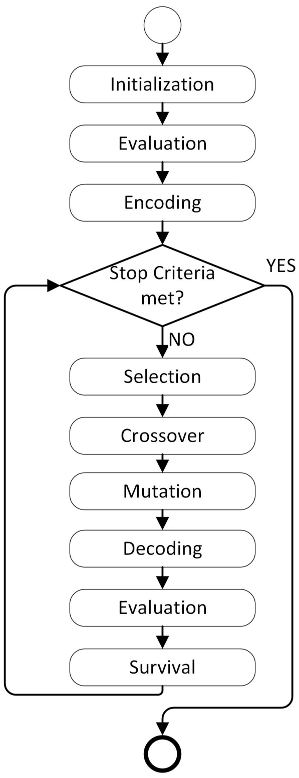


Figure 7.2: Working scheme of evolutionary algorithm

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) [28] has become very popular due to its effectiveness in many of the benchmarks in multi–objective optimization [27, 151].

7.3 The Drupal case study

In this section, we briefly explain the Drupal case study that we fully reported in Section §6 and in the work [118]. Drupal is a highly modular open source web content management framework written in PHP [18, 140]. This tool can be used to build a variety of websites including internet portals, e-commerce applications and online newspapers [140]. 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.

Figure §7.3 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 Figure §7.3. 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. The Drupal feature model represents $2.09E9$ different configurations [118].

In this work, we model the non-functional data from Drupal as feature attributes, depicted in Table §7.2. These data were obtained from the Drupal website, the Drupal Git repository and the Drupal issue tracking system (see Chapter §6 or work [118]). In particular, we use the following attributes:

- *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`).

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

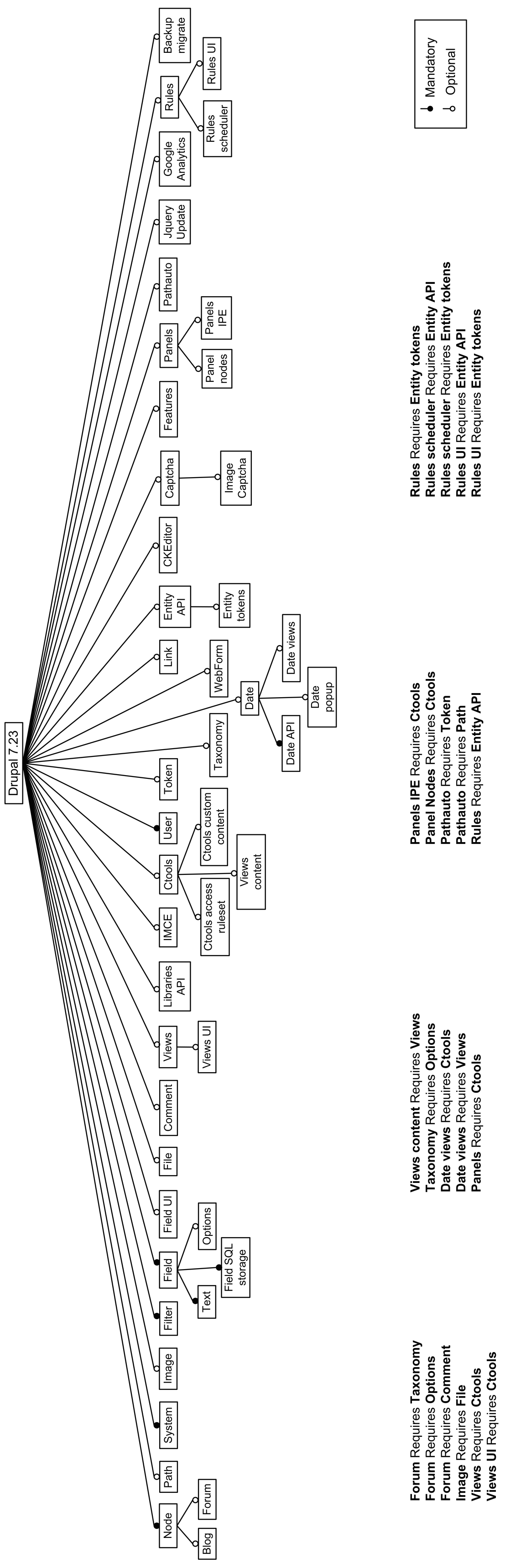


Figure 7.3: Drupal feature model

- *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. The number of faults ranges from 0 in features as `Options` to 1,091 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 §7.2.

7.4 Approach overview

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

7.4.1 Problem

The classical problem of test case prioritization consist in scheduling test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goal [111]. 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 Figure §7.1: {`Mobile Phone, Calls, Screen, Basic, Media, MP3`}.

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

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	3,115	15	17	1	17	1
CKEditor	13,483	40	197	11	197	9
Comment	5,627	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	2,696	9	44	3	44	3
Date API	6,312	11	41	1	41	1
Date popup	792	4	30	1	30	1
Date views	2,383	6	25	1	25	1
Entity API	13,088	14	175	18	175	18
Entity tokens	327	1	22	6	22	6
Features	8,483	72	97	9	97	9
Field	8,618	7	45	18	48	17
Field SQL sto.	1,292	2	3	2	3	2
Field UI	2,996	3	13	2	11	1
File	1,894	1	10	5	11	5
Filter	4,497	3	19	5	19	5
Forum	2,849	2	6	4	5	4
Google ana.	2,274	14	11	1	11	1
Image	5,027	3	10	8	9	6
Image captcha	998	0	3	0	3	0
IMCE	3,940	9	9	5	9	5
Jquery update	50,762	1	64	12	64	12
Libraries API	1,627	7	11	0	11	0
Link	1,934	11	82	4	82	4
Node	9,945	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	1,462	20	19	2	19	2
Path	1,026	20	3	1	2	1
Pathauto	3,429	2	54	9	54	9
Rules	13,830	5	240	15	240	15
Rules sch.	1,271	4	13	0	13	0
Rules UI	3,306	1	26	0	26	0
System	20,827	16	35	5	35	4
Taxonomy	5,757	4	15	22	19	22
Text	1,097	1	6	3	5	3
Token	4,580	10	37	7	37	7
User	8,419	12	20	25	19	22
Views	54,270	27	1,091	51	1,091	51
Views content	2,683	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**	

Table 7.2: Non–functional feature attributes in Drupal

Test suite. A test suite is an ordered set of test cases. Table §7.3 depicts a sample test suite of the model presented in Figure §7.1.

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.

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

Table 7.3: Mobile phone test suite

Given a feature model representing the HCS under test and an objective function, the problem of test case prioritization in HCSs consist in generating a test suite that optimize 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., optimal test suites) if there not exist a single solution that simultaneously optimizes all the objectives.

7.4.2 Our approach

In this work, we propose to model the multi–objective test case prioritization problem in HCSs as a multi–objective optimization problem. Figure §7.4 illustrates our approach. Given an attributed feature model and a set of objectives used as fitness functions, the problem consists in finding a set of optimal solutions (i.e., test suites) that optimize the target objectives. Once these optimal solutions are obtained, the user may identify the preferred suite and select it as the solution to the test case prioritization problem. This can be done either manually or automatically using other metrics. In this work we propose using the *Average Percentage of Faults Detected (APFD)* [39, 111, 134] metric (see Section §3.5) 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.

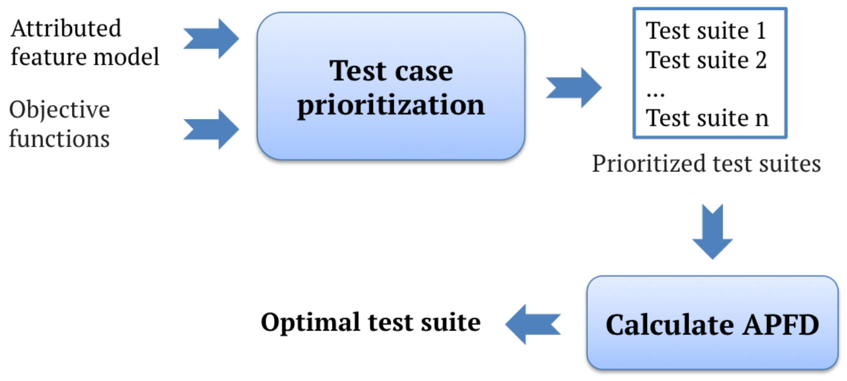


Figure 7.4: Our multi–objective test case prioritization approach for HCSs

7.4.3 Illustrative example

Table §7.4 shows the information of four test suites, using the test cases of Table §7.3. 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 §7.6. 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.

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

Table 7.4: A set of test suites for the mobile phone

Figure §7.5 depicts the Pareto front obtained when trying to find a test suite that maximizes both objectives. As denoted in the call-out of Figure §7.5, 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. Consider the faults detected by each test case shown in Table §7.5. 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%:

$1 - \frac{1+1+2+3+4+5}{5\times6} + \frac{1}{2\times5} = 0.57$

TS3 an APFD of 80%:

$1 - \frac{1+1+1+1+2+3}{5\times6} + \frac{1}{2\times5} = 0.8$

and TS4 an APFD of 53%:

$1 - \frac{3+4+3+4+2+1}{5\times6} + \frac{1}{2\times5} = 0.53$

Based on the previous results, TS3 is faster than TS1 and TS2 and therefore it would be the solution selected from the Pareto front.

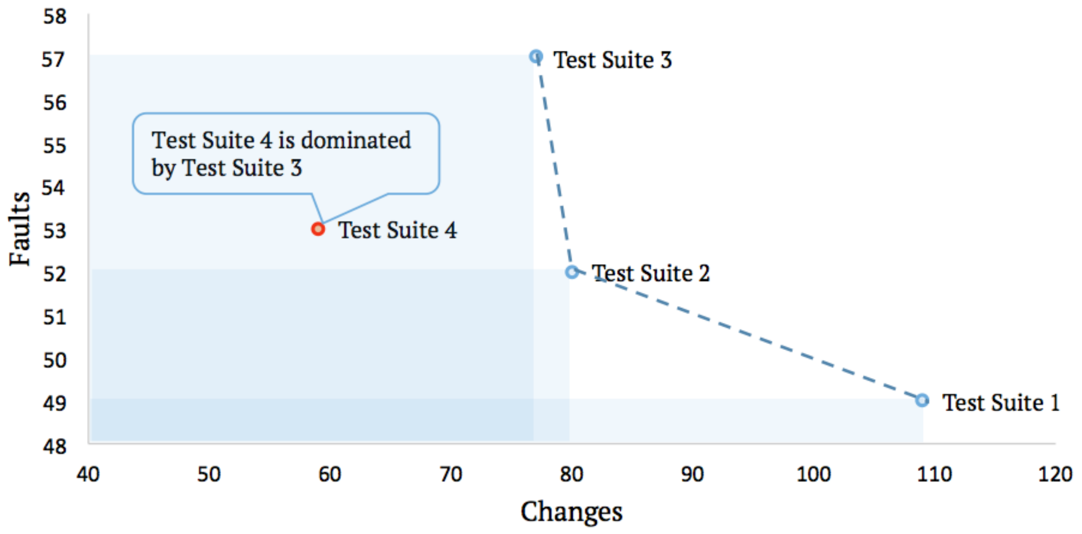


Figure 7.5: Test suites of table §7.4 as a pareto front for objectives *Changes* and *Faults* (both to be maximized)

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

Table 7.5: Test suite and faults exposed

7.5 Multi-objective optimization algorithm

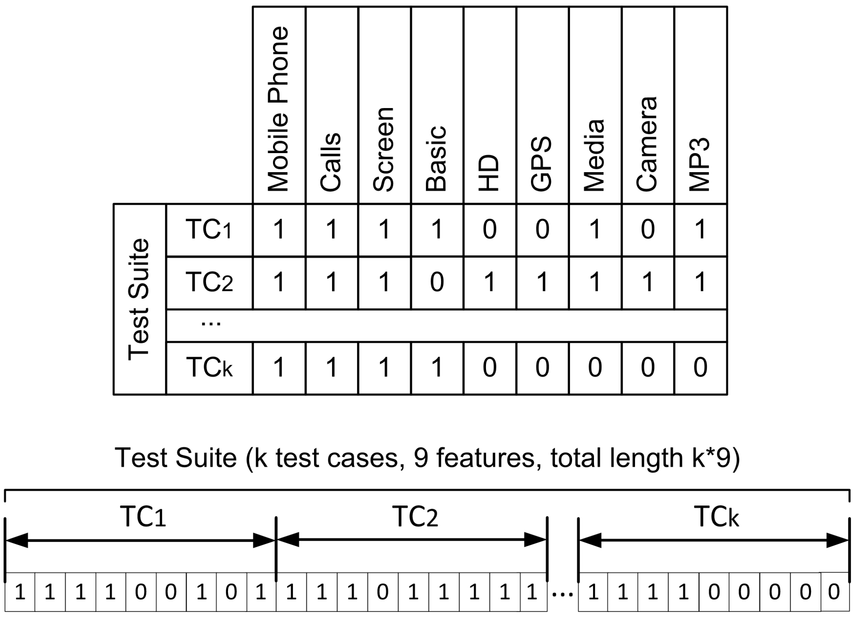
We used a MOEA to solve the multi–objective test case prioritization problem in HCSs. In particular, we adapted NSGA-II due to its popularity and good performance for many multi-objective optimization problems. In short, the algorithm receives an attributed feature model and a set of objective functions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

7.5.1 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 [122]. Figure §7.6 illustrates a test suite with its corresponding encoding based on the feature model showed in Figure §7.1 (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.

7.5.2 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. In [81], Lopez-Herrejon et al. 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



Test Suite		Mobile Phone	Calls	Screen	Basic	HD	GPS	Media	Camera	MP3
	TC1	1	1	1	1	0	0	1	0	1
	TC2	1	1	1	0	1	1	1	1	1
	...									
	TCk	1	1	1	1	0	0	0	0	0

Figure 7.6: Test suite encoding as a binary vector

random suites. Based on their finding, our initial population is composed of different orderings of a pairwise test suite generated by the CASA tool [51, 52] from the input feature model.

7.5.3 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. Figure §7.7 illustrates a sample crossover operation between two chromosomes of different sizes.

7.5.4 Mutation operators

We implemented three different mutation operators detailed below.

- *Test case swap*. This mutation operation exchanges the ordering of two randomly

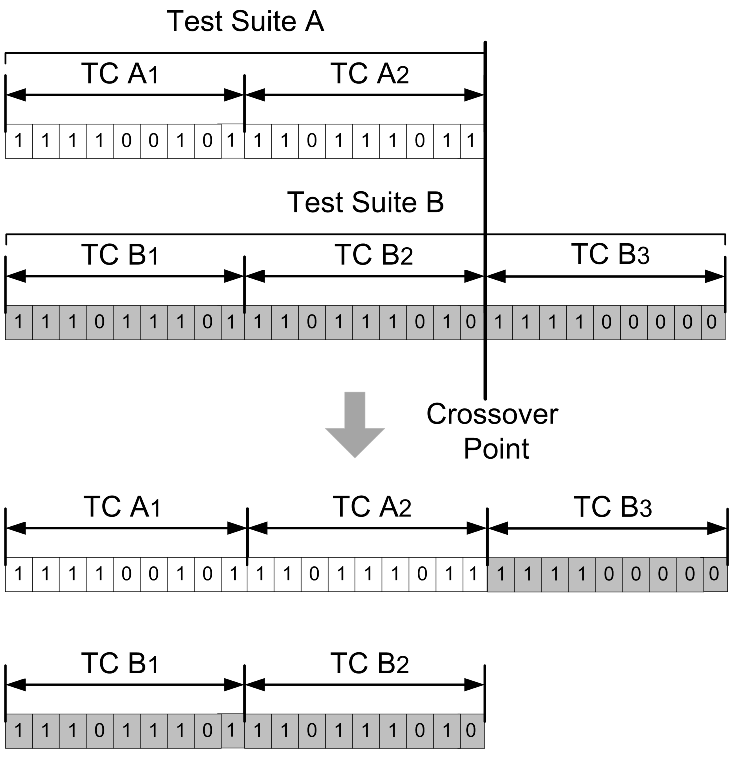


Figure 7.7: Crossover operator

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.
- *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.

7.6 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 §7.7). To illustrate each function, we use the feature model in Figure §7.1 as fm and the test suite $ts = [TC1, TC2]$ with two of the test cases shown in Table §7.3, which we reproduce next:

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

7.6.1 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 [8]. In Chapter §5, published in [114], 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 $tc_0..tc_{i-1}$ of test suite ts. This objective function is defined as follows:

$$Connectivity(fm, ts) = \sum_{i=0}^{|ts|-1} (|ts| - i) * coc(fm, ts.tc_i) \tag{7.1}$$

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 0 and TC2 in position 1, we calculate the function $Connectivity$ as follows:

$$Connectivity(fm, ts) = (2 - 0) * 13 \; + \; (2 - 1) * 5 = 2 * 13 + 1 * 5 = 31$$

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 [59, 114]. 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 $tc_0..tc_{i-1}$. This objective function is defined as follows:

$$Dissimilarity(fm, ts) = \sum_{i=0}^{|ts|-1} (|ts| - i) * df(fm, ts.tc_i) \tag{7.2}$$

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 0 and `TC2` in position 1, we calculate the function *Dissimilarity* as follows:

$$Dissimilarity(fm, ts) = (2-0) * 6 \ + \ (2-1) * 3 = 2 * 6 + 1 * 3 = 15$$

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. [47, 59, 114]). 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 $tc_0..tc_{i-1}$. This objective function is defined as follows:

$$Pairwise(fm, ts) = \sum_{i=0}^{|ts|-1} (|ts| - i) * pc(fm, ts.tc_i) \tag{7.3}$$

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 0 and TC2 in position 1, we calculate the function *Pairwise* as follows:

$$Pairwise(fm, ts) = (2-0)*36 \ + \ (2-1)*27 = 2*36+1*27 = 99$$

Variability Coverage and Cyclomatic Complexity. From a feature model, *Cyclomatic Complexity* measures the number of cross-tree constraints [8], while *Variability Coverage* measures the number of variation points [43]. 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 [43, 114]. 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 $tc_0..tc_{i-1}$. This objective function is defined as follows:

$$VCoverage(fm, ts) = \sum_{i=0}^{|ts|-1} (|ts| - i) * vc(fm, ts.tc_i) \tag{7.4}$$

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 0 and TC2 in position 1, we calculate the function *VCoverage* as follows:

$$VCoverage(fm, ts) = (2-0)*3 \ + \ (2-1)*3 = 2*3+1*3 = 9$$

7.6.2 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. [54, 150]). 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. 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 $tc_0..tc_{i-1}$. 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:

$$Changes(fm, ts) = \sum_{i=0}^{|ts|-1} (|ts| - i) * nc(fm, ts.tc_i) \tag{7.5}$$

Please refer to Table §7.1. 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 0 and `TC2` in position 1, we calculate the function *Changes* as follows:

$$Changes(fm, ts) = (2 - 0) * 29 \ + \ (2 - 1) * 22 = 2 * 29 + 1 * 22 = 80$$

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 [62, 129]. 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. 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 $tc_0..tc_{i-1}$. 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:

$$Faults(fm, ts) = \sum_{i=0}^{|ts|-1} (|ts| - i) * nf(fm, ts.tc_i) \tag{7.6}$$

Please refer to Table §7.1. 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 0 and TC2 in position 1, we calculate the function *Faults* as follows:

$$Faults(fm,ts) = (2-0)*27 \ + \ (2-1)*17 = 2*27+1*17 = 71$$

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 [77, 89, 118]. 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 $tc_0..tc_{i-1}$. This objective function is defined as follows:

$$Size(fm,ts) = \sum_{i=0}^{|ts|-1} (|ts|-i) * fs(fm,ts.tc_i) \tag{7.7}$$

Please refer to Table §7.1. 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 0 and TC2 in position 1, we calculate the function *Size* as follows:

$$Size(fm,ts) = (2-0)*3690 \ + \ (2-1)*1650 = 2*3690+1*1650 = 9030$$

7.7 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.

7.7.1 Research questions

In previous chapters, we investigated the effectiveness of functional (Chapter §5) and non–functional (Chanper §6) test case prioritization criteria for HCSs from a single–objective perspective. In this chapter, 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?*

7.7.2 Experimental setup

To answer our research questions, we implemented the algorithm and the objective functions described in Sections §7.5 and §7.6 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 algorithm was executed with all the possible combinations of 1, 2 and 3 of the objectives functions described in Section §7.6, 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 algorithm was executed 30 times to perform statistical analysis of the data. The configuration parameters of the algorithm are depicted in Table §7.6. These were selected based on the recommended parameters for NSGA-II [28] 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 algorithm was implemented using jMetal [34], a Java framework to solve multi–objective optimization problems. The non-functional objective functions were calcu-

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	$0.3 * (1/N)$

Table 7.6: Parameter settings for the evolutionary algorithm

lated using the Drupal feature attributes reported in Table §6.1. 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 [67] which generates all the possible pairs of features of an input feature model.

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.

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 §3.5. 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 NSGA-II and the initial pairwise suite generated by the CASA algorithm [51, 52] on each execution.

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.

7.7.3 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 §7.7. For each set of objectives, the table shows the results of 30 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 §7.7 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 0.934 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 differences between mono and multi–objective prioritization are small except for the *Pairwise* objective, which produced the worst results. This finding is also observed in the box plot of Figure §7.8 which illustrates the distributions of the maximum APFD values found on each execution of NSGA-II (30 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.

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 + 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 [87].

As a further analysis, we used Vargha and Delaney's $\widehat{A_{12}}$ statistic [6] to evaluate the effect size, i.e., determine which mono or multi–objective combinations perform better and to what extent. Table §7.8 shows the effect size statistic and the conclusions

Objectives	NSGA-II			CASA	
	Avg	Avg Max	Max	Avg	Max
Connectivity	0.942	0.943	0.958	0.882	0.954
Dissimilarity	**0.944**	0.945	0.954	0.885	0.940
Pairwise	0.894	0.895	0.952	0.882	0.934
VCoverage	0.938	0.943	0.957	0.866	0.935
Connectivity + Dissimilarity	**0.944**	0.946	0.956	0.866	0.924
Connectivity + Pairwise	0.935	0.945	**0.960**	0.876	0.942
Connectivity + VCoverage	**0.944**	0.945	**0.959**	0.866	0.943
Dissimilarity + Pairwise	0.934	0.946	0.957	0.867	0.940
Dissimilarity + VCoverage	0.943	0.945	0.958	0.864	0.933
Pairwise + VCoverage	0.926	0.943	0.954	0.883	0.940
Connectivity + Dissimilarity+ Pairwise	0.934	0.945	0.955	0.864	0.938
Connectivity + Dissimilarity + VCoverage	0.942	0.944	**0.959**	0.866	0.938
Connectivity + Pairwise + VCoverage	0.931	0.947	0.958	0.868	0.928
Dissimilarity+ Pairwise + VCoverage	0.929	0.943	0.954	0.876	0.944
Average	**0.934**	**0.941**	**0.960**	**0.872**	**0.954**

Table 7.7: APFD values achieved by functional prioritization objectives

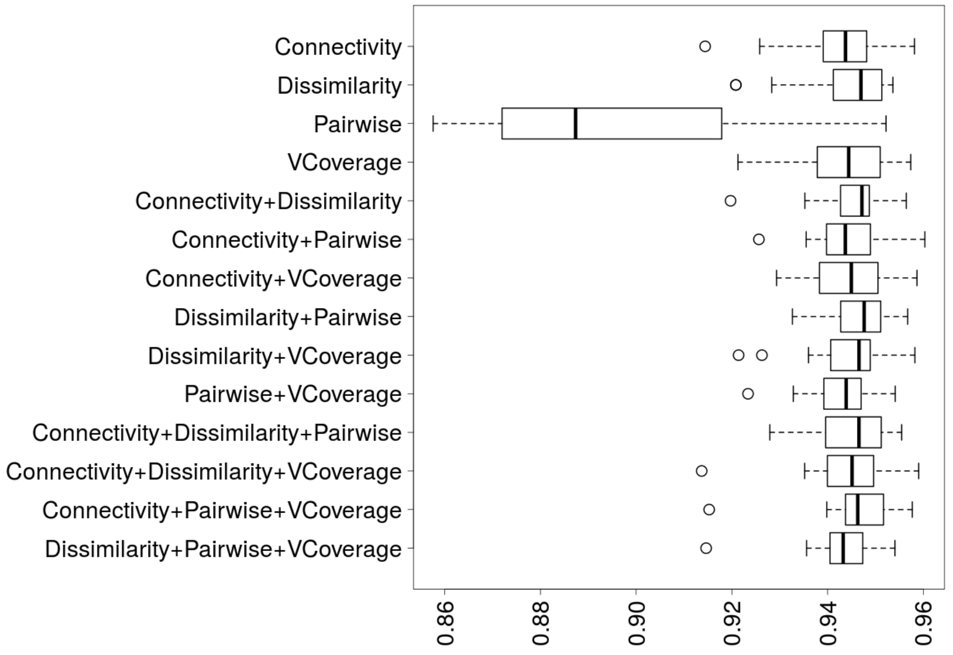


Figure 7.8: Box plot of the maximum APFD achieved on each execution (30 in total)

of hypothesis Tests. 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 [143] 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. It is also observed a noticeably superiority of the multi–objective *Connectivity* + *Pairwise* + *VCoverage* when compared to *Connectivity* as a mono–objective ($\widehat{A_{12}} = 0.66$). General results, however, suggest that there is no difference in the rate of fault detection achieved by mono–objective (except for *Pairwise*) against multi–objective prioritization using functional objectives.

Functional Multi-Objective	**Functional Mono-Objective**				CASA
	Connectivity	Dissimilarity	Pairwise	VCoverage	
Connectivity + Dissimilarity	0.620	0.491	**0.952**	0.577	**0.999**
Connectivity + Pairwise	0.555	0.419	**0.947**	0.506	**0.99**
Connectivity + VCoverage	0.569	0.470	**0.947**	0.536	**0.974**
Dissimilarity + Pairwise	0.636	0.508	**0.957**	0.581	**0.990**
Dissimilarity + VCoverage	0.582	0.446	**0.947**	0.533	**0.998**
Pairwise + VCoverage	0.500	0.377	**0.943**	0.464	**0.985**
Connectivity + Dissimilarity + Pairwise	0.560	0.462	**0.950**	0.529	**0.992**
Connectivity + Dissimilarity + VCoverage	0.562	0.426	**0.943**	0.513	**0.993**
Connectivity + Pairwise + VCoverage	**0.664**	0.524	**0.956**	0.607	**0.996**
Dissimilarity + Pairwise + VCoverage	0.516	0.360	**0.940**	0.466	**0.977**
CASA	**0.048**	**0.012**	0.374	**0.021**	-

Table 7.8: $\widehat{A_{12}}$ 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 (p-value < 0.05)

7.7.4 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 §7.9 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 top three best average and maximum APFD values (in boldface). This suggests that the number of faults in previous versions of the system is a key factor to accelerate the detection of faults. It is noteworthy that all the test suites generated detected at least 99% of the emulated faults.

Objectives	**NSGA-II**			**CASA**	
	Avg	**Avg Max**	**Max**	**Avg**	**Max**
Changes	0.944	0.944	0.956	0.874	0.935
Faults	**0.952**	0.952	**0.957**	0.857	0.921
Size	0.948	0.948	0.956	0.870	0.937
Changes + Faults	**0.950**	0.952	**0.960**	0.869	0.941
Changes + Size	0.946	0.950	0.956	0.877	0.930
Faults + Size	**0.950**	0.950	0.956	0.878	0.936
Changes + Faults + Size	0.948	0.951	**0.958**	0.864	0.928
Average	**0.948**	**0.950**	**0.960**	**0.870**	**0.941**

Table 7.9: APFD values achieved by non-functional prioritization objectives

Figure §7.9 depicts a box plot of the distributions of the maximum APFD value achieved on each execution of NSGA-II. The best median APFD value was achieved by the *Faults* objective function, and the lowest median value was obtained by the objective *Changes*. Interestingly, however, the maximum APFD value was obtained by the combination of both of them, *Changes* + *Faults*. This is a good example of the potential benefits of combining multiple objectives.

Table §7.10 shows the values of the $\widehat{A_{12}}$ effect size and the conclusions of hypothesis tests. CASA is excluded from the table since it was clearly outperformed by all other objectives. As observed in Table §7.10, the results point at *Faults* as the best objective with all the values of the column under 0.5. It is noteworthy, however, that all the multi–objective combinations improve the results obtained by *Changes* and *Size*. That

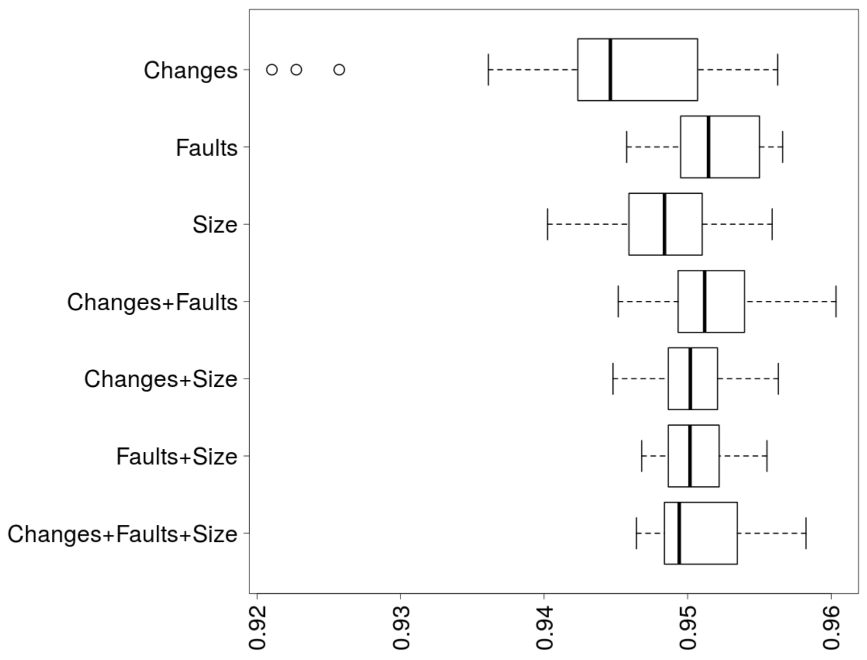


Figure 7.9: Box plot of the maximum APFD achieved on each execution (30 in total)

is, the rate of fault detection achieved by *Changes* and *Size* is always improved when combined with another non–functional objective. Overall, the best multi–objective combination was that formed by *Changes* + *Faults* with $\widehat{A_{12}}$ values over 0.72 in 2 out of 3 comparisons. No clear differences were found between the use of multi–objective prioritization with two or three objectives.

Non-Functional Multi-Objective	**Mono-Objective**		
	Changes	Faults	Size
Changes + Faults	**0.779**	0.451	**0.721**
Changes + Size	**0.724**	**0.337**	**0.649**
Faults + Size	**0.741**	**0.350**	**0.666**
Changes + Faults + Size	**0.759**	0.406	**0.681**

Table 7.10: $\widehat{A_{12}}$ 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 (p-value < 0.05)

7.7.5 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 §7.11. The top three best average and maximum APFD values of the table are highlighted in boldface. The results show that all the multi–objective combinations greatly improved CASA on both the average and maximum APFD values obtained. As in the previous experiment, 5 out of the 6 top best average 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. Analogously, 4 out of the 6 top best average 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 §7.12 shows the values of the $\widehat{A_{12}}$ effect size and statistical tests results on the comparison between mono and multi–objective combinations of functional and non–functional objectives. Values indicate a better performance of multi–objective prioritization compared to mono–objective prioritization with the exception of *Faults* where most cells were under 0.5. The overall preeminence, however, was observed in the combination of objectives *Changes* + *Dissimilarity* + *VCoverage* followed by *Changes* + *Connectivity* + *Faults*, with values over 0.5 in all cells and over 0.75 in 6 out of 7 columns.

Analogously, Table §7.13 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. Note that the combinations of functional objectives are not considered since we found no differences with the use of single functional objectives in Experiment 1. In contrast to the results shown in Table §7.12, $\widehat{A_{12}}$ values reveal no significant differences between multi–objective prioritization using non–functional objectives and multi–objective prioritization using both functional and non–functional objectives. More specifically, only five large differences were observed in favour of the multi–objective combinations of non–functional objectives. The preeminent combinations were again *Changes* + *Dissimilarity* + *VCoverage* and *Changes* + *Connectivity* + *Faults*, outperforming all other functional and non–functional objectives, either single or combined.

Objectives	**NSGA-II**			**CASA**	
	Avg	**Avg Max**	**Max**	**Avg**	**Max**
Changes + Connectivity	0.944	0.948	0.954	0.871	0.942
Changes + Dissimilarity	0.944	0.948	0.956	0.875	0.938
Changes + Pairwise	0.935	0.946	0.957	0.867	0.935
Changes + VCoverage	0.947	0.951	**0.960**	0.878	0.946
Connectivity + Faults	**0.948**	0.951	0.958	0.873	0.940
Connectivity + Size	0.944	0.947	0.959	0.877	0.942
Dissimilarity + Faults	**0.948**	0.951	0.958	0.867	0.940
Dissimilarity + Size	0.947	0.950	0.956	0.881	0.943
Faults + VCoverage	0.947	0.951	0.957	0.880	0.945
Faults + Pairwise	0.941	0.949	0.959	0.883	0.953
Pairwise + Size	0.937	0.949	0.954	0.877	0.937
Size + VCoverage	0.944	0.949	0.955	0.868	0.929
Changes + Connectivity + Dissimilarity	0.943	0.949	0.954	0.868	0.951
Changes + Connectivity + Faults	**0.948**	0.952	0.957	0.871	0.943
Changes + Connectivity + Pairwise	0.936	0.949	0.957	0.864	0.946
Changes + Connectivity + Size	0.945	0.951	0.959	0.863	0.944
Changes + Connectivity + VCoverage	0.942	0.948	0.958	0.869	0.924
Changes + Dissimilarity + Faults	0.947	0.951	0.957	0.866	0.936
Changes + Dissimilarity + Pairwise	0.938	0.951	0.957	0.874	0.942
Changes + Dissimilarity +Size	0.944	0.949	0.956	0.867	0.939
Changes + Dissimilarity + VCoverage	0.946	0.952	0.959	0.873	0.943
Changes + Faults + Pairwise	0.938	0.950	0.956	0.877	0.947
Changes + Faults + VCoverage	0.945	0.951	**0.960**	0.868	0.935
Changes + Pairwise + Size	0.934	0.949	0.957	0.879	0.947
Changes + Pairwise + VCoverage	0.938	0.949	0.956	0.873	0.936
Changes + Size + VCoverage	0.944	0.951	0.958	0.872	0.948
Connectivity + Dissimilarity + Faults	**0.948**	0.950	0.956	0.867	0.943
Connectivity + Dissimilarity + Size	0.947	0.950	**0.960**	0.876	0.939
Connectivity + Faults + Pairwise	0.941	0.951	0.957	0.870	0.931
Connectivity + Faults + Size	0.947	0.951	0.959	0.872	0.945
Connectivity + Faults + VCoverage	0.947	0.951	0.958	0.862	0.933
Connectivity + Pairwise + Size	0.942	0.951	0.958	0.882	0.946
Connectivity + Size + VCoverage	0.947	0.951	0.958	0.873	0.937
Dissimilarity + Faults + Pairwise	0.943	0.952	0.957	0.869	0.940
Dissimilarity + Faults + Size	**0.949**	0.951	0.958	0.868	0.940
Dissimilarity + Faults + VCoverage	0.945	0.950	0.955	0.873	0.938
Dissimilarity + Pairwise + Size	0.936	0.949	0.959	0.863	0.951
Dissimilarity + Size + VCoverage	**0.950**	0.957	0.957	0.872	0.949
Faults + Pairwise + Size	0.941	0.950	0.957	0.862	0.922
Faults + Pairwise + VCoverage	0.942	0.951	**0.960**	0.866	0.927
Faults + Size + VCoverage	0.945	0.950	0.957	0.879	0.934
Pairwise + Size + VCoverage	0.935	0.949	0.959	0.865	0.941
Average	**0.950**	**0.960**	**0.960**	**0.871**	**0.953**

Table 7.11: APFD values achieved by functional and non-functional prioritization objectives

Mixed Multi-Objective	Functional Objectives				Non-Functional Objectives		
	Connectivity	Dissimilarity	Pairwise	VCoverage	Changes	Faults	Size
Changes + Connectivity	**0.727**	0.560	**0.964**	0.645	0.624	**0.229**	0.499
Changes + Dissimilarity	**0.729**	0.610	**0.967**	**0.668**	**0.652**	**0.316**	0.547
Changes + Pairwise	0.644	0.516	**0.956**	0.596	0.573	**0.234**	0.440
Changes + VCoverage	**0.824**	**0.740**	**0.979**	**0.779**	**0.777**	0.470	**0.707**
Connectivity + Faults	**0.822**	**0.710**	**0.977**	**0.754**	**0.759**	0.413	**0.678**
Connectivity + Size	**0.664**	0.546	**0.958**	0.616	0.572	**0.256**	0.464
Dissimilarity + Faults	**0.821**	**0.719**	**0.977**	**0.759**	**0.766**	0.428	**0.691**
Dissimilarity + Size	**0.788**	**0.702**	**0.974**	**0.734**	**0.737**	0.419	**0.671**
Faults + Pairwise	**0.768**	0.623	**0.970**	**0.698**	**0.697**	**0.296**	0.564
Faults + VCoverage	**0.817**	**0.729**	**0.982**	**0.768**	**0.770**	0.426	**0.682**
Pairwise + Size	**0.746**	0.594	**0.964**	**0.670**	**0.654**	**0.278**	0.565
Size + VCoverage	**0.739**	0.642	**0.971**	**0.689**	**0.679**	0.365	0.592
Changes + Connectivity + Dissimilarity	**0.741**	0.612	**0.964**	**0.678**	**0.671**	**0.317**	0.581
Changes + Connectivity + Faults	**0.858**	**0.781**	**0.982**	**0.806**	**0.809**	0.504	**0.763**
Changes + Connectivity + Pairwise	**0.761**	0.624	**0.970**	**0.698**	**0.682**	**0.327**	0.577
Changes + Connectivity +Size	**0.808**	**0.691**	**0.974**	**0.748**	**0.739**	0.387	**0.650**
Changes + Connectivity + VCoverage	**0.723**	0.574	**0.968**	**0.665**	**0.653**	**0.239**	0.517
Changes + Dissimilarity + Faults	**0.806**	**0.697**	**0.973**	**0.746**	**0.751**	0.414	**0.672**
Changes + Dissimilarity + Pairwise	**0.809**	**0.727**	**0.978**	**0.756**	**0.764**	0.432	**0.693**
Changes + Dissimilarity + Size	**0.764**	0.635	**0.972**	**0.701**	**0.696**	**0.316**	0.564
Changes + Dissimilarity + VCoverage	**0.855**	**0.804**	**0.983**	**0.814**	**0.823**	0.518	**0.774**
Changes + Faults + Pairwise	**0.794**	**0.682**	**0.975**	**0.733**	**0.735**	0.358	**0.652**
Changes + Faults + VCoverage	**0.828**	**0.696**	**0.977**	**0.757**	**0.749**	0.388	**0.677**
Changes + Pairwise + Size	**0.751**	0.624	**0.968**	**0.693**	**0.677**	**0.316**	0.562
Changes + Pairwise + VCoverage	**0.759**	0.613	**0.968**	**0.693**	**0.683**	**0.274**	0.559
Changes + Size + VCoverage	**0.824**	**0.722**	**0.977**	**0.767**	**0.757**	0.406	**0.683**
Connectivity + +Dissimilarity + Faults	**0.801**	**0.676**	**0.977**	**0.730**	**0.738**	0.355	**0.652**
Connectivity + Dissimilarity + Size	**0.786**	**0.678**	**0.973**	**0.730**	**0.714**	0.380	0.617
Connectivity + Faults + Pairwise	**0.827**	**0.722**	**0.976**	**0.764**	**0.764**	0.403	**0.701**
Connectivity + Faults + Size	**0.842**	**0.733**	**0.981**	**0.777**	**0.777**	0.431	**0.701**
Connectivity + Faults + VCoverage	**0.830**	**0.718**	**0.978**	**0.769**	**0.772**	0.422	**0.709**
Connectivity + Pairwise + Size	**0.806**	**0.766**	**0.983**	**0.777**	**0.786**	0.498	**0.717**
Connectivity + Size + VCoverage	**0.816**	**0.728**	**0.979**	**0.767**	**0.753**	0.428	**0.688**
Dissimilarity + Faults + Pairwise	**0.850**	**0.759**	**0.984**	**0.786**	**0.803**	0.478	**0.757**
Dissimilarity + Faults + Size	**0.843**	**0.723**	**0.978**	**0.773**	**0.774**	0.436	**0.723**
Dissimilarity + Faults + VCoverage	**0.810**	**0.674**	**0.973**	**0.738**	**0.732**	**0.349**	**0.656**
Dissimilarity + Pairwise + Size	**0.727**	0.607	**0.966**	**0.673**	**0.651**	**0.294**	0.545
Dissimilarity + Size + VCoverage	**0.789**	**0.676**	**0.973**	**0.736**	**0.732**	0.397	0.627
Faults + Pairwise + Size	**0.784**	**0.648**	**0.972**	**0.724**	**0.721**	**0.338**	0.602
Faults + Pairwise + VCoverage	**0.826**	**0.714**	**0.977**	**0.766**	**0.757**	0.426	**0.704**
Faults + Size + VCoverage	**0.797**	**0.665**	**0.972**	**0.726**	**0.721**	**0.329**	0.622
Pairwise + Size + VCoverage	**0.751**	0.608	**0.970**	**0.683**	**0.669**	**0.263**	0.525

Table 7.12: $\widehat{A}_{12}$ 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 (p-value < 0.05)

Mixed Multi-Objective	**Non-Functional Multi-Objective**			
	Changes+Faults	Changes+Faults+Size	Changes+Size	Faults+Size
Changes + Connectivity	**0.262**	**0.324**	0.354	**0.334**
Changes + Dissimilarity	**0.351**	0.389	0.443	0.417
Changes + Pairwise	**0.256**	**0.287**	**0.318**	**0.299**
Changes + VCoverage	0.494	0.534	0.599	0.587
Connectivity + Faults	0.438	0.498	0.552	0.528
Connectivity + Size	**0.281**	**0.342**	0.362	**0.351**
Dissimilarity + Faults	0.459	0.487	0.558	0.542
Dissimilarity + Size	0.468	0.483	0.563	0.552
Pairwise + Faults	**0.328**	0.379	0.418	0.402
Faults + VCoverage	0.471	0.498	0.579	0.540
Pairwise + Size	**0.317**	0.385	0.414	0.417
Size + VCoverage	0.413	0.423	0.500	0.482
Changes + Connectivity + Dissimilarity	0.361	0.394	0.446	0.436
Changes + Connectivity + Faults	0.538	0.619	**0.671**	**0.649**
Changes + Connectivity + Pairwise	0.362	0.406	0.453	0.442
Changes + Connectivity + Size	0.418	0.466	0.520	0.501
Changes + Connectivity + VCoverage	**0.272**	**0.296**	**0.337**	**0.329**
Changes + Dissimilarity + Faults	0.452	0.502	0.562	0.547
Changes + Dissimilarity + Pairwise	0.484	0.518	0.588	0.565
Changes + Dissimilarity + Size	**0.341**	0.371	0.422	0.397
Changes + +Dissimilarity + VCoverage	0.554	0.604	**0.668**	**0.657**
Changes + Faults + Pairwise	0.398	0.418	0.492	0.483
Changes + Faults + VCoverage	0.422	0.489	0.533	0.522
Changes + Pairwise + Size	**0.344**	0.377	0.426	0.412
Changes + Pairwise + VCoverage	**0.309**	**0.351**	0.396	0.379
Changes + Size + VCoverage	0.451	0.532	0.576	0.559
Connectivity + Dissimilarity + Faults	0.414	0.436	0.511	0.487
Connectivity + Dissimilarity + Size	0.408	0.454	0.506	0.484
Connectivity + Faults + Pairwise	0.462	0.524	0.582	0.561
Connectivity + Faults + Size	0.470	0.539	0.601	0.573
Connectivity + Faults + VCoverage	0.462	0.516	0.570	0.554
Connectivity + Pairwise + Size	0.552	0.566	**0.652**	0.622
Connectivity + Size + VCoverage	0.477	0.531	0.589	0.563
Dissimilarity + Faults + Pairwise	0.534	0.570	**0.662**	0.634
Dissimilarity + Faults + Size	0.473	0.561	0.601	0.583
Dissimilarity + Faults + VCoverage	0.399	0.449	0.507	0.486
Dissimilarity + Pairwise + Size	**0.333**	0.368	0.414	0.404
Dissimilarity + Size + VCoverage	0.417	0.482	0.521	0.504
Faults + Pairwise + Size	0.359	0.386	0.441	0.422
Faults + Pairwise + VCoverage	0.463	0.535	0.579	0.562
Faults + Size + VCoverage	0.379	0.437	0.477	0.454
Pairwise + Size + VCoverage	**0.291**	**0.332**	0.362	**0.348**

Table 7.13: $\widehat{A_{12}}$ 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 dark grey (in favour of the column). Values in boldface reveal statistically significant differences (p-value < 0.05)

7.7.6 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–functional objectives, both single and combined. Table §7.14 shows the values of the $\widehat{A_{12}}$ effect size. Again, note that the functional multi–objectives were not considered since we found no differences with the use of single functional objectives in Experiment 1. The values indicate a clear preeminence of non–functional objectives, both single and combined, over functional objectives, being the objective *Changes* the only exception. As observed in Experiment 2 and 3, the objective *Faults* consistently show the largest differences followed by the combination of *Changes* + *Faults* and *Changes* + *Faults* + *Size*.

Functional Objectives	**Non-Functional Objectives**						
	Changes	Faults	Size	Changes+Faults	Changes+Faults+Size	Changes+Size	Faults+Size
Connectivity	0.420	**0.152**	**0.271**	**0.153**	**0.162**	**0.183**	**0.182**
Dissimilarity	0.543	**0.222**	0.429	**0.266**	**0.282**	**0.341**	**0.318**
Pairwise	**0.059**	**0.019**	**0.036**	**0.021**	**0.020**	**0.027**	**0.025**
VCoverage	0.469	**0.199**	**0.344**	**0.214**	**0.222**	**0.264**	**0.250**
Connectivity + Dissimilarity	0.542	**0.196**	0.396	**0.213**	**0.233**	**0.263**	**0.253**
Connectivity + Pairwise	0.451	**0.150**	**0.299**	**0.166**	**0.193**	**0.212**	**0.202**
Connectivity + VCoverage	0.502	**0.203**	0.374	**0.223**	**0.243**	**0.270**	**0.261**
Dissimilarity + Pairwise	0.550	**0.206**	0.433	**0.232**	**0.258**	**0.295**	**0.288**
Dissimilarity + VCoverage	0.486	**0.167**	**0.350**	**0.179**	**0.196**	**0.224**	**0.210**
Pairwise + VCoverage	0.418	**0.094**	**0.247**	**0.110**	**0.122**	**0.142**	**0.129**
Connectivity + Dissimilarity + Pairwise	0.494	**0.197**	0.367	**0.230**	**0.233**	**0.273**	**0.258**
Connectivity + Dissimilarity + VCoverage	0.457	**0.166**	**0.331**	**0.179**	**0.214**	**0.224**	**0.226**
Connectivity + Pairwise + VCoverage	0.572	**0.227**	0.425	**0.251**	**0.261**	**0.302**	**0.278**
Dissimilarity + Pairwise + VCoverage	0.396	**0.072**	**0.237**	**0.083**	**0.112**	**0.118**	**0.109**

Table 7.14: $\widehat{A_{12}}$ 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 (p-value < 0.05)

7.7.7 Discussion

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

RQ1: Mono vs. multi–objective prioritization using functional objectives. The results of experiment 1 revealed no significant differences between mono and multi–objective prioritization using functional objectives. The only exception was found in the *Pairwise* objective function which consistently achieved the worse rate of fault detection, which suggest that pairwise coverage is not an effective prioritization criterion. In the light of these results, RQ1 is answered as follows:

Mono and multi–objective prioritization driven by functional objectives have similar effectiveness 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. A clear exception was found in the objective function *Faults* which outperformed all other combinations. 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 two main conclusions on the multi–objective combination of functional and non–functional objectives. First, the results showed that the multi–objective prioritization using functional and non–functional objectives is consistently better than mono–objective prioritization. Again, the only exception was found in the function *Faults*. This could be explained, as detailed above, by the use of two consecutive versions of the framework and thus this result is not fully conclusive. Second, results revealed no significant differences between multi–objective prioritization driven by functional and non–functional objectives and multi–objective prioritization using non–functional objectives only. Interestingly, however, the best results were obtained by mixed combinations of three functional and non–functional objectives, i.e. *Changes + Dissimilarity + VCoverage* and *Changes + Connectivity + 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, but equally well as 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 superiority 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 [118]. 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.

7.8 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 work, we used standard parameter values for the NSGA-II algorithm [28]. 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.

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 [118]. 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.

7.9 Summary

This chapter 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. Results also showed that multi–objective prioritization using non–functional objectives performs better than mono–objective prioritization. Furthermore, we found that multi–objective prioritization, where at least one of the objectives is non–functional, usually results in faults being detected faster. To the best of our knowledge, this is the first work on multi–objective test case prioritization in HCS 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 approach have been integrated into *smarTest* (see Section §A), a Drupal test prioritization module developed by the authors and recently presented at the International Drupal Conference [117] with very positive feedback from the community.

Part of the results described in this chapter was submitted to the Journal of System and Software. 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).

Part IV

Final remarks

CONCLUSIONS AND FUTURE WORK

There will come a time when you believe everything is finished.
That will be the beginning.

Louis Dearborn, 1908-1988, American writer,

8.1 CONCLUSIONS

The development of HCSs provides clear benefits for the production process but significant challenges for the validation. The high number of configurations that can be potentially derived from an HCS may lead to thousand or even millions, e.g. the Drupal HCS has more than 2 billions of configurations. This makes that testing every single HCS configuration is unfeasible. As a result, a number of testing techniques to reduce the space of testing in HCSs are rapidly proliferating. However, even after reducing the test space, the number of configurations under test may still be large and expensive to run. Thus, test case prioritization techniques appeared to identify the efficient ordering of execution of tests to maximize certain performance goals, such as accelerating the detection of faults. Nevertheless, current approaches for test case prioritization in HCSs are scarce and suffer from some limitations. First, most of them are a combination of sampling (usually using CIT) and prioritization (using weights) to generate the test cases, which is not a pure prioritization. Second, the prioritization is mainly driven by a single objective or a combination of several objectives into a single function by assigning them weights proportional to their relative importance, which neglects the potential benefits of combining multiple equally important criteria to guide the detection of faults. Third, generally, prioritization testing techniques in HCS were driven by functional objectives without taking advantage of all available information about the system under test to guide the testing. And fourth, the evaluation of HCS testing approaches are basically conducted using synthetic data rather than industry-strength case studies, which weakens the value of research proposals.

In this dissertation, we have addressed aforementioned limitations by proposing new prioritization objectives based on functional and non-functional properties, new testing techniques focused on single-objective and multi–objective prioritization, new industry-strength case studies to evaluate the testing approaches and a test case prioritization tool, SmarTest, integrated into a real open-source HCS. Also, we analyzed 63 combinations of up to three prioritization objectives to find the best combinations for accelerating the detection of faults in an industry HCS tool. Among other results, we found that prioritization driven by non–functional objectives, such as the number of faults found in a previous version of the system, accelerates the detection of bugs more effectively than functional prioritization objectives. Results also showed that, in general, multi–objective prioritization using non–functional objectives performs better than mono–objective prioritization. Furthermore, we found that multi–objective prioritization, where at least one of the objectives is non–functional, usually results in faults being detected faster for our case study. The results also showed that non–functional objectives effectively accelerate the detection of faults of both random and pairwise-based HCS test suites. This suggests that our approach could be a nice complement for current techniques for test case selection. To the best of our knowledge, we are pioneers in the support of multi–objective test case prioritization in HCSs using industry-strength data and comparing the effectiveness of functional and non-functional prioritization objectives to accelerate the detection of bugs.

8.2 Discussion and limitations

We next discuss some of the decisions that we have made in this dissertation highlighting its main limitations.

Parameter settings. We adapted the NSGA-II algorithm to solve the multi-objective test case prioritization problem in HCSs. Inadequate parameter setting is a common internal validity threat. However, we used standard parameter values for the NSGA-II algorithm. Furthermore, to consider the effect of stochasticity, the algorithm was executed multiple times with each combination of objective functions and their results analyzed using statistical tests.

Drupal Case study. The reengineering process could have influenced the final feature model of Drupal and therefore the evaluation results. To alleviate this threat, we followed a systematic approach and mapped Drupal modules to features. This is in line with the Drupal documentation, which defines an enabled module as a feature providing certain functionality to the system. This also fits in the definition of feature given by Batory, who defines a feature as an increment in product functionality. In turn, submodules were mapped to subfeatures since they provide extra functionality to its parent module and they have no meaning without it. Finally, we used the dependencies defined in the information file of each Drupal module to model CTCs. Regarding the data collected about integration faults in Drupal, it is possible that we missed some integration faults, and conversely, we could have misclassified some individual faults as integration faults. To mitigate this threat as much as possible, we manually checked each candidate integration fault trying to discard those that were clearly not caused by the interaction among features. This was an extremely time-consuming and challenging task that required a good knowledge of the framework. We may emphasize that the candidate author has more than one year of experience in industry as a Drupal developer. Also, as a further validation, the work was discussed with two members of the Drupal core team who approved the followed approach and gave us helpful feedback. Besides, the faults in additional modules are not related to a specific Drupal subversion, i.e. they are reported as faults in Drupal v7.x. Therefore, we assumed that the faults in those modules equally affected the versions 7.22 and 7.23 of Drupal. This is a realistic approach since it is common in open-source projects that unfixed faults affect several versions of the system.

Correlation study of prioritization objectives. As the correlation study of prioritization objectives is based on the Drupal framework, there are some limitations since the Drupal feature model and their attributes were manually mined from different sources and therefore they could slightly differ from their real shape. Also, the number of faults in Drupal v7.22 remained in Drupal v7.23, which may introduce a bias in the prioritization based on historical of faults. 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.

Generalizability of results. Finally, the statistical and prioritization results reported in part of this dissertation 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 objectives in a multi–objective problem as good drivers for test case prioritization in open–source HCSs as the Drupal framework.

8.3 Future work

We identify a number of motivating topics to be explored in our future research, namely:

Many-objective test case prioritization. Due to restrictions on the type of the algorithm chosen, we have explored the combination of up to three objectives, therefore, 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.

Development of other case studies. The Drupal case study presented in this dissertation provides researchers and practitioners with helpful information about the distribution of faults and test cases in a real HCS and it is a valuable assets to evaluate variability testing techniques in realistic settings. This work encourages us to keep exploring on the use of industry-strength non–functional attributes in the context of HCS testing and to develop similar case studies that allow us and another researchers to study the generalizability of the research conclusions.

Tool support. Part of the results of this dissertation have been integrated into SmarTest (see Section §A) with very positive feedback from the Durpal Community. We have included our test case prioritization techniques driven by a mono-objective perspective obtaining efficient results in revealing bugs earlier in Drupal. We plan to integrate the multi-objective prioritization techniques proposed by developing our genetic algorithm in SmarTest. We are confident that our test case multi-objective approach could further improve the testing in Drupal.

Part V

Appendices

SMARTEST

A picture is worth a thousand words.
An interface is worth a thousand pictures.

Ben Shneiderman, 1947, Usamerican computer scientist,

In this dissertation, we have motivated the need for automated support for improving the process of testing in HCSs. In particular, we have showed how the prioritization of the test cases execution can reduce the effort of testing by revealing faults faster and enabling software engineers to begin debugging and correcting faults earlier. However, most of tools resulting from research are exploited in the academic community rather than in industry. In order to go a step further to get a tool useful for industry realm, we have integrated our testing techniques presented in this dissertation in Drupal, a well-known web content management framework as a testing module called SmarTest. This appendix describes the SmarTest tool. In Section §A.1, we describe the development context of SmarTest. Section §A.2 provides a general overview of the SmarTest. Finally, a summary of the appendix is presented in Section §A.3.

A.1 SmarTest context

throughout this document, we focused on defining testing solutions for families of configurations. Now, we asked us why not finish the complete process of testing of families of configurations, i.e., why not continue testing the individual configurations derived from a family of configurations and apply our prioritization techniques to them? And, why not to do it in a real environment?

In order to carry out this idea, we decided to work with a real configuration of the Drupal web content management framework [18, 140]. Drupal can be used to build a variety of web sites including internet portals, e-commerce applications and online

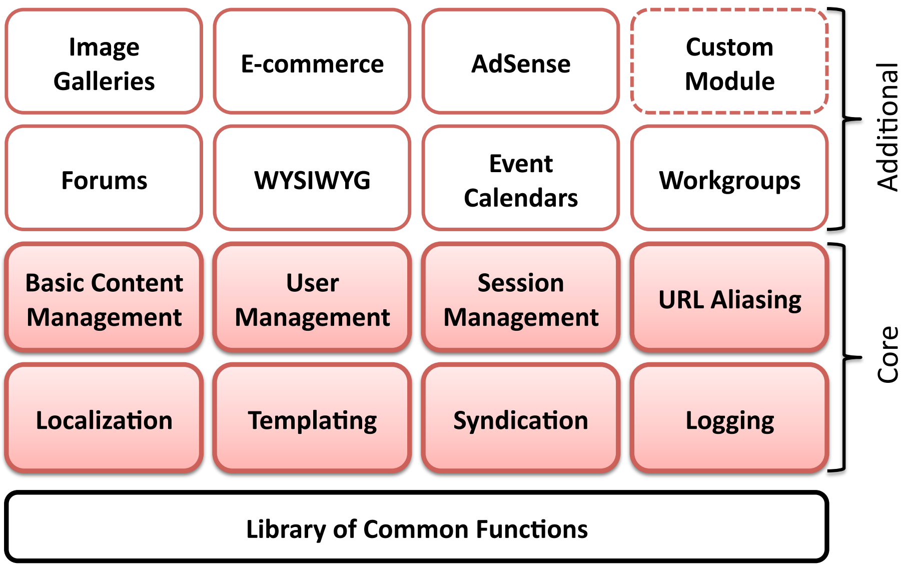


Figure A.1: Several Drupal core and additional modules. Taken from [140]

newspapers. It is composed of modules. A module is a collection of functions that provide certain functionality to the system. As we described in Section §6.2, the modules in Drupal can be classified into core modules and additional modules. The core modules are approved by the core developers of the Drupal community and they are included by default in the basic installation of Drupal framework. They include code that allows the system to bootstrap when it receives a request, a library of common functions frequently used with Drupal, and modules that provide basic functionality like user management and templating. Additional modules can be divided into contributed modules and custom modules. Contributed modules are written by the Drupal community and shared under the same GNU Public License (GPL) as Drupal. Custom modules are those created by external contributors. Figure §A.1 illustrates some popular core and additional Drupal modules.

In this context, we analyzed the Drupal testing system. Drupal has a core module called SimpleTest that allows defining and executing automated tests. SimpleTest reads the drupal tests defined in the system and shows the list of tests grouped by categories such as tests for views or tests for users. Then, the module enables the selection or deselection of tests to be executed. Although SimpleTest has a powerful and easy to use functionality for testing, we think that this module could be improved by applying our testing solutions proposed in this dissertation as an extension of this core module. Thus, SmarTest arises.

A.2 SmarTest overview

SmarTest is a testing module for Drupal that includes part of the results obtained in this dissertation. It is written in PHP and it is integrated in the Drupal sandbox project[1] distributed as GNU General Public License, version 2. SmarTest supports the analysis of the system providing useful information to guide the testers and it allows applying different prioritization testing techniques to reveals bugs faster. Current version of SmarTest provides the following features:

- **Customizable dashboard with information at run-time**. The main view of SmarTest is a dashboard that allows showing actual information about the Drupal system such as the size of the modules (in terms of lines of code), the number of tests per module that passed and failed in last tests executions or the complexity of the modules. All these data have been showed to be closely related with the propensity to faults of Drupal modules. For example, in Section §6.6, we performed a correlation study that revealed a strong positive correlation between the size of the Drupal modules and the number of faults in the modules. This means that the number of lines of code could be used as a good estimation of their fault propensity. In addition to this information, the dashboard may display the percentage of code coverage that cover the tests, the modules with less test coverage, the modules with more failures detected in last executions, the percentage of test passed and failed in last test executions or even the time taken by the last execution. In addition to this useful functionality for testers, the dashboard is totally customizable and configurable. This means that we can include new widgets of information (e.g. chart showing the relation of lines of code in the modules and their ciclomatic complexity) and in the format that we need (e.g. tag clouds, column graphs or row graphs). Figure §A.2 illustrates the screenshot of a dashboard created in SmarTest.
- **Prioritization of tests**. SmarTest allows selecting different types of test prioritization criteria based on real data of the Drupal system. Thus, we can order the execution of tests based on the complexity of the tests, based on the size of the modules, based on the number of commits (taken from the Drupal project Github), based on the code coverage, based on the number of tests that failed in last executions, etc. We include these prioritization criteria according to our statistic correlation study presented in Section §6.6 that showed the correlation

[1]https://www.drupal.org/sandbox/annasan/2503695

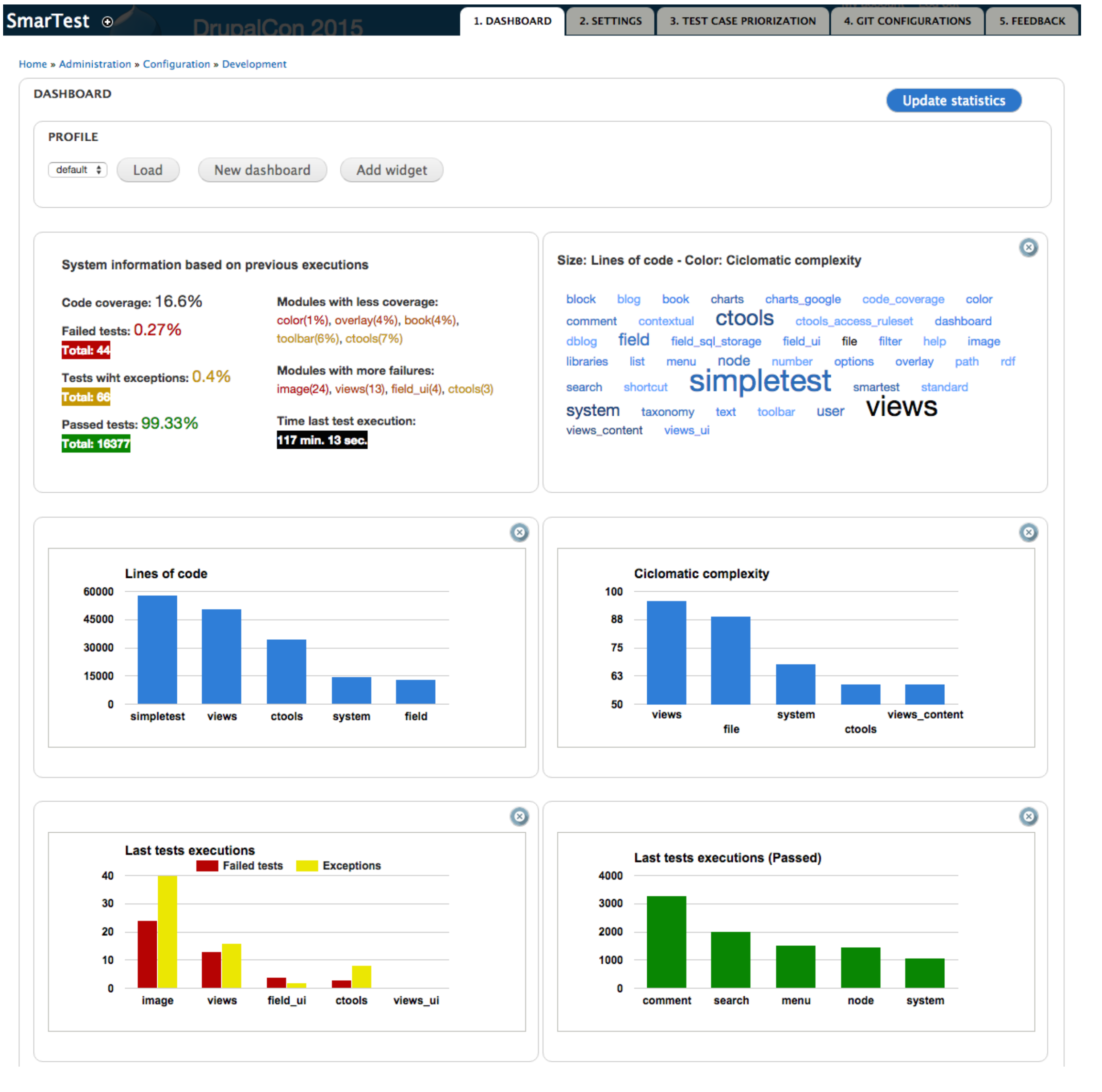
of these properties of the code with its error-prone. Also, we can indicate the time that we want tests are running (given in minutes or hours). Once we have chosen the prioritization criteria and the execution time, SmarTest displays all of the tests of the system ordered by that criterion, showing the value for that criterion of each group of tests and the time taken in the last execution. Then, the system allows us to select all the tests or some of them to be run in the established order. As an example, consider the view in Figure §A.3, the tester have chosen the prioritization criterion driven by Git changes to guide the testing. Thus, the modules in Drupal with higher number of Git changes will have higher priority to be tested as illustrates the ordering of tests in the figure. It is remarkable that when a test finishes, SmarTest returns a detailed report of the results immediately (see Figure §A.4). This enables a continuous feedback of the progress of testing in real time that reduces time to the software engineers to start fixing the bugs. This is relevant since running all the tests in Drupal can take many hours or days and the current system of testing of Drupal (SimpleTest module) does not return the results of the tests after all the them have finished.

The SmarTest module has been presented in two Drupal conferences, the National Conference DrupalCampSpain [119] and the Europe Conference DrupalConEurope [117] where professionals of Drupal meet annually, achieving a positive acceptance. The module is freely distributed under GNU GPL v2 license and can be downloaded from the Drupal sandbox project[2], where it is waiting to be accepted as a contributed Drupal module. Also, SmarTest has a web site (see Figure §A.5)

A.3 Summary

In this appendix, we have presented SmarTest, a testing module for Drupal that integrates some of the contributions implemented in this dissertation. Its main goal is to make our work accessible and useful to both the academic and the industrial community. Furthermore, we have distributed SmarTest under the GNU GPL v2 license recommended by Drupal to allow to other Drupal developers take part in the development and improvement of the tool. In fact, some Drupal software engineers are currently working on SmarTest as can be seen in the SmarTest sandbox project in Drupal: https://www.drupal.org/sandbox/annasan/2503695.

[2]https://www.drupal.org/sandbox/annasan/2503695



System information based on previous executions

Code coverage: 16.6%

Failed tests: 0.27%
Total: 44

Tests wiht exceptions: 0.4%
Total: 66

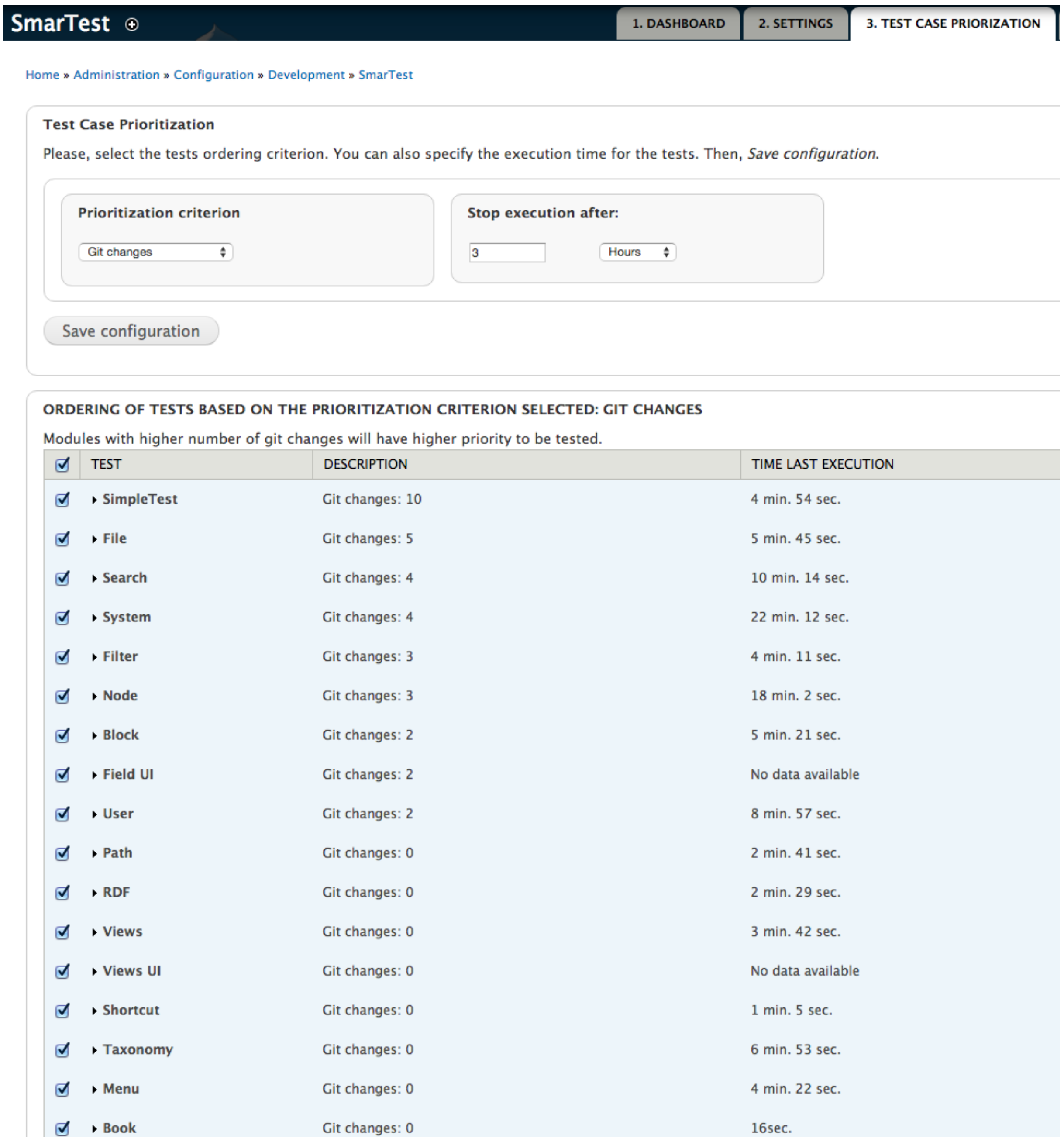
Passed tests: 99.33%
Total: 16377

Modules with less coverage:
color(1%), overlay(4%), book(4%), toolbar(6%), ctools(7%)

Modules with more failures:
image(24), views(13), field_ui(4), ctools(3)

Time last test execution:
117 min. 13 sec.

Figure A.2: SmarTest's dashboard view


SmarTest
1. DASHBOARD
2. SETTINGS
3. TEST CASE PRIORIZATION
Home » Administration » Configuration » Development » SmarTest

Test Case Prioritization

Please, select the tests ordering criterion. You can also specify the execution time for the tests. Then, *Save configuration*.

Prioritization criterion: Git changes

Stop execution after: 3 Hours

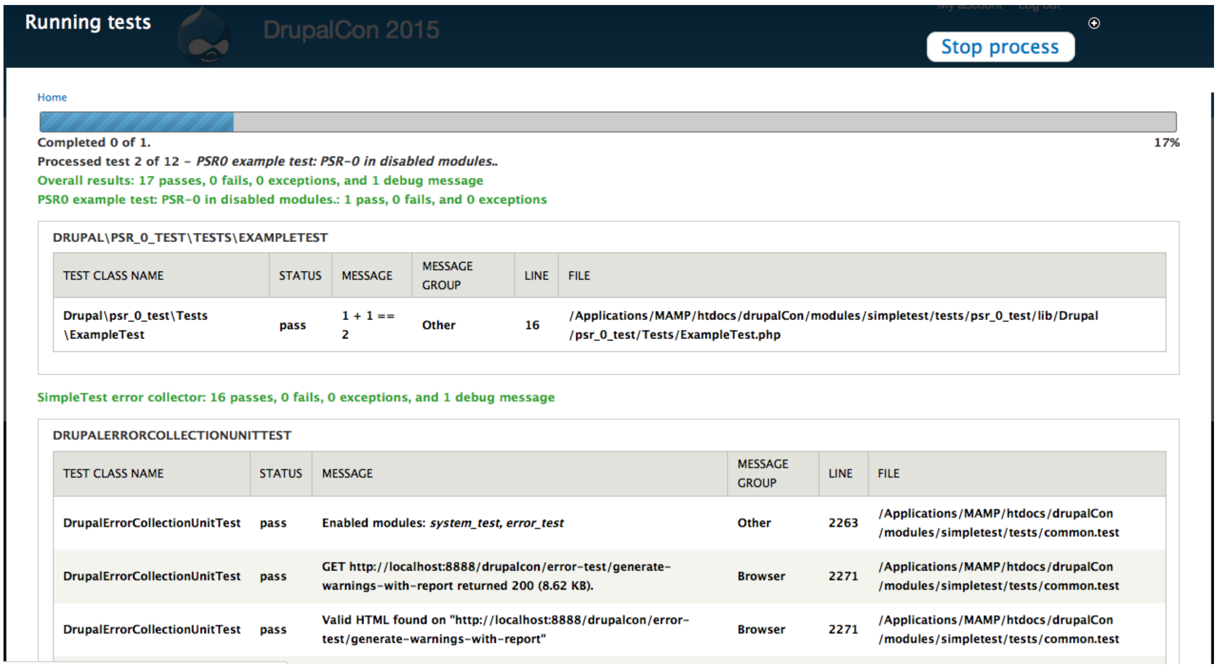
Save configuration

ORDERING OF TESTS BASED ON THE PRIORITIZATION CRITERION SELECTED: GIT CHANGES

Modules with higher number of git changes will have higher priority to be tested.

✓	TEST	DESCRIPTION	TIME LAST EXECUTION
✓	SimpleTest	Git changes: 10	4 min. 54 sec.
✓	File	Git changes: 5	5 min. 45 sec.
✓	Search	Git changes: 4	10 min. 14 sec.
✓	System	Git changes: 4	22 min. 12 sec.
✓	Filter	Git changes: 3	4 min. 11 sec.
✓	Node	Git changes: 3	18 min. 2 sec.
✓	Block	Git changes: 2	5 min. 21 sec.
✓	Field UI	Git changes: 2	No data available
✓	User	Git changes: 2	8 min. 57 sec.
✓	Path	Git changes: 0	2 min. 41 sec.
✓	RDF	Git changes: 0	2 min. 29 sec.
✓	Views	Git changes: 0	3 min. 42 sec.
✓	Views UI	Git changes: 0	No data available
✓	Shortcut	Git changes: 0	1 min. 5 sec.
✓	Taxonomy	Git changes: 0	6 min. 53 sec.
✓	Menu	Git changes: 0	4 min. 22 sec.
✓	Book	Git changes: 0	16sec.

Figure A.3: SmarTest's test case prioritization view



Running tests

Stop process

Home

Completed 0 of 1. 17%

Processed test 2 of 12 – *PSR0 example test: PSR-0 in disabled modules.*

Overall results: 17 passes, 0 fails, 0 exceptions, and 1 debug message

PSR0 example test: PSR-0 in disabled modules.: 1 pass, 0 fails, and 0 exceptions

DRUPAL\PSR_0_TEST\TESTS\EXAMPLETEST

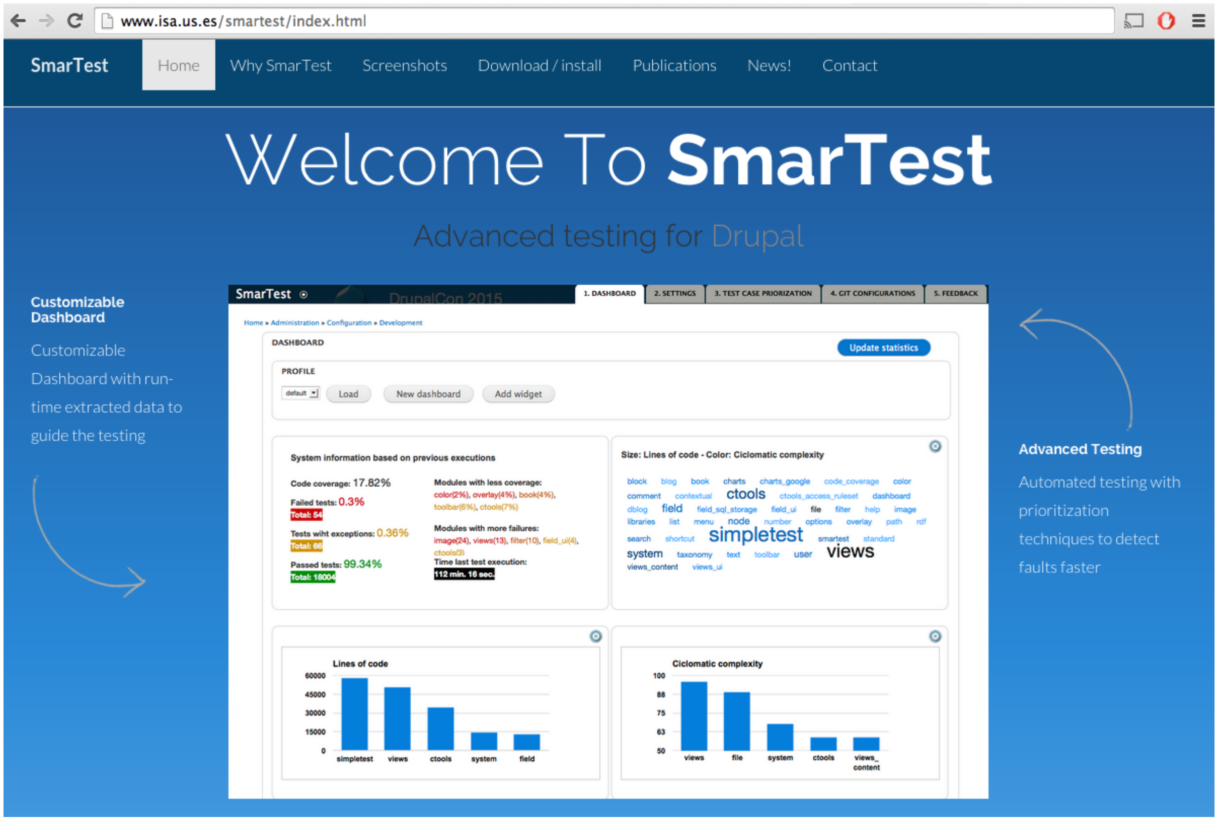
TEST CLASS NAME	STATUS	MESSAGE	MESSAGE GROUP	LINE	FILE
Drupal\psr_0_test\Tests \ExampleTest	pass	1 + 1 == 2	Other	16	/Applications/MAMP/htdocs/drupalCon/modules/simpletest/tests/psr_0_test/lib/Drupal /psr_0_test/Tests/ExampleTest.php

SimpleTest error collector: 16 passes, 0 fails, 0 exceptions, and 1 debug message

DRUPALERRORCOLLECTIONUNITTEST

TEST CLASS NAME	STATUS	MESSAGE	MESSAGE GROUP	LINE	FILE
DrupalErrorCollectionUnitTest	pass	Enabled modules: *system_test, error_test*	Other	2263	/Applications/MAMP/htdocs/drupalCon /modules/simpletest/tests/common.test
DrupalErrorCollectionUnitTest	pass	GET http://localhost:8888/drupalcon/error-test/generate-warnings-with-report returned 200 (8.62 KB).	Browser	2271	/Applications/MAMP/htdocs/drupalCon /modules/simpletest/tests/common.test
DrupalErrorCollectionUnitTest	pass	Valid HTML found on "http://localhost:8888/drupalcon/error-test/generate-warnings-with-report"	Browser	2271	/Applications/MAMP/htdocs/drupalCon /modules/simpletest/tests/common.test

Figure A.4: SmarTest's test case execution view



www.isa.us.es/smartest/index.html

SmarTest | Home | Why SmarTest | Screenshots | Download / install | Publications | News! | Contact

Welcome To **SmarTest**

Advanced testing for Drupal

Customizable Dashboard

Customizable Dashboard with run-time extracted data to guide the testing

Advanced Testing

Automated testing with prioritization techniques to detect faults faster

Figure A.5: SmarTest web site (http://www.isa.us.es/smartest/index.html)

Drupal information

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

Table B.1: Drupal modules included in the case study

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

Fault	Feature (ID from Table §B.1)																																													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
F70						X											X																														
F71																	X		X																									X			
F72																	X										X																				
F73						X											X																														
F74															X		X																														
F75																	X		X																									X			
F76																	X		X								X																				
F77						X											X																														
F78						X												X																													
F79																		X	X																												
F80																			X																									X			
F81																			X																									X			
F82																			X				X																					X			
F83			X												X				X																									X			
F84													X						X																												
F85																			X																			X									
F86																	X		X																									X			
F86																			X																									X			
F88																			X																									X			
F89																			X																												X
F90																				X																								X			
F91																	X				X																										
F92																							X	X																							
F93													X											X																							
F94																								X																				X			
F95														X										X																							
F96														X										X																							
F97												X				X								X																							
F98		X																						X																							
F99														X										X																							
F100																										X					X																
F101																											X												X								
F102														X														X																			
F103																	X		X										X																		
F104														X																	X																
F105														X																	X																
F106						X																									X																
F107																															X	X															
F108																															X								X								
F109																		X					X								X																
F110																															X								X								
F111																			X												X																
F112																	X														X																
F113																							X	X								X															
F114																															X	X															
F115																																		X				X									
F116																																			X				X								
F117																	X																		X												
F118																																			X		X										
F119																	X																		X												
F120																																				X								X			
F121																																	X			X											
F122													X																							X											
F123	X														X																					X											
F124																																	X			X											
F125																																	X			X											
F126																																				X								X			
F127																																	X				X										
F128																							X														X										
F129																							X														X										
F130																																	X			X		X									
F131													X																									X									
F132																	X																					X	X								
F133					X																															X		X									
F134					X																																	X									
F135													X																										X								
F136																															X								X								
F137																															X								X								
F138																																							X								X
F139																															X	X							X								

Fault	Feature (ID from Table §B.1)																																													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
F140																							X																X								
F141							X																																X								
F142													X																										X								
F143																															X	X							X								
F144															X																								X								
F145				X															X																							X					
F146																			X																									X			
F147																																												X			X
F148																	X																											X			
F149	X														X																													X			
F150																			X																									X			
F151																																						X						X			
F152																																X													X		
F153																			X																											X	
F154																																												X		X	
F155														X																																	X
F156													X																																		X
F157														X																																	X
F158														X																																	X
F159											X																																				X
F160														X																																	X

Table B.2: Integration faults in Drupal v7.23

Acronyms

HCS	Highly-Configurable Software System.
SPL	Software Product Line.
FM	Feature Model.
AFM	Attributed Feature Model.
CIT	Combinatorial Interaction Testing.
APFD	Average Percentage of Faults Detected.
SPLOT	Software Product Lines Online Tools.
SXFM	Simple XML Feature Model format.
XML	eXtensible Mark-up Language.
IT	Information Technology.
NSGA-II	Non-dominated Sorting Genetic Algorithm-II.
CTC	Cross-Tree Constraint.

Bibliography

[1] Debian Wheezy. http://www.debian.org/releases/wheezy/, accesed February 2016. (page 47).

[2] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information Sofware Technology*, 51(6):957–976, June 2009. ISSN 0950-5849. doi: 10.1016/j.infsof.2008.12.005. URL http://dx.doi.org/10.1016/j.infsof.2008.12.005. (page 44).

[3] ahead. AHEAD Tool Suite. http://www.cs.utexas.edu/users/schwartz/ATS.html, accessed February 2016. (page 57).

[4] A. A. Ahmed, M. Shaheen, and E. Kosba. Software testing suite prioritization using multi-criteria fitness function. In *22nd International Conference on Computer Theory and Applications (ICCTA), 2012*, pages 160–166, Oct 2012. doi: 10.1109/ICCTA.2012.6523563. (pages 44, 45).

[5] M. Al-Hajjaji, T. Thum, J. Meinicke, M. Lochau, and G. Saake. Similarity-based prioritization in software product-line testing. In *Software Product Line Conference*, pages 197–206, 2014. doi: 10.1145/2648511.2648532. (pages 6, 8, 31, 33, 38, 48, 52, 77, 106, 107).

[6] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014. (page 128).

[7] C. Artho, K. Suzaki, R. D. Cosmo, R. Treinen, and S. Zacchiroli. Why do software packages conflict? In *Conference on Mining Software Repositories*, 2012. (pages 89, 90).

[8] E. Bagheri and D. Gasevic. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Control*, 2011. (pages 58, 59, 61, 84, 95, 121, 123).

[9] E. Bagheri, F. Ensan, and D. Gasevic. Grammar-based test generation for software product line feature models. In *Conference of the Centre for Advanced Studies on Collaborative Research*, 2012. (pages 50, 57, 66, 73).

[10] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference, LNCS 3714*, pages 7–20, 2005. (pages 28, 102).

[11] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, 49(12):45–47, Dec. 2006. ISSN 0001-0782. doi: 10.1145/1183236.1183264. URL http://doi.acm.org/10.1145/1183236.1183264. (page 28).

[12] D. Benavides. *On the Automated Analyisis of Software Product Lines using Feature Models. A Framework for Developing Automated Tool Support*. PhD thesis, University of Seville, 2007. (page 28).

[13] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated reasoning on feature models. In O. Pastor and J. Falco e Cunha, editors, *International Conference on Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-26095-0. doi: 10.1007/11431855_34. URL http://dx.doi.org/10.1007/11431855_34. (page 28).

[14] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analyses of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010. doi: 10.1016/j.is.2010.01.001. (pages 4, 56, 57, 58, 81, 108).

[15] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, Dec 2013. ISSN 0098-5589. doi: 10.1109/TSE.2013.34. (pages 76, 84).

[16] S. Bergmann. pholoc. http://github.com/sebastianbergmann/phploc, accessed March 2014. URL http://github.com/sebastianbergmann/phploc. (page 86).

[17] R. C. Bryce, S. Sampath, and A. M. Memon. Developing a single model and test prioritization strategies for event-driven software. *IEEE Transactions on Software Engineering*, 37(1):48–64, Jan 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.12. (pages 33, 40).

[18] D. Buytaert. Drupal Framework. http://www.drupal.org, accessed in February 2016. (pages 12, 14, 15, 76, 78, 86, 87, 111, 149).

[19] C. Catal and D. Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 2012. (pages 34, 35, 37, 38, 39, 45, 56).

[20] X. Chen, Q. Gu, X. Wang, A. Li, and D. Chen. A hybrid approach to build prioritized pairwise interaction test suites. In *International Conference on Computational Intelligence and Software Engineering*, pages 1–4, Dec 2009. doi: 10.1109/CISE.2009.5365886. (pages 33, 41).

[21] D. M. Cohen, S. Dalal, M. Fredman, and G. Patton. The aetg system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, Jul 1997. ISSN 0098-5589. doi: 10.1109/32.605761. (page 49).

[22] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *Transactions on software engineering*, 34(5):633–650, 2008. (pages 3, 21, 106).

[23] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In R. Nord, editor, *Software Product Lines Conference*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22918-6. doi: 10.1007/978-3-540-28630-1_17. URL http://dx.doi.org/10.1007/978-3-540-28630-1_17. (pages 26, 27).

[24] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005. (pages 26, 27, 28).

[25] D. R. W. D. Richard Kuhn and A. M. Gallo. Software fault interactions and implications for software testing. *Transactions on Software Engineering*, 30, 2004. (page 67).

[26] P. A. da Mota S. Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira. A systematic mapping study of software product lines testing. *Information & Software Technology*, 53(5):407–423, 2011. (page 49).

[27] K. Deb and K. Deb. Multi-objective optimization. In E. K. Burke and G. Kendall, editors, *Search Methodologies*, pages 403–449. Springer US, 2014. ISBN 978-1-4614-6939-1. doi: 10.1007/978-1-4614-6940-7_15. URL http://dx.doi.org/10.1007/978-1-4614-6940-7_15. (page 110).

[28] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2): 182–197, Apr 2002. ISSN 1089-778X. doi: 10.1109/4235.996017. (pages 110, 126, 139).

[29] X. Devroey, G. Perrouin, M. Cordy, P. Schobbens, A. Legay, and P. Heymans. Towards statistical prioritization for software product lines testing. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive (VAMOS)*, 2014. (pages 8, 33, 39, 51, 106).

[30] X. Devroey, G. Perrouin, and P. Schobbens. Abstract test case generation for behavioural testing of software product lines. In *Software Product Line Conference*, volume 2, pages 86–93. ACM, 2014. doi: 10.1145/2647908.2655971. (pages 6, 48, 106).

[31] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *21st IEEE International Conference on Software Maintenance*, number 411-420, 2005. (pages 33, 40).

[32] I. do Carmo Machado, J. D. McGregor, Y. C. Cavalcanti, and E. S. de Almeida. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, 56(10):1183 – 1199, 2014. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2014.04.002. (page 49).

[33] DrupalJITmodule. Drupal JIT module. http://www.drupal.org/project/thejit, accessed February 2016. (page 84).

[34] J. J. Durillo and A. J. Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771, 2011. (page 126).

[35] M. B. C. M. B. Dwyer and J. Shi. Coverage and adequacy in software product line testing. In *ISSTA 2006 workshop on Role of software architecture for testing and analysis, ser. ROSATEA*, 2006. (page 50).

[36] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1050-7. URL http://0-dl.acm.org.fama.us.es/citation.cfm?id=381473.381508. (page 46).

[37] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *Transactions on Software Engineering*, 2002. (pages 31, 60).

[38] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software Testing, Verification and Reliability*, 13(2):65–83, 2003. ISSN 1099-1689. doi: 10.1002/stvr.263. URL `http://dx.doi.org/10.1002/stvr.263`. (page 6).

[39] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 2004. (pages 45, 115).

[40] S. engineering institute: Hall of Fame. Available at: www.splc.net/fame.html. Software engineering institute: Hall of Fame. `www.splc.net/fame.html`, accessed February 2016. (pages 4, 47).

[41] E. Engstrm and P. Runeson. Software product line testing a systematic mapping study. *Information and Software Technology*, 53(1):2 – 13, 2011. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2010.05.011. URL `http://www.sciencedirect.com/science/article/pii/S0950584910001709`. (pages 47, 49).

[42] A. Ensan, E. Bagheri, M. Asadi, D. Gasevic, and Y. Biletskiy. Goal-oriented test case selection and prioritization for product line feature models. In *Conference Information Technology:New Generations*, 2011. (pages 8, 33, 37, 51, 56, 106).

[43] F. Ensan, E. Bagheri, and D. Gasevic. Evolutionary search-based test generation for software product line feature models. In *Conference on Advanced Information Systems Engineering (CAiSE'12)*, 2012. (pages 50, 57, 59, 62, 66, 72, 73, 77, 94, 96, 98, 123).

[44] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *International Symposium on Software Testing and Analysis*, 2015. (pages 33, 34, 40, 43, 44, 45).

[45] L. Etxeberria, G. Sagardui, and L. Belategi. Modelling variation in quality attributes. In *First International Workshop on Variability Modelling of Softwareintensive Systems*, 2007. (page 28).

[46] fama. Prestashop. https://www.prestashop.com/, accessed January 2016. (page 34).

[47] J. Ferrer, F. Chicano, and E. Alba. Evolutionary algorithms for the multi-objective test data generation problem. *Software Practice and Experience*, 2012. (page 122).

[48] J. Ferrer, P. Kruse, F. Chicano, and E. Alba. Evolutionary algorithm for prioritized pairwise test data generation. In *Genetic and Evolutionary Computetion Conference (GECCO)*, 2012. (pages 33, 36, 39, 43).

[49] J. A. Galindo, D. Benavides, and S. Segura. Debian packages repositories as software product line models. towards automated analysis. In *ACOTA*, 2010. (page 76).

[50] J. García-Galán, O. Rana, P. Trinidad, and A. Ruiz-Cortés. Migrating to the cloud: a software product line based analysis. In *3rd International Conference on Cloud Computing and Services Science*, pages 416–426, 2013. (page 4).

[51] B. Garvin, M. Cohen, and M. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011. ISSN 1382-3256. doi: 10.1007/s10664-010-9135-7. URL http://dx.doi.org/10.1007/s10664-010-9135-7. (pages 119, 127).

[52] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *International Symposium on Search Based Software Engineering*, pages 13–22, 2009. (pages 97, 119, 127).

[53] M. GJ. and S. C. *The Art of Software Testing*. John Wiley & Sons, 2004. (pages 4, 6).

[54] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. Technical report, National Institute of Statistical Sciences, 653–661, 1998. (pages 87, 124).

[55] M. Griss, J. J. Favaro, and M. d'Alessandro. Integrating feature modeling with the rseb. In *Fifth International Conference on Software Reuse*, pages 76–85, Jun 1998. doi: 10.1109/ICSR.1998.685732. (page 23).

[56] A. E. Hassan. Predicting faults using the complexity of code changes. In *International Conference on Software Engineering*, pages 78–88, 2009. (page 90).

[57] H. Hemmati and L. Briand. An industrial investigation of similarity measures for model-based test case selection. In *ISSRE*, 2010. (page 64).

[58] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Multi-objective test generation for software product lines. In *International Software Product Line Conference*, pages 62–71, 2013. (pages 6, 8, 48, 51, 106, 107).

[59] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon. Bypassing the combinatorial explosion: using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, 40:1, 2014. (pages 6, 8, 33, 38, 50, 51, 64, 72, 77, 96, 98, 106, 122).

[60] C. Henard, M. Papadakis, M. Harman, and Y. L. Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 517–528, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL `http://dl.acm.org/citation.cfm?id=2818754.2818819`. (page 51).

[61] K. Herzig, S. Just, A. Rau, and A. Zeller. Predicting defects using change genealogies. In *International Symposium on Software Reliability Engineering*, pages 118–127, 2013. (page 90).

[62] Y. Huang, C. Huang, J. Chang, and T. Chen. Design and analysis of cost-cognizant test case prioritization using genetic algorithm with test history. In *Computer Software and Applications Conference*, pages 413–418, 2010. (pages 33, 35, 43, 124).

[63] M. M. Islam, A. Marchetto, A. Susi, and G. Scanniello. A multi-objective technique to prioritize test cases based on latent semantic indexing. In *16th European Conference on Software Maintenance and Reengineering*, pages 21–30, March 2012. doi: 10.1109/CSMR.2012.13. (pages 44, 45).

[64] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *International Conference on Automated Software Engineering*, pages 233–244, Nov 2009. doi: 10.1109/ASE.2009.77. (pages 33, 41).

[65] M. Johansen, O. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *SPLC*, 2012. (pages 50, 72, 96, 99).

[66] M. Johansen, ¯. Haugen, F. Fleurey, E. Carlson, J. Endresen, and T. Wien. A technique for agile and automatic interaction testing for product lines. In *Conference Testing Software and Systems*, pages 39–54, 2012. (page 6).

[67] M. F. Johansen, O. Haugen, and F. Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In *MODELS*, 2011. (pages 70, 72, 127).

[68] M. F. Johansen, O. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen. Generating better partial covering arrays by modeling weights on sub-product lines. In *International Conference MODELS*, 2012. (pages 4, 8, 33, 42, 51, 106).

[69] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. In *SEI*, 1990. (pages 22, 81, 108).

[70] G. M. Kapfhammer and M. L. Soffa. Using coverage effectiveness to evaluate test suite prioritizations. In *1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, WEASELTech '07, pages 19–20, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-880-0. doi: 10.1145/1353673.1353677. URL http://doi.acm.org/10.1145/1353673.1353677. (page 46).

[71] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24rd International Conference on Software Engineering*, pages 119–129, May 2002. doi: 10.1109/ICSE.2002.1007961. (pages 33, 34, 40).

[72] R. Krishnamoorthi and S. A. S. A. Mary. Factor oriented requirement coverage based system test case prioritization of new and regression test cases. *Information and Software Technology*, 51(4):799 – 808, 2009. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2008.08.007. URL http://www.sciencedirect.com/science/article/pii/S0950584908001286. (pages 33, 37, 38, 46).

[73] B. P. Lamancha and M. P. Usaola. Testing product generation in software product lines using pairwise for feature coverage. In *International conference on Testing Software and Systems*, 2010. (pages 49, 56).

[74] B. P. Lamancha, M. Polo, and M. Piattini. *5th International Conference Software and Data Technologies*, chapter Systematic Review on Software Product Line Testing,

pages 58–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-29578-2. doi: 10.1007/978-3-642-29578-2_4. URL http://dx.doi.org/10.1007/978-3-642-29578-2_4. (pages 6, 48, 49).

[75] J. Lee, S. Kang, and D. Lee. A survey on software product line testing. In *16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 31–40, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1094-9. doi: 10.1145/2362536.2362545. URL http://doi.acm.org/10.1145/2362536.2362545. (page 49).

[76] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *International Symposium on Software Reliability Engineering*, 2003. (pages 33, 38, 40).

[77] K. S. Lew, T. S. Dillon, and K. E. Forward. Software complexity and its impact on software reliability. *Transactions on software engineering*, 14:1645–1655, 1988. (pages 85, 125).

[78] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 2007. (pages 31, 44, 56).

[79] R. Lopez-Herrejon, F. Chicano, J. Ferrer, A. Egyed, and E. Alba. Multi-objective optimal test suite computation for software product line pairwise testing. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013. (page 51).

[80] R. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed. 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*, pages 1–10, 2015. doi: 10.1109/ICSTW.2015.7107435. URL http://dx.doi.org/10.1109/ICSTW.2015.7107435. (page 49).

[81] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, and E. Alba. Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of software product lines. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, July 6-11, 2014*, pages 387–396, 2014. doi: 10.1109/CEC.2014.6900473. URL http://dx.doi.org/10.1109/CEC.2014.6900473. (pages 8, 107, 118).

[82] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, E. N. Haslinger, A. Egyed, and E. Alba. A parallel evolutionary algorithm for prioritized pairwise testing of software

product lines. In *Genetic and Evolutionary Computation Conference*, pages 1255–1262, 2014. (pages 6, 8, 33, 38, 41, 42, 48, 51, 106).

[83] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, and E. Alba. *Computational Intelligence and Quantitative Software Engineering*, chapter Evolutionary Computation for Software Product Line Testing: An Overview and Open Challenges, pages 59–87. Springer International Publishing, Cham, 2016. ISBN 978-3-319-25964-2. doi: 10.1007/978-3-319-25964-2_4. URL http://dx.doi.org/10.1007/978-3-319-25964-2_4. (page 47).

[84] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the linux kernel variability model. In *SPLC*, 2010. (page 4).

[85] I. Machado, J. D. McGregor, and E. S. de Almeida. Strategies for testing products in software product lines. *SIGSOFT Software Engineering*, 2012. (pages 48, 56).

[86] A. Malishevsky, J. Ruthruff, G. Rothermel, and S. Elbaum. Cost-cognizant test case prioritization. Technical report, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 2006. (pages 33, 43).

[87] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60, 03 1947. doi: 10.1214/aoms/1177730491. URL http://dx.doi.org/10.1214/aoms/1177730491. (page 128).

[88] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu. Practical pairwise testing for software product lines. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 227–235, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1968-3. doi: 10.1145/2491627.2491646. URL http://doi.acm.org/10.1145/2491627.2491646. (pages 6, 48, 50, 106).

[89] S. Matsumoto, Y. kamei, A. Monden, K. Matsumoto, and M. Nakamura. An analyses of developer metrics for fault prediction. In *International Conference on Predictive Models in Software Engineering*, number 18, 2010. (pages 85, 87, 125).

[90] P. McMinn. Search-based software test data generation: A survey: Research articles. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004. ISSN 0960-0833. doi: 10.1002/stvr.v14:2. URL http://dx.doi.org/10.1002/stvr.v14:2. (page 44).

[91] M. Mendonca. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, 2009. (pages 57, 65, 72).

[92] M. Mendonca, M. Branco, and D. Cowan. S.p.l.o.t. - software product lines online tools. In *OOPSLA*, 2009. (pages 56, 66, 76, 84).

[93] M. Mendonca, A. Wasowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the Sofware Product Line Conference*, 2009. (page 58).

[94] G. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing, 3rd Edition*. Wiley John & Sons, 2011. ISBN 0471469122. (page 6).

[95] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *International Symposium on Software Reliability Engineering*, pages 309–318, 2010. (page 90).

[96] H. V. Nguyen, C. Kästner, and T. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *International Conference on Software Engineering*, pages 907–918, 6 2014. (page 103).

[97] T. Noguchi, H. Washizaki, Y. Fukazawa, A. Sato, and K. Ota. History-based test case prioritization for black box testing using ant colony optimization. In *International Conference on Software Testing, Verification and Validation*, pages 1–2, April 2015. doi: 10.1109/ICST.2015.7102622. (pages 33, 35).

[98] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *SPLC*, 2010. (page 56).

[99] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *International symposium on Software testing and analysis*, pages 86–96, 2004. (page 92).

[100] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Conference Software Testing, Verification and Validation*, 2010. (pages 50, 56).

[101] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Budry, and Y. le Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 2011. (page 49).

[102] N. Prakash and T. R. Rangaswamy. Multiple criteria based test case prioritization for regression testing. *European Journal of Scientific Research*, 84(1):36–45, 2012. (page 7).

[103] S. R. Pressmen. *Software Engineering: A practitioners Approach*. McGraw Hill, International Edition-5, 2001. (page 86).

[104] X. Qu, M. B. Cohen, and K. M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *IEEE International Conference on Software Maintenance*, pages 255–264, Oct 2007. doi: 10.1109/ICSM.2007.4362638. (pages 46, 56).

[105] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *International Symposium in Software Testing and Analysis*, 2008. (pages 8, 33, 36, 41, 107).

[106] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. URL `http://www.R-project.org`. (page 92).

[107] M. Riebisch, K. Bllert, D. Streitferdt, and I. Philippow. Extending feature diagrams with uml multiplicities. 6th World Conference on Integrated Design & Process Technology, 2002. (page 26).

[108] M. Riebisch, D. Streitferdt, and I. Pashov. Modeling variability for object-oriented product lines. In *Workshop Reader Object-Oriented Technology*, volume 3013 of *Lecture Notes in Computer Science*, pages 165–178. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22405-1. doi: 10.1007/978-3-540-25934-3_16. URL `http://dx.doi.org/10.1007/978-3-540-25934-3_16`. (page 26).

[109] S. Roongruangsuwan and J. Daengdej. Test case prioritization techniques. *Theoretical and Applied Information Technology*, 2010. (pages 34, 35, 37, 38).

[110] G. Rothermel, R. Untch, C. Chengyun, and M. Harrold. Test case prioritization: an empirical study. In *IEEE International Conference on Software Maintenance*, pages 179–188, 1999. doi: 10.1109/ICSM.1999.792604. (pages 31, 56).

[111] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions and Software Engineering*, 27:929–948, 2001. (pages 6, 7, 31, 33, 40, 45, 56, 113, 115).

[112] A. B. Sánchez and S. Segura. Automated testing on the analysis of variability-intensive artifacts: An exploratory study with sat solvers. In *XVII Jornadas de Ingeniería del Software y de Bases de Datos (JISBD 2012)*, Almera, Spain., 2012. (page 12).

[113] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. Priorización de casos de prueba. avances y retos. *Novática: Revista de la Asociación de Técnicos de Informática*, (224): 27–32, Aug 2013. ISSN Digital. (page 12).

[114] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. A comparison of test case prioritization criteria for software product lines. In *IEEE International Conference on Software Testing, Verification, and Validation*, pages 41–50, Cleveland, OH, April 2014. doi: 10.1109/ICST.2014.15. (pages 10, 13, 73, 77, 95, 96, 98, 106, 121, 122, 123).

[115] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. The drupal framework: A case study to evaluate variability testing techniques. In *8th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2014. doi: http://dx.doi.org/10.1145/2556624.2556638. (pages 10, 11, 13, 104).

[116] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. Towards multi-objective test case generation for variability-intensive systems. In *Jornadas de Ingeniería del Software y de Bases de Datos (JISBD 2014)*, pages 205–210, Cádiz, Spain, 09/2014 2014. (pages 11, 13).

[117] A. B. Sánchez, S. Segura, and A. R. Cortés. Smartest: Accelerating the detection of faults in drupal. In *DrupalConEurope 2015*, 09/2015 2015. (pages 12, 14, 15, 140, 152).

[118] A. B. Sánchez, S. Segura, J. A. Parejo, and A. Ruiz-Cortés. Variability testing in the wild: The drupal case study. *Software and Systems Modeling Journal*, pages 1–22, Apr 2015. ISSN 1619-1366. doi: 10.1007/s10270-015-0459-z. (pages 10, 11, 13, 104, 106, 107, 111, 125, 139).

[119] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. Smartest: Proposal for accelerating the detection of faults in drupal. In *DrupalCampSpain 2015*, Cadiz, 05/2015 2015. (pages 12, 14, 15, 152).

[120] A. S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *International Conference on Software Engineering*, pages 492–501, 2013. (page 51).

[121] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 2(51):456–479, 2007. (page 29).

[122] S. Segura. Automated analysis of feature models using atomic sets. In *First Workshop on Analyses of Software Product Lines (ASPL 2008). SPLC'08*, pages 201–207, Limerick, Ireland, September 2008. (page 118).

[123] S. Segura and A. Ruiz-Cortés. Benchmarking on the automated analyses of feature models: A preliminary roadmap. In *VaMoS*, 2009. (page 76).

[124] S. Segura, J. Galindo, D. Benavides, J. Parejo, and A. Ruiz-Cortés. Betty: Benchmarking and testing on the automated analysis of feature models. In *International Workshop on Variability Modelling of Software-intensive Systems*, 2012. (pages 66, 72).

[125] S. Segura, A. B. Sánchez, and A. Ruiz-Cortés. Automated variability analysis and testing of an e-commerce site: An experience report. In *International Conference on Automated Software Engineering*, pages 139–150. ACM, 2014. doi: 10.1145/2642937.2642939. (pages 11, 13, 103).

[126] S. Segura, A. Durn, A. B. Snchez, D. L. Berre, E. Lonca, and A. Ruiz-Corts. Automated metamorphic testing of variability analysis tools. *Software Testing, Verification and Reliability*, 25(2):138–163, 2015. ISSN 1099-1689. doi: 10.1002/stvr.1566. URL http://dx.doi.org/10.1002/stvr.1566. (page 14).

[127] S. She, R. Lotufo, T. Berger, A. Wasowski, and k. Czarnecki. The variability model of the linux kernel. In *International Workshop on Variability Modelling of Software-intensive Systems*, 2010. (pages 4, 76).

[128] M. Sherriff, M. Lake, and L. Williams. Prioritization of regression tests using singular value decomposition with empirical change records. In *The 18th IEEE International Symposium on Software Reliability*, pages 81–90, Nov 2007. doi: 10.1109/ISSRE.2007.25. (pages 33, 35).

[129] C. Simons and E. C. Paraiso. Regression test cases prioritization using failure pursuit sampling. In *International Conference on Intelligent Systems Design and Applications*, pages 923–928, 2010. (pages 33, 34, 38, 124).

[130] SPL2GO. SPL2GO repository. http://spl2go.cs.ovgu.de/, accessed February 2016. (page 76).

[131] splot. S.P.L.O.T.: Software Product Lines Online Tools. http://www.splot-research.org/, accessed February 2016. (pages 76, 84).

[132] H. Srikanth and L. Williams. On the economics of requirements-based test case prioritization. In *Seventh International Workshop on Economics-driven Software Engineering Research*, EDSER '05, pages 1–3, New York, NY, USA, 2005. ACM. ISBN 1-59593-118-X. doi: 10.1145/1082983.1083100. URL `http://doi.acm.org/10.1145/1082983.1083100`. (page 46).

[133] H. Srikanth and L. Williams. Requirements-based test case prioritization. Technical report, Department of Computer Science, North Carolina State University, 2005. (pages 33, 37, 38, 41).

[134] H. Srikanth, M. B. Cohen, and X. Qu. Reducing field failures in system configurable software: Cost-based prioritization. In *20th International Symposium on Software Reliability Engineering*, pages 61–70, 2009. (pages 33, 34, 43, 45, 115).

[135] H. Srikanth, S. Banerjee, L. Williams, and J. Osborn. Towards the prioritization of system test cases. *Software testing, Verification and Reliability*, (24):320–337, 2013. (pages 33, 37, 38, 41, 42).

[136] L. Tahat, B. Korel, M. Harman, and H. Ural. Regression test suite prioritization using system models. *Software Testing, Verification and Reliability*, 22(7):481–506, Nov. 2012. ISSN 0960-0833. doi: 10.1002/stvr.461. URL `http://dx.doi.org/10.1002/stvr.461`. (pages 33, 39).

[137] P. N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2006. (pages 64, 98).

[138] T. Thomas, D. Batory, and C. Kastner. Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 254–264, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070526. URL `http://dx.doi.org/10.1109/ICSE.2009.5070526`. (page 22).

[139] U. Tiwari and S. Kumar. Cyclomatic complexity metric for component based software. In *Software Engineering Notes*, volume 39, pages 1–6, January 2014. (page 86).

[140] T. Tomlinson and J. K. VanDyk. *Pro Drupal 7 development: third edition*. 2010. (pages xi, xii, 15, 76, 78, 79, 83, 102, 111, 149, 150).

[141] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A. Jimenez. Fama framework. In *Software Product Line Conference*, 2008. (page 57).

[142] E. Uzuncaova, S. Khurshid, and D. S. Batory. Incremental test generation for software product lines. *Transactions Software Engineering*, 36(3):309–322, 2010. doi: 10.1109/TSE.2010.30. URL http://doi.ieeecomputersociety.org/10.1109/TSE.2010.30. (page 108).

[143] A. Vargha and H. D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000. (page 130).

[144] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger. Presence-condition simplification in highly configurable systems. In *International Conference on Software Engineering*, 2015. (pages 4, 106).

[145] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *International Symposium on Software Testing and Analysis*, ISSTA '06, pages 1–12, New York, NY, USA, 2006. ACM. ISBN 1-59593-263-1. doi: 10.1145/1146238.1146240. URL http://doi.acm.org/10.1145/1146238.1146240. (page 44).

[146] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, and M. Liaaen. Multi-objective test prioritization in software product line testing: An industrial case study. In *Software Product Line Conference*, pages 32–41, 2014. (pages 6, 8, 33, 36, 41, 43, 44, 47, 48, 52, 106).

[147] S. Wang, S. Ali, and A. Gotlieb. Cost-effective test suite minimization in product lines using search techniques. *Journal of Systems and Software*, 103:370–391, 2015. doi: 10.1016/j.jss.2014.08.024. URL http://dx.doi.org/10.1016/j.jss.2014.08.024. (page 7).

[148] Y. Wang, X. Zhao, and X. Ding. An effective test case prioritization method based on fault severity. In *International Conference on Software Engineering and Service Science*, pages 737–741, Sept 2015. doi: 10.1109/ICSESS.2015.7339162. (pages 33, 37).

[149] Z. Xu, M. B. Cohen, W. Motycka, and G. Rothermel. Continuous test suite augmentation in software product lines. In *Software Product Line Conference*, 2013. (pages 8, 107).

[150] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. In *Software Testing, Verification and Reliability*, volume 22, pages 67–120, 2012. (pages 7, 31, 34, 35, 37, 38, 87, 124).

[151] A. Zhou, B. Qu, H. Li, S. Zhao, P. N. Suganthan, and Q. Zhang. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, 1(1):32 – 49, 2011. ISSN 2210-6502. doi: http://dx.doi.org/10.1016/j.swevo.2011.03.001. URL `http://www.sciencedirect.com/science/article/pii/S2210650211000058`. (page 110).