Software Quality Journal manuscript No.
(will be inserted by the editor)

Testing variability intensive systems using automated analysis. An application to Android

José A. Galindo · Hamilton Turner · David Benavides · Jules White.

Received: date / Accepted: date

Abstract Software product lines are used to develop a set of software products that, while being different, share a common set of features. Feature models are used as a compact representation of all the products (e.g., possible configurations) of the product line. The number of products that a feature model encodes may grow exponentially with the number of features. This increases the cost of testing the products within a product line. Some proposals deal with this problem by reducing the testing space using different techniques. However, a daunting challenge is to explore how the cost and value of test cases can be modelled and optimized in order to have lower cost testing processes. In this paper, we present TESting vAriAbiLity Intensive Systems (TESALIA), an approach that uses automated analysis of feature models to optimize the testing of variability intensive systems. We model test value and cost as feature attributes and then we use a constraint satisfaction solver to prune, prioritize and package product line tests complementing prior work in the software product line testing literature. A prototype implementation of TESALIA is used for validation in an Android example showing the benefits of maximizing the mobile market-share (the value function) while meeting a budgetary constraint.

Keywords Testing · Software product lines · Automated analysis · Feature models · Android

1 Introduction

Software product line engineering encompasses the methods, techniques and tools for developing a set of varying software products that share a subset of common functionality (van der Linden et al, 2007). The concrete functionality that varies across products in the product line is encapsulated using an abstraction known as *feature*. To represent the common and varying features within a product line, a variety of variability modeling techniques have been proposed (Sinnema and Deelstra, 2007). However, feature models are one of the most widely used techniques to model the variability of a software product line (Kang et al, 1990). To encode the set of all software product line products in a feature model's tree-like structure, a variety of parent-child relationships are used to represent the constraints governing the possible feature permutations of valid products in the product line. The number of products encoded in a feature model grows with the number of features. Given n features and no constraints on valid feature combinations, there are 2^n possible products. To deal with this complexity, automated mechanisms

José A. Galindo · David Benavides
Dept. Lenguajes y Sistemas Informáticos, University of Seville,
Avda. Reina Mercedes s/n, 41012, Seville, Spain.
E-mail: {jagalindo, benavides}@us.es

Hamilton Turner · Jules White
Bradley Department of Electrical and Computer Engineering, Virginia Tech,
Blacksburg, Virginia, 24060, United States.
E-mail: hamiltont@vt.edu

Jules White
Department of Electrical Engineering and Computer Science, Vanderbilt University
Nashville, TN, 37212 United States.
E-mail: jules@dre.vanderbilt.edu

are used to extract information from feature models, such as features present in every product. These automated mechanisms are known as *automated feature model analysis* techniques (Benavides et al, 2010).

Some software product lines, such as those describing operating systems (She et al, 2010; Galindo et al, 2010), have thousands of features (Batory et al, 2006). When a product line has an exponential number of products relative to its feature count, testing all or a large percentage of its products is an error-prone and expensive task, both in terms of time and computing resources. To address the issue of not being able to test all the possible products of a software product line, combinatorial testing (Nie and Leung, 2011) can be used. Combinatorial testing reduces the number of tests to execute by taking advantage of the shared features across different software product line products. Roughly speaking, the hypothesis behind combinatorial testing is that if a given feature has already been tested, then the testing process can skip further tests containing this feature. A variety of heuristics can be used to decide which feature combinations to test, however, researchers have primarily used t-wise heuristics for pruning the set of products to test.

The testing component of a software engineering process can be time-consuming and expensive (Beizer, 2003). One of the parameters to be considered when selecting test cases, is test case cost and effort. The same is true in software product line engineering. Past literature has investigated techniques for reducing testing effort in software product lines (Rothermel and Hall, 1997; Boehm, 2005; Lamancha and Usaola, 2010; Perrouin et al, 2010; Withey, 1996; Perrouin et al, 2011; Boehm and Sullivan, 1992; Sneed, 2009; Johansen et al, 2012b; Oster et al, 2010; Johansen et al, 2012a; Binkley and Society, 1997; Srikanth et al, 2005; Sayyad et al, 2013; Henard et al, 2013; Olaechea et al, 2012). Traditionally, testing cost has not been modeled explicitly. For example, when applying t–wise based approaches (Lamancha and Usaola, 2010; Perrouin et al, 2010, 2011; Johansen et al, 2012b; Oster et al, 2010), the cost to test a feature is extracted from the number of product configurations that include a specific feature. Modeling cost information in a more explicit way can help to reduce the number of products to be tested in a software product line. For example, if we are testing in the cloud, the time and budget required for executing a test may vary between different executions. Time and budget are examples of testing cost. Current approaches for testing software product lines do not consider these situations where test cost is not uniform.

When the time to test all products in a product-line outstrips the available testing budget, determining how much value is obtained by running a test enables the prioritization of the products to be tested. When there is limited time for testing due to deadlines, test prioritization enables the critical parts of the software to be tested when not all tests can be executed (Spillner et al, 2011). Test value Boehm (2005) can be measured in different ways such as the measuring the feature's relative importance in terms of the project requirements (e.g., points on associated user stories) or deriving importance based on the time left until its expected completion date in the project schedule. The value of a software product changes as time goes by. For example, a new mobile phone may be widely used when it is first introduced in the market. Over time, as its feature set becomes older and less attractive, it may have fewer and fewer users. Therefore, the value of thoroughly testing an application running on a particular phone may provide more or less value depending on when the tests are run. Mechanisms to model the test value in software product lines can be used to properly reduce the amount of products to test while maximizing the value of the tests that are run.

To improve the testing process researchers have primarily proposed two mechanisms; i) *test pruning*, to selectively execute tests based on test-cost when there are a large number of tests and (Nie and Leung, 2011) ii)*test prioritization*, to ensure that the most important tests always run first in case testing is interrupted (Spillner et al, 2011). With the first approach, developers can reduce the number of tests to execute while maintaining good test coverage. With the second approach, developers execute the most valuable tests first in order to ensure that critical software components are tested. Moreover, when companies use test-driven development approaches (Beck, 2003), the large number of test executions when using software product lines can result in high costs. Being able to fix a maximum budget for a test-suite enables developers to bound the cost of the testing phase while maximizing testbed value.

Automated analysis of feature models is the process of extracting information from feature models using computer aided mechanisms (Benavides et al, 2010), such as constraint solvers. More than thirty different approaches have been proposed to extract varying kinds of information from feature models, such as how commonly a feature is used or the number of unused features (Benavides et al, 2010).

In this paper we propose to use automated analysis of feature models to automate the pruning and prioritization of the tests to be executed within a product line. We present TESALIA, a method to analyze the set of products encoded in a software product line, as well as feature value and cost information, in order to prioritize the products to be tested. We also propose a method to build a product test suite that adheres to a fixed testing budget while maximizing the value of the tested products. This paper provides the following contributions:

- A mechanism to store, update and reason about value and product cost data.
- A CSP-based approach to prune the product set encoded in a feature model. This approach extends prior work (Lamancha and Usaola, 2010; Perrouin et al, 2010, 2011; Johansen et al, 2012b; Oster et al, 2010) by enabling the use of a variety of objective functions for pruning test cases. For example, a 2-wise function or a function based on test execution time can be used to prune the tests.
- A CSP technique to prioritize the list of products to be executed in order to maximize testing value.
- A technique for packaging tests with cost and value data to bound testing cost while maximizing test suite value.
- A prototype implementation (TESALIA) and a validating mobile phone use-case study based on Android.

The remainder of this paper is structured as follows: Section 2 presents background information on software product line testing and feature modelling. A motivating example based on mobile phones is presented in Section 3. In Section 4 we present i) a method to capture cost and value data in a feature model (Section 4.1); ii) a CSP-model to prune the products set (Section 4.3); iii) a knapsack based approach to prioritize the product list for testing (Section 4.4) and package the tests to obtain the maximum value for a fixed cost (Section 4.5). An evaluation of our approach based on the mobile phone case-study is presented in Section 5. Finally, in Sections 6 and 7 we present concluding remarks and lessons learned.

2 Preliminaries

In this section we present a brief overview of key feature modeling and software product line testing concepts.

2.1 Feature models

Feature models are commonly used to represent the commonality and variability of the products inside a software product line. Feature models were first defined by Kang (Kang et al, 1990) and have become the "defacto" standard for describing variabilities and commonalities when using software product lines. An example feature model based on the mobile industry is presented in Figure 1. A feature model represents all the different products in a software product line. Feature Models use different kinds of relationships to define the shared and distinct features between different software products. Usually, two different groups of relationships are defined: *i*) hierarchical relationships to define a variation point in a product line where features share a common parent and *ii*) cross-tree constraints to define restrictions on features that do not share a common parent in the feature model tree. Different feature model representations can be found in the literature (Benavides et al, 2010). The most well-known feature model representations are:

Basic Feature Models. Figure 1 shows a graphical representation of this feature model representation. The basic feature model defines four kinds of hierarchical relationships:

- **mandatory**: this relationship specifies that when a parent feature is present in the product, the child feature also must be present;
- **optional**, this relationship refers to features that are not required to be in a product when their parent feature is present;
- **set**, this relationship is between one parent feature and a group of child features and implies that between 1 and N of the child features should be present if the parent is present in the product. Special cases of the set relationship are **alternative**, choose only one of the features in the set; and **or**, choose any of the features in the set;

In addition to these hierarchical constraints, three cross-tree constraints are defined:

- **requires** indicates that when the origin feature is present in the product, the destination feature must be also present;
- **excludes** ensures that when the origin feature is present in the product, the destination feature is not in the product and vice-versa.
- **cnf-based cross–tree constraints** are more complex constraints that can be encoded as propositional formulas, e.g. "feature a requires b or c" but only considering boolean variables.

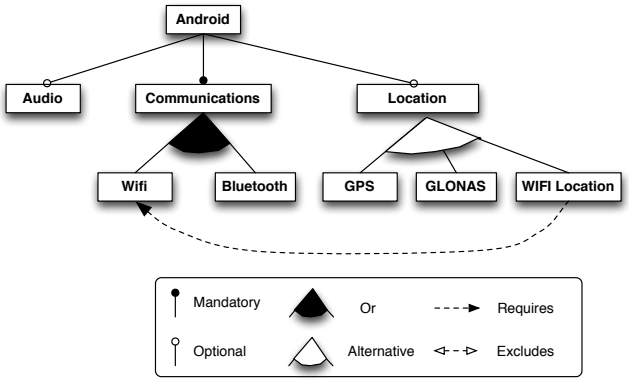

Fig. 1 Feature model example from the mobile industry.

Attributed feature models. An attributed feature models is a feature modeling extension that includes attributes on features. Also constraints, called complex cross-tree constraints, can be added. Complex cross-tree constraints are allowed between features and attributes such as "Every attribute called cost must have a value greater than 10", this is, constraints involving features and attributes. There are a variety of approaches to describe feature model attributes (Roos-Frantz et al, 2012), however, most of them share some characteristics. Usually attributes are defined by the use of a name, a domain, a value and a null value. The attribute domain represents the set of values that the attribute can assume. The value, is the value of the attribute when its associated feature is present in the product, the null value refers to the value of the attribute when its associated feature is not present in the product. In Figure 2 shows a concrete attributed feature model. In the figure, quality attributes are presented showing the accuracy of each location service in a mobile phone.

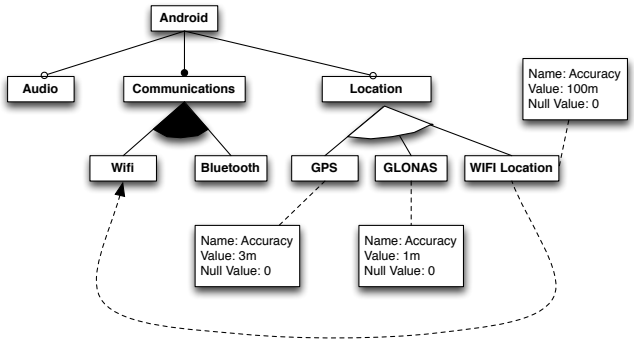

Fig. 2 Attributed feature model example.

2.2 Software product line testing.

Software product line testing represents a new challenge for software testing researchers (Pohl and Metzger, 2006). When testing software product lines, each product shares some common functionality with one or more other products, while differing in at least one feature. Software product lines add testing complexity because they require testing a set of products rather than a single product. These products, however, share common functionality or artifacts, enabling the reuse of some tests across the entire product line.

According to (Pohl and Metzger, 2006), varying strategies can be used to test software product line products. These testing strategies can be summarized as follows: *i*) testing product by product and *ii*) incremental testing, and *iii*) Reusable asset instantiation. Testing product by product is a strategy that tests all products one by one, as if they were not part of a product line. With this strategy the test process covers all possible interactions between features but grows exponentially in cost as a function of the number of features in the product line. Incremental testing is a strategy that starts by testing the first developed product and creates new unit tests for each new feature added. Using this strategy, the commonalities in the software product line are exploited to reduce testing effort. However, when a new feature is introduced, all the interactions between the new feature and the old ones must also be tested, which can be challenging for large software product lines. Reusable asset instantiation relies on data captured in the domain analysis stage of product line creation to develop a set of abstract test cases that cover all features (but not necessarily configurations) in the product line. These abstract tests cases are mapped to concrete requirements in the application engineering stage. These testing strategies are designed to reduce the software product line combinatorial explosion in testing cost as a function of the feature count.

Combinatorial testing. Combinatorial interaction testing (Nie and Leung, 2011) exploits the commonalities of a software product to reduce the cost incurred when testing. This approach has been applied to the testing of software product lines because of the functionality encapsulation used in software product line engineering. Concretely t-wise approaches (Lamancha and Usaola, 2010; Perrouin et al, 2010, 2011; Johansen et al, 2012b; Oster et al, 2010) have been used to apply combinatorial testing to software product lines. For example, when using t-wise coverage, at least every feature is covered t times. Combinatorial testing has been proven to produce good results in empirical studies (Dalal et al, 1999; Smith and Feather, 2000), mainly because points of interaction between software artifacts have been proven (Kuhn et al, 2004) to be key sources of errors. Nevertheless, there is still consensus that concrete techniques are needed in software product line engineering to aid in reducing testing cost while maximizing the test value because of the large number of potential products that must be tested.

3 Motivating scenario

Figure 3 shows the motivating scenario that promoted this research. Mobile phone applications *("apps")* are typically executed on a variety of different underlying mobile platform configurations, such as different versions or configurations of Android or iOS. Each platform configuration has different mobile platform features, such as screen resolution or communication capabilities (*e.g.*, 3G, LTE, etc.) that the software must interact with and may be configured differently depending on the platform version. For example, iOS 6 on the iPhone 3GS does not have turn by turn directions but iOS 6 on the iPhone 5 does. Also, the iPhone 3GS-4S has the same screen aspect ratio but the iPhone 5 does not. The Android emulator [1], which allows developers to emulate the configuration options of real-world devices, currently supports 46 different mobile platform features, which (presuming all user configurable features could be combined with no restrictions) potentially leads to 2^{46} different platform variant configurations. The high variability that exists in the Android ecosystem makes testing hard. Being able to describe the variability of the Android ecosystem as a software product line would allow developers to apply existing and new testing techniques to optimize testing strategies.

Because of the large number of in-use platform configurations, it is difficult for developers, regardless of development team size or proficiency, to test their software products on all or even most platform configurations before release. One of the most common approaches is to pick a set of the most popular mobile devices and then to test exclusively for that subset of devices. For example, the Android

[1] `http://developer.android.com/tools/help/emulator.html`

Skype application is installed on thousands of unique Android platform configurations [2], but the Skype application is only officially certified to work on fewer than 25 Android devices[3].

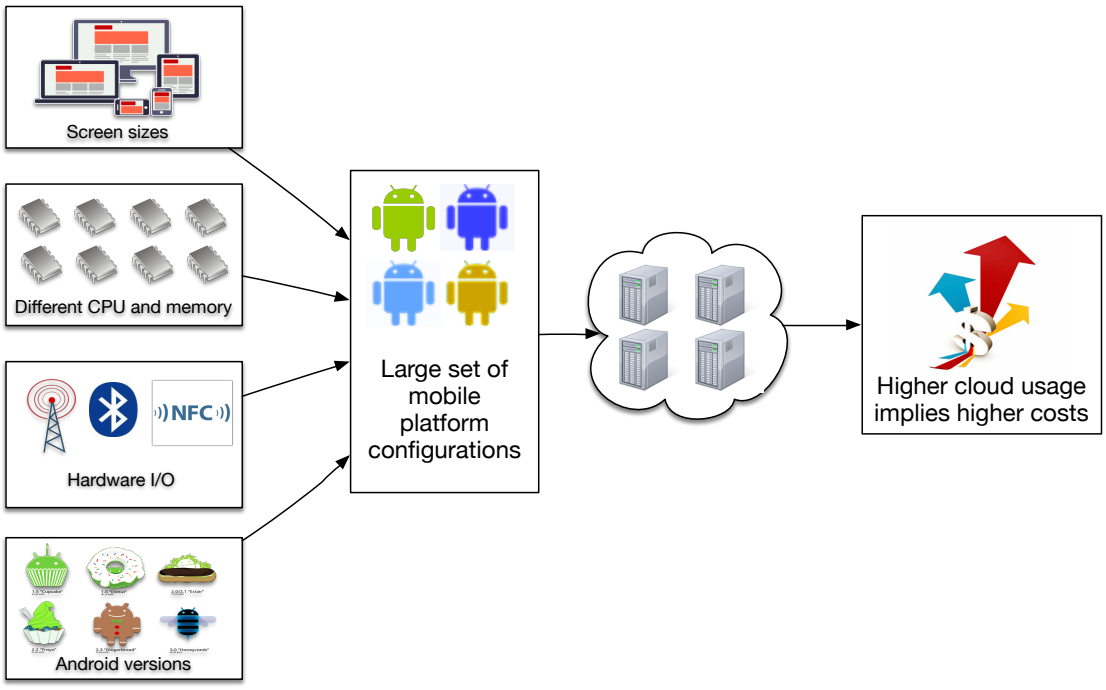

Fig. 3 Android variability impacting testing costs.

A primary challenge of testing on a large number of mobile platforms is the cost of acquiring so many unique smartphone devices. As an alternative to testing on actual devices, developers frequently turn to platform emulation to mimic real-world platform configurations. However, running a complete suite of tests on hundreds of uniquely configured platform emulator instances requires substantially more computing power than the average developer possesses. An option to address this problem is to use the cloud as a supporting platform for running virtual instances of the emulating configurations.

To guarantee the correct operation of Android applications developers can adopt a test-driven development process (Beck, 2003). Test-driven development relies on the repetition of a set of tests in short development cycles (e.g. every time we release a new version, every time we add a new functionality). When developing applications that potentially will run in a wide range of different platform configurations users need to select which configurations will be used to run the tests. Usually developers may want to improve the value of their tests by defining different metrics such as market-share impact, which is the number of real devices (mobile platform features) and mobile platform features covered by the tests. Market-share analysis can be used to help determine which tests to run and automate this testing in the cloud.

To reduce the complexity of testing software product lines, a number of challenges need to be addressed:

Modeling the variability in the mobile platform ecosystem. A daunting challenge of developing mobile applications is the rapid pace that the underlying mobile platforms and device hardware capabilities are evolving. This rapid pace of evolution has led to platform fragmentation, which is the large variability in platform configurations used on different mobile devices in the market. Fragmentation may be caused by differences in manufacturer modifications to an open-source platform, such as on Android, or the evolution of a single manufacturer's product line, such as Apple's iOS products. For example the bluetooth API changed substantially between Android versions, making apps designed

[2] `http://opensignal.com/reports/fragmentation.php`

[3] `http://www.skype.com/intl/en-us/get-skype/on-your-mobile/skype-mobile/android/`

for initial versions fail in current flag-ship devices. Many of these configuration variations, such as differences in how the API to access a user's contacts functions, can directly impact the functionality of apps. The fragmentation creates issues when testing because of the large amount of testing required to ensure that the app works properly across the entire platform configuration space.

Defining the cost of testing a mobile application. Regardless of whether real mobile devices or emulator instances executed in a cloud, such as Amazon EC2, are used, developers typically do not have the time or money to test their app on top of every possible mobile platform configuration. A key issue, therefore, is determining how many and which mobile platform configurations to test a mobile app on without exceeding development time or cost budgets. For example, although a cloud can be used to test thousands of emulated mobile device instances, most cloud-computing providers charge fees based on the amount of CPU time consumed by an application, which can lead to substantial cost. For example, using Amazon EC2 [4] to test an Android application that runs on top of 1000 unique platform configurations may require 1000 cloud instances. If these instances have, on average, a cost of $1.006 per hour and each test consumes $\frac{1}{2}$ hour, the total testing cost will be $503, or $1006 per hour. Thus, although developers *can* test their software on thousands of platform configurations, they must determine the number of configurations to test given their desired cost, time, and coverage goals (tests profits).

The cost of executing a test can be measured in different ways depending on a company's requirements. For example, it can be measured in terms of money invested for running an emulator in the cloud, or in terms of carbon emissions. Moreover, it also can be measured by executing a test and capturing the time required to run it or the number of lines of code executed.

Determining the revenue of executing a test. A key approach that developers often use is to leverage sales data from a vendor, such as Amazon, in an attempt to maximize the market-share of the devices that their app has been tested on. The overall goal of developers is to select a set of top selling phones from a period of time in an attempt to minimize the number of mobile platform configurations that the app is installed on that it has not been tested on top of. However, to date, these approaches to selecting which mobile platform configurations to test on are manual, ad-hoc processes that are not optimized. Further, these manual processes to selecting mobile platform configurations to test on top of do not consider the complex tradeoffs between development budgets, time, and market-share coverage. Finally, these existing approaches of selecting top selling devices to test on have not been evaluated for market-share coverage or compared to other approaches.

Different value functions can be used to achieve different testing scopes. For example, if we use the market share metric, our test will reach as many users possible but might leave out android installations without networking capabilities (e.g., no Wifi or cellular modem hardware). If we want to look for errors on rare devices we can try to maximize the number of mobile platform features covered by our tests.

In Section 5, we describe how we use TESALIA to model Android variability, add cost and value information, and prioritize, prune, and package tests. For the sake of simplicity, in the next Section, we use a simplified version of the motivating example to present the key ideas. This example is shown in Figure 4. The simplified example is based on a feature model with three concrete features (Audio, Wifi and Bluetooth), all the features have a "cost" attribute representing the relative cost within the product and a "value" attribute for specifying the relative benefit of having the associated feature in a product. To simplify, the attributes are expressed as integer values.

4 The TESALIA solution

The TESALIA (TESting vAriAbiLity Intensive Systems) approach provides a mechanism for finding the most valuable set of products to test while meeting a fixed budget (cost). First, the variability present in an software product line is described using a feature model. Second, the feature model is attributed with information modeling test cost and value. Note that only features impacting test cost and value have to be attributed. Finally, using automated mechanisms, a list of products defining the selected product for testing is generated. In the remainder of this section, we present the key components of the TESALIA approach.

[4] http://aws.amazon.com/s3/

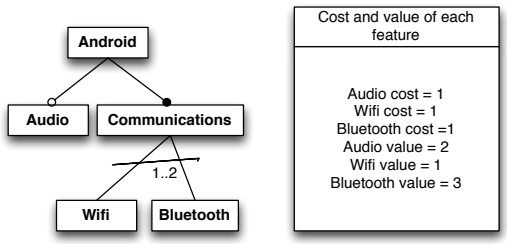

Fig. 4 feature model example based on the smartphone's motivating scenario.

Figure 5 shows the TESALIA solution approach. The key benefits of TESALIA are that it enables i) the pruning of the possible products to test; ii) the prioritization of the products to test based on value; and iii) the packaging of products into testable sets that meet a bounded cost constraint while maximizing the overall test value of the set. Each component of the solution can be executed sequentially. In other words, testing practitioners can decide either to execute all operations or only some. For pruning and packaging, developers must provide both a cost and value estimation function, based on feature attributes.

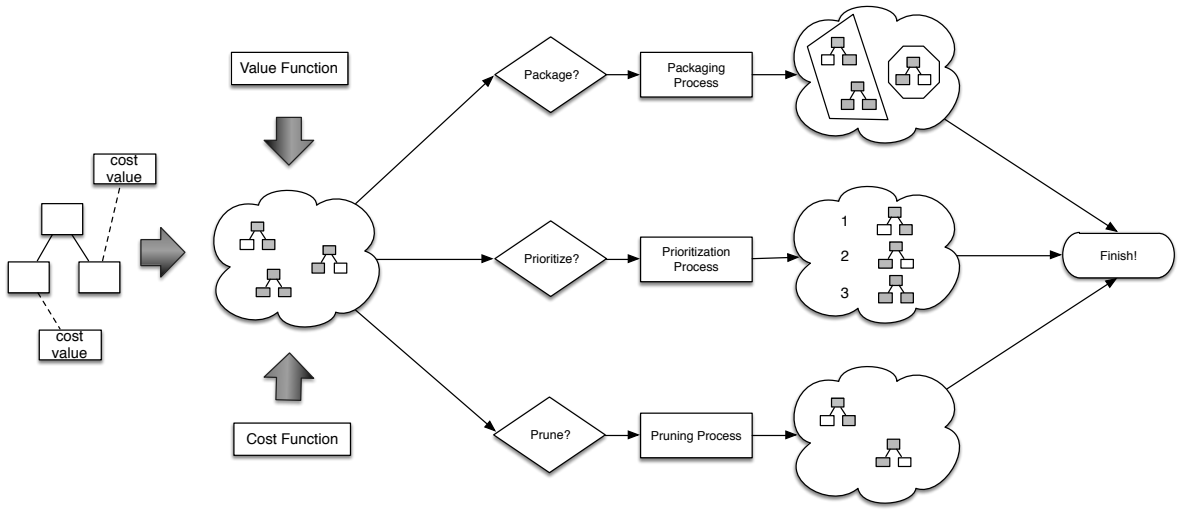

Fig. 5 Solution overview.

4.1 Capturing testing variability with feature models

In order to optimize the number of tests to run in a product line, computer-aided mechanisms are needed to describe the valid products that can be targeted at when testing. The number of different possible valid products in a product line can grow up to 2^n, where n is the number of features. In order to describe the valid and non-valid products in a product line, researchers use a variety of variability modeling approaches.

Although feature models have been used for modeling software product line variability, they can also be used to model other kinds of variability, such as cloud configurations (García-Galán et al, 2013; Dougherty et al, 2012) or Linux–based distributions (Galindo et al, 2010; She et al, 2010). Also, there is a recent trend to use feature models to describe the variability in an application's testing space (Rothermel and Hall, 1997; Boehm, 2005; Lamancha and Usaola, 2010; Perrouin et al, 2010; Withey, 1996; Perrouin et al, 2011; Boehm and Sullivan, 1992; Sneed, 2009; Johansen et al, 2012b; Oster et al, 2010; Johansen et al, 2012a; Binkley and Society, 1997; Srikanth et al, 2005; Sayyad et al, 2013; Henard et al, 2013; Olaechea et al, 2012).

For example, as motivated in Section 3, when testing Android applications on multiple Android devices, there are a large number of potential feature permutations of the Android emulator, such as audio and connectivity capabilities, screen size or CPU type. With such a large number of permutations, it is easy to create an invalid emulator configuration for testing, such as selecting Android version 1.5 along with a screen resolution of WVGA, which is not supported by that platform version. Therefore, when attempting to test across a wide range of configurations, a method of defining the valid configuration options and automatically discarding invalid configurations is required.

TESALIA uses feature models to describe the set of feature permutations (testing variability) available for testing as some other have proposed (Rothermel and Hall, 1997; Boehm, 2005; Lamancha and Usaola, 2010; Perrouin et al, 2010; Withey, 1996; Perrouin et al, 2011; Boehm and Sullivan, 1992; Sneed, 2009; Johansen et al, 2012b; Oster et al, 2010; Johansen et al, 2012a; Binkley and Society, 1997; Srikanth et al, 2005; Sayyad et al, 2013; Henard et al, 2013; Olaechea et al, 2012). Having the testing variability described in a feature model makes it possible to use automated techniques to identify valid feature permutations (Benavides et al, 2010). Moreover, it allows automated reasoning about the evolution of the product-line over time (White et al, 2014), such as the cost of transitioning to a different configuration containing a new Android version. For example, the valid products described by the feature model presented in Figure 4 are shown in Table 1. The processes for constructing these feature models is beyond the scope of this paper. However, there are a variety of techniques that have been described in prior work that can be used to reverse engineer feature models from product descriptions, such as domain engineering approaches (Kang et al, 1990) or computer-aided algorithms (Lopez-Herrejon et al, 2012; She et al, 2014).

Name	Android	Communication	Audio	Wifi	Bluetooth
P1	✓	✓	✓	✓	
P2	✓	✓	✓		✓
P3	✓	✓	✓	✓	✓
P4	✓	✓		✓	
P5	✓	✓			✓
P6	✓	✓		✓	✓

Table 1 Set of products described by the example presented in Figure 4.

4.2 Attributing feature models with testing data

Once the testing variability is captured in a feature model there are a number of prior approaches that can be used to reduce the number of products to be tested by applying, for instance, t-wise methods (Lamancha and Usaola, 2010; Perrouin et al, 2010, 2011; Johansen et al, 2012b; Oster et al, 2010). TESALIA uses feature attributes in the feature model to specify value and cost of tests to aid in pruning the testing space and later prioritizing the set of test cases to run.

Combinatorial explosion is a well known problem in software testing (Hartman, 2005). Because of the large number of software configurations, execution paths, and inputs, it is generally impossible to execute sufficient tests to completely test every execution path and input configuration of a software artifact. This problem also occurs in software product line testing because the number of software product line products is an exponential function of the number of features in the product line. One of the goals of software product line testing is to select the best set of test cases that maximizes the chance of detecting a bug while reducing the testing effort. To do so, the *cost* and *value* of a test case must be measured to quantify its importance. In this paper, we refer to the cost of a test case as a measure that quantifies the effort of executing a test case. This can be measured in terms of resources, such as budget, person-months, cloud computing time consumption, or other measures (Beizer, 2003). Additionally, the value of a test case is the measure that determines the importance of a test case. It can be measured in terms of number of tested features or market coverage (Zhu et al, 1997; Boehm, 2005).

To add cost and value information to the set of products in TESALIA there are two possible approaches. The first approach is to specify the cost/value information for each product by executing the tests associated with the product and storing their cost or value for further use. The second approach is to manually attribute the model with the relative cost or value of each feature. However, it requires

manually updating the information associated with each test and product, as well as updating this information as the software product line evolves. The cost and value of each product can be obtained by using a function that provides the cost of each product by using the values of the attributes present in the model. This can be done by, for example, extracting usage information from the Android play store in the case of testing mobile apps. Benefits and drawbacks of each approach should be determined by domain experts in each case. However, there are studies modelling product costs of cloud services (measured in monetary units) (García-Galán et al, 2013) or modelling space restrictions when designing circuit boards (measured in mm^2) (Tang et al, 2007).

In this paper, we use feature attributes to add cost and value information per feature. This information, together with value and cost *functions*, are used to automatically prune and prioritize the products to be tested as described in Sections 4.3 and 4.4.

Adding cost data → In order to optimize the number of tests executed, we use feature attributes to store the cost of the resources needed to run each test. Estimating testing costs requires both understanding the resource needs of the tests (e.g. in order to rent virtual machine instances in the cloud), as well as understanding the domain-specific properties (e.g. pricing model of the cloud provider for using those resources and time required by each test).

As stated in the previous sections, its expensive to execute every test on every mobile platform configuration. Therefore, it is necessary to select the mobile platform configurations that are worth testing. Knowing the cost of executing tests enables developers to evaluate the worthiness of executing those tests. Although appealing, the idea of adding value information may be considered infeasible in practice. However, we present a number of examples where test value can be practically determined (see Sections 3 and 5). For instance, testing an Android application that is running on an Android emulator will consume varying hardware resources, such as CPU, depending upon the emulator configuration.

When testing Android applications, users may want to test on as many valid hardware configurations (a.k.a emulator options) as possible. However, the number of potential emulator and device variations makes exhaustive testing infeasible. Further, because many organizations have limited development budgets, is important to provide mechanisms to maximize hardware coverage, in terms of feature coverage, given bounded testing resources.

Adding value data → Estimating the return on investment by executing a test is a difficult task. However, there are a number of scenarios where the benefits of individual tests can be derived. For example, when testing on mobile devices, the device market-share covered by the specified emulator configuration of the test can be used to estimate value. The greater the market-share associated with the platform configuration being tested, the more important the test is to execute.

An important attribute of the market-share driven approach to selecting mobile platform configurations to test on top of is that this approach can optimize the selection of platform configurations *with respect to the market-share coverage of the features that the product contains*. For example, if the five best selling Android devices are chosen for testing, these devices may be newer and have similar features to one another, such as large screens, that are not characteristic of the large installed base of previous generation phones. If screen size is a feature that has a direct impact on the product being tested, limiting the diversity of the screen sizes tested is not beneficial. Moreover, the product being tested will have a high likelihood of being installed on an older model device with a smaller screen size that it has not been tested on.

Value data may vary depending on external factors, such as new customers, new hardware requirements, or changes in the application design. TESALIA uses attributes to specify the value of testing each individual feature because they allow developers to update the value of each feature easily.

Defining the cost and value function → Referring to the motivating scenario presented in Section 3, an Android emulator with an extra high pixel density (high dots per inch [dpi]) screen uses substantially more processing power on the host computer than an emulator with a low pixel density (low dpi) screen. However, configuration options cannot be considered independently. For example, two hdpi (high number of dots per inch) screen emulators with different Android versions (e.g. 1.6 and 4.0) will have vastly different CPU utilization due to improvements in the operating system, and it is possible for an Android 4.0 hdpi emulator to use less CPU on the host computer than an Android 1.6 ldpi emulator. Without running all possible emulator configurations, it is hard to determine how much hardware, such as CPU, each platform configuration will require on a specific Android emulator configuration.

Defining the value of each test is also a key step when trying to optimize the set of tests to run for a software product line. Again, TESALIA allows users to define the function to obtain the value

of testing each product. In the example presented in Figure 6, the function $\sum$ *Audio.value, Wifi.value, Bluetooth.value*, is used as the value of each product (for the sake of simplicity).

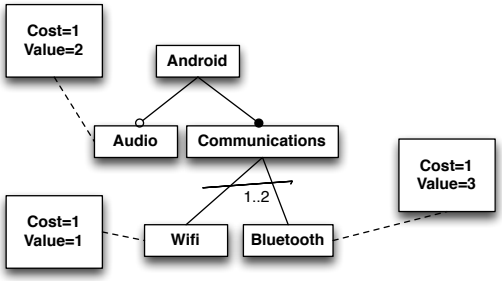

Fig. 6 Example based on the smartphone's motivating scenario.

Figure 6 shows the example feature model with cost and value data on its attributes.

4.3 Pruning the testing scope.

Testing on a large set of different product configurations can be very expensive, as shown in Section 3. When trying to reduce the number of tests, some products cannot be tested. To discard products when testing a product line, the majority of approaches described in prior literature apply t-wise coverage to each feature sub-set (Lamancha and Usaola, 2010; Perrouin et al, 2010, 2011; Johansen et al, 2012b; Oster et al, 2010). T-wise testing consists of testing all products that contain the unique possible combinations of the selected t features. For example, if we plan to apply 2-wise (t=2) testing on the feature model presented in Figure 6, every combination of tests involving unique combinations two features in the product line must be tested. On the other hand, if we plan to test by using 1-wise testing, every feature should be in at least one tested product.

When using TESALIA, any function can be used to define product testing value, since TESALIA is based on an underlying CSP solver. TESALIA can use the value of the cost attribute defined in each feature to obtain the cost of the product or can use the cost value to simply detect if a feature is present and then apply any t-wise based function to obtain the subset of products to be tested.

In order to automatically prune products for testing, TESALIA applies prior work on automated CSP-based derivation of feature model configurations to derive configurations sets (*e.g.*, mobile platform configurations to test on). Moreover, the configuration sets optimize a function over the features selected for testing without exceeding a testing cost budget(*e.g.*, cloud computing related costs). Once these products have been derived from the feature model, standard testing tools, such as JUnit, can be used to deploy and test a product in the cloud on the derived platform configurations.

A CSP is a set of variables and a set of constraints governing the allowed values of those variables. A set of values for the variables is called a *labeling*. A valid labeling of the CSP's variables requires that all of the constraints in the CSP are satisfied. For example, in the CSP "$0 < x < y$," where x and y are integer variables, $x = 4, y = 5$ is a valid labeling of the variables. Often, CSPs are used to derive variable labelings that maximize or minimize an objective function. For example, $x = 1, y = 2$ is a solution that minimizes the objective function $x+y$. A key attribute of CSPs is that automated tools, called constraint solvers, can be used to programmatically derive labelings of a CSP's variables.

The first step in deriving a set of products is to be able to derive all products from the feature model that do not exceed the testing budget. TESALIA builds on prior work for automated derivation of feature model configurations using CSPs to derive each individual platform configuration (Benavides et al, 2010). Note that we present the pruning of feature models as a new automated analysis operation extending the 30 existing approaches in the software product line community Benavides et al (2010). Given a feature model, a set of feature costs, a function to obtain the product cost and a maximum cost, the pruning operation returns a set of products that does not exceed the maximum cost provided. The TESALIA CSP model for deriving a single platform configuration from the feature model to test on top of is constructed as follows:

CSP model used by TESALIA to prune the products that exceed the maximum budget:

$$TESALIA =< F, FC, A, AC, f(AC), CC >$$

where:

- F is a set of variables, $f_i \in F$, representing the selection state of each feature in the product line feature model. If the i_{th} feature is selected for a product test, then $f_i = 1$ and, otherwise, $f_i = 0$.
- FC is the set of constraints that define the different relationships between different features (e.g. if the i_{th} feature is a mandatory child feature of the j_{th} feature, then $f_i <=> f_j$) according to the mapping presented by (Benavides et al, 2010).
- A is the set of attributes describing the estimated costs required to include each feature as part of a test product, where $a_i \in A$ is the cost to include the i_{th} feature in a product to be tested. Note that depending on the testing scope the cost can be defined in different ways (*e.g.*, cost to execute the test in a machine, cloud related cost for testing, ...).
- $ac_i \in AC$ is a variable describing the current cost contributed by the i_{th} feature to the currently selected product. The cost of a feature is only incurred if it is selected, thus: $ac_i = f_i * a_i$.
- $f(AC)$ is a function that calculates the total estimated cost of testing the currently selected product. Several different functions can be used depending on the objective to maximize. For example, if we use $\sum_{i=0}^{n} ac_i$ to calculate the sum of each feature cost in a product we can specify not to exceed a concrete budget.
- CC is the maximum allowed cost for testing a product, thus: $f(AC) \leq CC$.

In order to ensure that only valid products are derived, a few additional constraints must be added. The set of variables, F, is used to describe the selection state of each feature. Initially, the root feature, f_0 of the feature model is labeled with value 1 (*e.g.*, indicating that the root feature is selected). Because the product being tested will typically require a subset of the features, such as minimum platform version or a rear-facing camera, these additional constraints are encoded into the CSP by labeling the corresponding variable f_i of a required feature with value 1 and the corresponding variable f_k of a deselected feature with value 0. Pre-labeling these feature variables removes them from consideration by the constraint solver and reduces the configuration space size. In other words, is possible to start with a partial configuration that contains the basics for making our product meet a minimum requirements, therefore reducing the computing power required by the CSP solver.

Once the initial labeling of the CSP has been performed, a constraint solver is used to derive values for the remaining variables in F, such that none of the feature model constraints encoded in the CSP are violated (an extensive discussion of encoding feature model constraints in CSPs is available in (Benavides et al, 2010)). Moreover, the CSP includes the additional constraint that the total cost of testing on top of the selected platform configuration, described by the labeled variables in F, does not exceed the testing budget CC. The result of this automated labeling process is that the constraint solver produces a set of values for the variables in F, where the features that have been selected for the platform configuration are labeled with value 1 (*e.g.* if the feature f_i is selected then $f_i = 1$). The variables in F that are labeled with value 1 represent the configuration that has been derived for testing.

Applying this pruning method to our model presented in Figure 4 requires transforming the feature model into a CSP by using the following assignations of variables.

- F is the set of variables composed by Android, Communication, Audio, Wifi, Bluetooth, and Product.
- FC is the set of constraints representing the following relationships, Android **mandatory** Communication, Android **optional** Audio, Communication **set with cardinality 1..2** Wifi and Bluetooth.
- A is the set of attributes describing the estimated costs, in this example the features Audio, Wifi and Bluetooth have a cost of one $ per each one. For the sake of simplicity we are using dollars as a cost unit, however, other metrics such as CO2 emissions can be used.
- $f(AC)$ for this example we are using the sum of cost as a function to obtain the total cost of a product.
- CC is the maximum allowed cost, for the illustration purposed we fixed it to 3.
  This CSP returns all products having a $ cost less than 3, thus, filtering the products to test. In Table 2 all products with their cost are presented. Note that the gray line are products not returned by the TESALIA solution.

Name	Android	Communication	Audio	Wifi	Bluetooth	Product Cost
P1	✓	✓	✓	✓		2
P2	✓	✓	✓		✓	2
~~P3~~	~~✓~~	~~✓~~	~~✓~~	~~✓~~	~~✓~~	~~3~~
P4	✓	✓		✓		1
P5	✓	✓			✓	1
P6	✓	✓		✓	✓	2

Table 2 Sub-set of products applying the optimization function showed in Section 4.3.

4.4 Prioritizing the tests list.

There are circumstances where knowing which tests fit a bounded cost is not enough and it is also important to prioritize the set of tests to execute. Having tests prioritized, allows testers to begin with the tests that provide more value, and thus, obtain more testing benefit in less time. TESALIA uses the value-attribute of each feature and a function to determine how much benefit is obtained by testing a product.

TESALIA requires developers to define a function to calculate the value of each product. This function should be adapted to each testing context. For example, if we want to value tests based on the number of features involved in a product, the value function would be $\sum f_i$. If we want to increase the market-share coverage of our tests we should attribute the feature model with market-share information and, thus, specify our function as $\sum value_i$ where $value_i$ is the value attribute of a feature present in the product.

CSP model used by TESALIA to prioritize the set of products to test:

$$TESALIAPrioritization =< V, VA, VAC, f(VAC) >$$

where:

- V is a set of variables and constraints presented in Section 4.3.
- VA is the set of attributes showing the estimated value obtained by including or not, each feature as part of a product, where $va_i \in VA$ is the value obtained by adding the feature i in a product to be tested. Different value functions may have more or less sense depending on the testing scenario we were working on.
- $vavar_i \in VAC$ is a variable describing the current value contributed by the i_{th} feature to the currently selected product. The benefits of having a feature on a product only appears if it is selected, thus: $vavar_i = f_i * va_i$.
- $f(VAC)$ is a function that calculates the total estimated benefits obtained by testing the currently selected product. Again, different functions can be used to prioritize the test list. For example, if we decided to value our test by a market-share based function, the value of each product can be represented by the product of the relative market-share of each feature.

Going back to the small example presented in Figure 4, and applying the CSP presented in this section we obtain a pruned list of products with a total value. Table 3 shows the list of products with its value attached. For doing so, we need to use the following values for each problem component.

- V is a the set of variables used to prune the solution space that ended by returning the set of products presented in Table 2.
- VA are the value attributes for each feature as shown in Figure 2.
- $vavar_i \in VAC$ is a variable that contains the relative benefit that is provided by testing the i_{th} feature. The benefits of having a feature on a product only appears if it is selected, thus: $vavar_i = f_i * va_i$. In this example we are using market-share coefficients as values for the tests.
- $f(VAC)$ is a function that calculates the total estimated value obtained by testing the currently selected product. For the sake of simplicity, we are using integers in the small example presented in Figure 4, therefore we can simply use the $\sum$ function.

For example, the following prioritization maximizes the value of the tested configurations: P2, P6, [P1,P5], P4. Note that the products P1 and P5 can both be executed in any order because they have the same assigned priority. By executing the products in this order, the tester knows that if testing is interrupted for some reason, the tests that will have been run are the highest value tests.

Name	Android	Communication	Audio	Wifi	Bluetooth	Product Cost	Product Value
P1	✓	✓	✓	✓		2	3
P2	✓	✓	✓		✓	2	5
P4	✓	✓		✓		1	1
P5	✓	✓			✓	1	3
P6	✓	✓		✓	✓	2	4

Table 3 Sub-set of products with testing value.

4.5 Packaging the most profitable set of products.

After pruning and prioritizing the sets of product that meets budgetary and product-line constraints, being able to automatically derive the group of tests that maximize the value while keeping the costs bounded to a maximum cost is appealing. Note that the implementation of this knapsack is not multi-objective as the maximum cost is fixed. Thus, this solution will return the best value possible for a concrete maximum cost. Also, this operation requires both value and cost functions to calculate the weight and value of each element (product) in the knapsack.

For example, suppose that TESALIA has pruned the sets of possible configurations that do not exceed a 20$ cost and TESALIA also provided the value of testing each product. In such as situation, how can we maximize the testing value of a 30$ cloud testing budget? To derive multiple products, TESALIA starts by discarding those products that exceed the total budget by themselves. This reduces the computational effort to build the product sets. Later, each derived product is represented as a new unique labeling of the variables in F. Once this set is derived, TESALIA constructs a knapsack problem to select the optimal subset of these mobile platform configurations to test on top of to maximize market-share

The knapsack problem is a classic problem in computer science. Several studies have proved that it is an NP-complete problem (Martello and Toth, 1990; Coffman Jr et al, 1996). In TESALIA we are using the 0,1 approach based on dynamic programming to solve it (Martello and Toth, 1990). However, heuristics methods have been proposed to reduce the computational complexity of solving knapsack problems, such as the modified heuristic (M-HEU) algorithm (Akbar et al, 2001). The formal definition of the problem in TESALIA is defined as follows:

$$TESALIAmulti =< Sp, Cp, CMax, f(VAC) >$$

where:

- Sp is a set of variables describing the selection state of each possible configuration that could be tested. If the i_{th} possible product in P is selected for testing, then $Sp_i = 1$, if it is deselected then $Sp_i = 0$, where $Sp_i \in SP$. Sp define the set of elements to introduce in the knapsack. This is used to discard those products that exceed the maximum allowed budget by themselves.
- Cp is the cost of executing a test on top of the i_{th} platform configuration in P (*e.g.*, the derived result of $f(AC)$ from the previously described pruning step). This cost $f(AC)$ represents the volume in the knapsack problem.
- $CMax$ is the maximum allowed cost of executing the set of tests on the selected platform configurations. $CMax$ represent the maximum volume in the knapsack.
- $f(VAC)$ is the set of values per each product that represents the value of testing our product. As described in previous sections, one concrete metric for determining the value of a product is by using the market share it represents.

The goal of using this knapsack problem formulation is to derive the set of platform configurations that maximize the value of the test-suite being tested. This objective function is encoded as:

$$Maximize(\sum_{i=0}^{n} Sp_i * f(VAC)_i)$$

Nevertheless, because developers have a limited testing budget, an additional constraint to bound testing cost should be considered:

$$CMax \geq \sum_{i=0}^{n} Sp_i * f(VAC)_i$$

Referring back to our simplified example presented in Figure 6, the results of deriving a set of products to test that fit within a budget of 6 cost units are shown in Table 4. The best combination of products to test for the given budget is captured in *G3*.

Group Name	Products set	Set Cost	Set Value
~~G1~~	~~{P1,P2,P4,P5}~~	~~6~~	~~12~~
~~G2~~	~~{P1,P2,P6}~~	~~6~~	~~12~~
G3	{P2,P4,P5,P6}	6	13

Table 4 Optimized test products for a budget of 6 cost units.

5 Validating TESALIA

Developers for Google's Android smartphone platform are increasingly facing the challenge of fragmentation, whereby a growing number of distinct Android hardware/software configurations are forcing developers to test their application much more extensively than desired. For example, a report by OpenSignalMaps [5] found that their application was installed on 4000 unique device configurations, which would require significant testing time and effort to completely cover.

Due to the large number of in-use platform configurations, it is difficult for developers, regardless of development team size or proficiency, to test their software products on all or even most platform configurations before release. A balance of comprehensive testing and reasonable consumption of testing resources (e.g. money, time, computation power, storage, etc) is needed.

In this section, we use TESALIA to address this challenge in the context of the Android ecosystem. Figure 7 shows the different steps TESALIA takes to optimize test selection, including examining product variability, product computation cost, and product market share to identify valuable testing configurations. In order to perform this analysis, we have represented the configuration of the Android platform using a software product line, with the Android emulator being one possible product feature. By using a software product line to represent many different mobile platform features of an Android device, such as such as screen resolution or communication capabilities (*e.g.*, 3G, LTE, etc.), we were able to easily apply TESALIA and optimize the test selection. Using the current Android emulator, which contains 46 features, we can test up to 2^{46} unique platform configurations. Note that, we conservatively calculate the maximum number of unique platform configurations by using binary options and mutual exclusion.

Figure 7 shows the process we performed to validate our approach. First, we describe how we extracted value and cost information (market share and cloud usage) and how we used it to attribute a feature model describing the Android emulator options. Later we go through the four different experiments we developed to test the prioritization, pruning and packaging capabilities of TESALIA.

5.1 Experimentation Data

In order to execute the analysis operations presented in Section 4 a feature model with cost and value information needs to be built. To build this model, we used the Android emulator as the basis of our analysis. First, we encoded the variability present in the emulator options. Second we extracted cost value from the costs of executing a test in a cloud. Finally, we obtained publicly available market share data to create value attributes [6]. In the rest of this section we provide detailed information about how we built the inputs for the three experiments shown in Section 5.2.

5.1.1 Representing All Valid Android Testing Configurations Using a Feature Model

Feature models describe individual features, or units of functionality, using a tree-like structure, as shown in Figure 8.In the model presented in Figure 8 abstract features have been used to group the characteristics of the different mobile platform configurations defined in the Android ecosystem.

[5] `http://opensignal.com/reports/fragmentation.php`

[6] `http://developer.android.com/tools/help/emulator.html`

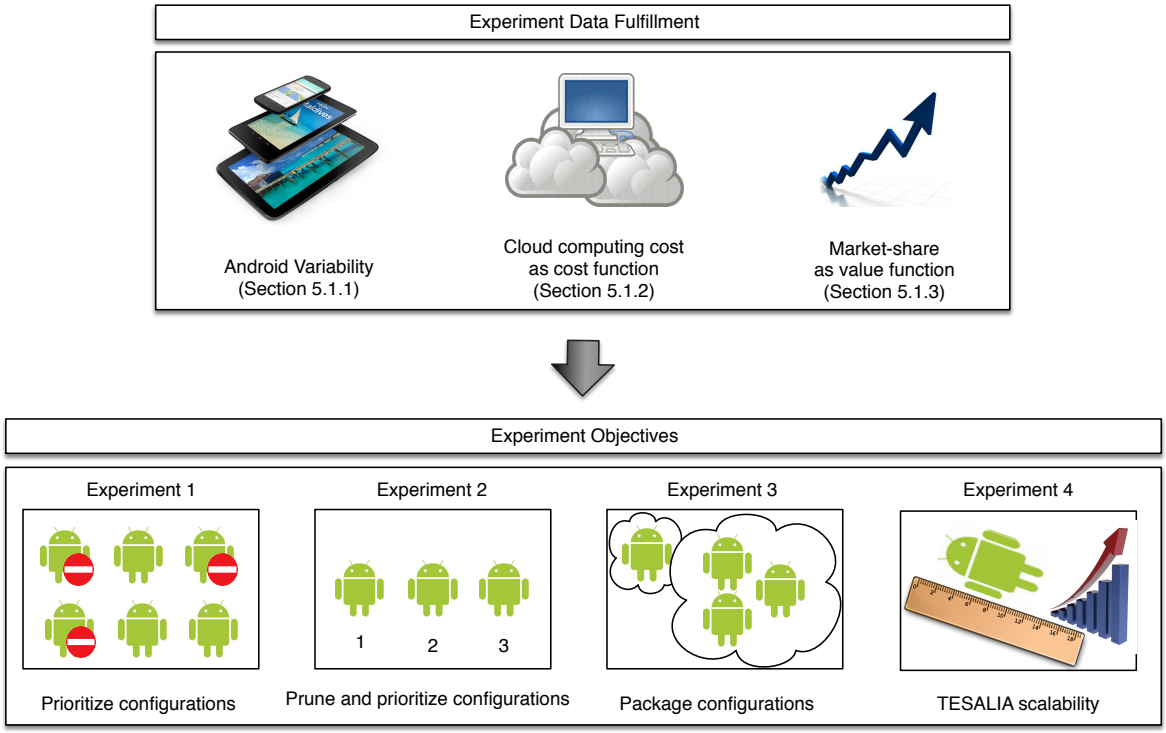

Fig. 7 Steps for optimizing the testing of android applications using TESALIA.

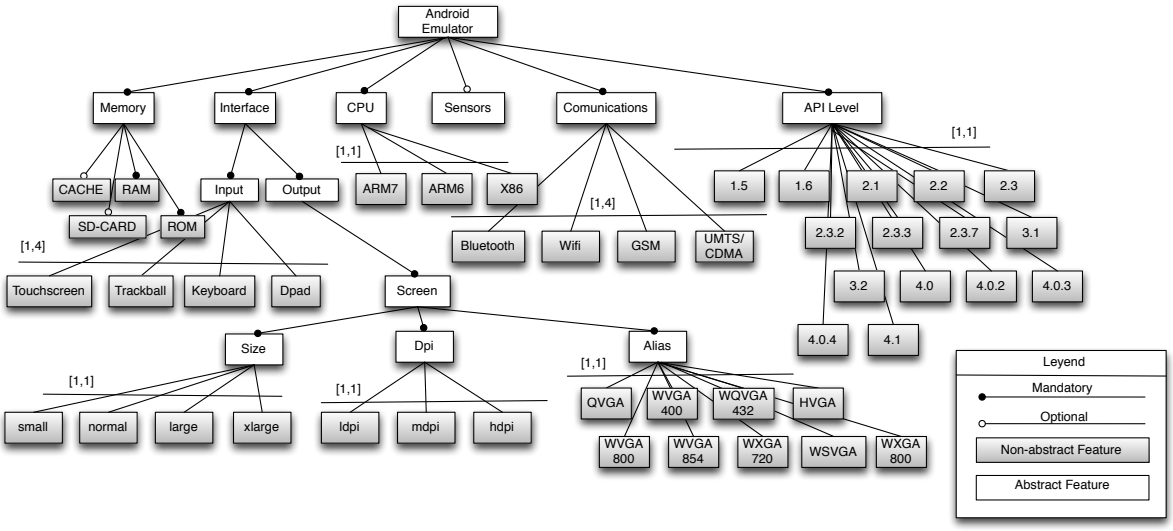

Fig. 8 An Android feature model.

Using a feature model of a mobile software platform's configuration rules, the goal is to select platform feature configurations that adhere to the constraints encoded into the feature model. An important property of feature models is that there are well-known approaches for automatically deriving valid or optimized feature configurations from a feature model by converting the model to a constraint satisfaction problem (CSP) or a SAT problem.

TESALIA uses the CSP formulations presented in Section 4 to automatically prune, prioritize, and package the set of platforms configurations to test. Figure 5 shows the general process carried out by TESALIA. Moreover, users can scope the testing process by annotating features that have to be present or not in the products. Also, users may want to add 'complex' constraints representing any other

exclusion between mobile platform features or costs related to the model, such as *"If the mobile platform configuration contains Android 1.5, then the mobile platform configuration should not contain more than 250mb of RAM memory."* Note that those annotations are not present in the feature diagram presented in Figure 8. They are inputs for the different operations and may vary depending on user preferences. The feature model representation allows us to exclude invalid platform configurations (a.k.a software product line products) and to dynamically construct a CSP for pruning, prioritizing, and packaging the products to test. In order to provide input for the operations presented in Section 4, we created a feature model representing the various options available to the Android emulator by extracting the rules outline in the emulator documentation. We included cross-tree constraints to ensure that no incorrect configurations are generated. As mentioned in Section 4.1, there are automated mechanisms that enable the extraction of feature models from a set of valid product configurations. However, when building the Android feature model, we found it simpler to identify constraints instead of listing valid configurations and therefore chose to directly create the feature model.

5.1.2 Using feature count as a proxy for the cost of execution in a cloud environment.

Cloud computing, which is promoted as a low-cost, effectively infinite computing solution, enables the average developer to temporarily rent computing resources powerful enough to test hundreds of unique platform configurations. Running such a test suite was previously impractical due to the prohibitive cost of assembling and maintaing computing equipment dedicated for testing. Commoditized cloud computing allows developers to perform large-scale application testing on multiple platforms with a minimal amount of added development time and zero equipment investment overhead.

Renting multiple virtual machines in the cloud can be expensive. To define the cost of testing a software product in a cloud, a tester must consider items such as the time required to test each configuration, the number of configurations under test, and the number of cloud virtual machines that must be rented. In Android, many platform features have complex non-linear interactions with other platform features, which affect both product performance and the resources needed to execute and test the Android emulator product. Modeling the exact computation time required to execute a single Android emulator test is beyond the scope of this work. Additionally, modeling the cloud resource usage of a subset of configurations, such as all configurations that use Android 1.6, by manually calculating the value of a few fully specific configurations. For example, {Android 1.6, screen size 240x320, 16MB RAM, has 'volume' hardware buttons, does not have 'Home' hardware button, etc} is difficult and also beyond the scope of this work.

To define the cost of executing an emulator instance in a virtual machine, we count the number of features included in each product. This means that instances executing more options (e.g. instances which are likely to consume more resources), have more impact in the calculated cost of a product. The $\sum$ function is used to calculate the cost of each product by making the total cost of a product equal to the number of concrete features composing that product.

5.1.3 Market-share as a value function

A key aspect of TESALIA is that it can rely on different value-functions, such as market-share data $f(VAC)$, for each individual mobile platform feature configuration. Although the process for obtaining this data is beyond the scope of this paper, two potential approaches are briefly outlined. First, a number of commercial vendors provide access to fine-grained mobile platform market share data, such as comscore [7]. A second, more commonly used approach by application developers, is to directly instrument their application and collect platform configuration data. For example, Skype keeps track of all known platform configurations that their software has been executed on. In general, application developers may find it most effective to use data that they have obtained from instrumenting their application, since it provides an accurate picture of the market-share of platform features in-use by the consumers of the developer's application rather than the market in general. Figure 9 shows an example of the market share data that was obtained from Google [8] and used for the experiments in this paper.

[7] `http://www.comscore.com`

[8] `http://developer.android.com/about/dashboards/index.html/`

An important attribute of the market-share driven approach to selecting mobile platform configurations for testing is that this approach can optimize the selection of platform configurations *with respect to the market-share coverage of the features that the application actually uses*. For example, if the five best selling Android devices are chosen for testing, these devices may be newer and have similar features to one another, such as large screens, that are not characteristic of the large installed base of previous generation phones. If screen size is a feature that has a direct impact on the application being tested, limiting the diversity of the screen sizes tested is not beneficial. Moreover, the app being tested will have a high likelihood of being installed on an older model device with a smaller screen size that it has not been tested on.

Because the mobile platform feature model can be pruned to focus on only the features that directly impact the application, it is possible to derive mobile platform configurations to test on that maximize the market-share coverage with respect to the subset of features that matter to the application. For example, a developer can choose to only support newer devices with Android version 4.0 and higher by pre-labeling the corresponding feature variable in F with value 1 (*e.g.*, requiring every derived mobile platform configuration to include it). In this case, the developer can maximize the market-share coverage with respect to the devices that actually have the supported platform version.

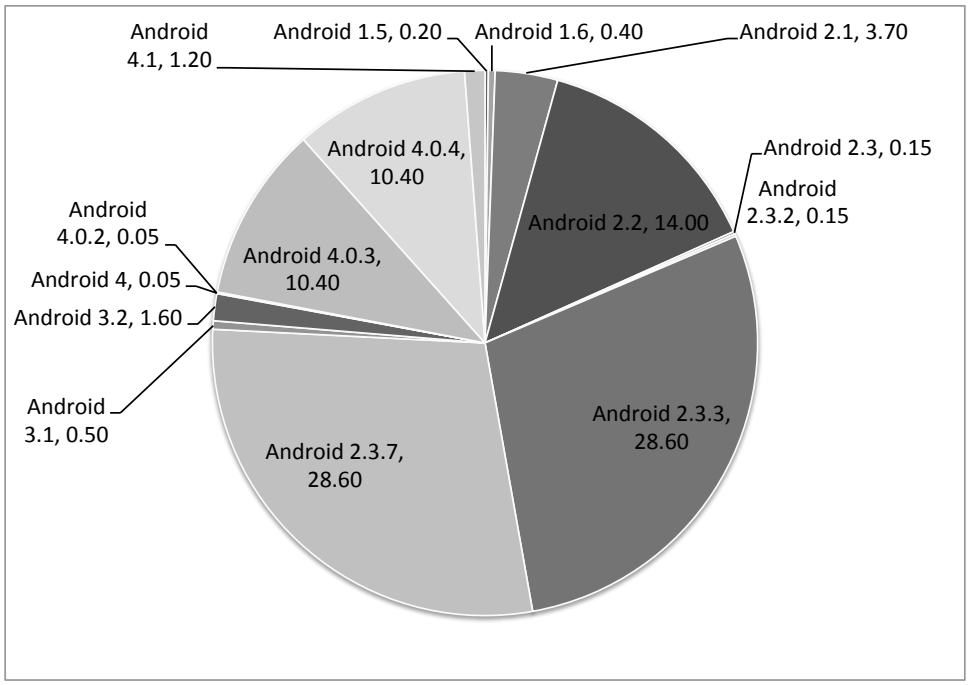

Fig. 9 Market-share example data by September'12.

The market-share data used in the experiments was freely obtained from Google Android market as a whole rather than for a specific application. We focused on a subset of the market-share data related to screen size, resolution, and Android version, since these are mobile platform features that impact the vast majority of Android apps. For other platform features, the experiments used a uniformly distributed market-share valuation. For example, the market-share data of the communication options is fixed to $1/numberOfAvailableOptions$, and with four options (GSM, CDMA, Bluetooh and Wifi) every option will be fixed to 25% market share. TESALIA generally performs better in a real-world non-uniform scenario, as optimized mobile platform selections have a substantially greater market-share than non-optimized choices in this type of model. We chose to use the uniform model because we had incomplete data on some platform features.

5.2 Experiments

Four different experiments were conducted to evaluate the TESALIA approach. These experiments compare and analyze: 1) the market-share prioritization of TESALIA-derived mobile platform configurations for testing versus the market-share coverage of a set of the most popular phones on Amazon; 2) a cost/benefit analysis of the configurations produced by TESALIA versus selecting a set of popular mobile phones using the pruning and prioritization operations; 3) the TESALIA validity for packaging configurations using the Android feature model attributed with Google market share; and 4)the TESALIA scalability with different model sizes and topologies.

Hypotheses of Experiment 1	
Null Hypothesis (H_0)	TESALIA does not optimize the percentage of market share covered features versus the traditional approach of buying the most sold phones.
Alt. Hypothesis (H_1)	TESALIA optimizes the percentage of market share covered features versus the traditional approach of buying the most sold phones.
Dependent variable	The market share coverage.
Blocking variables	The most sold phones and the market share indexes.
Model used as input	*Android feature model presented in Figure 8*
Hypotheses of Experiment 2	
Null Hypothesis (H_0)	The use of TESALIA will not result in a higher market-share impact metric than selecting the most commonly sold phones, for a given maximum budget.
Alt. Hypothesis (H_1)	The use of TESALIA will result in a higher market-share impact metric than selecting the most commonly sold phones, for a given maximum budget.
Model used as input	*Android feature model presented in Figure 8*
Blocking variables	The most sold phones, market share indexes and the maximum cost allowed set to 600$.
Model used as input	*Android feature model presented in Figure 8*
Hypotheses of Experiment 3	
Null Hypothesis (H_0)	TESALIA cannot derive the products to test in a reasonable time.
Alt. Hypothesis (H_1)	TESALIA is able to derive the products that maximize the value function in less than 5 minutes.
Model used as input	*Android feature model presented in Figure 8*
Hypotheses of Experiment 4	
Null Hypothesis (H_0)	TESALIA cannot cope with models having 2000 hundreds features.
Alt. Hypothesis (H_1)	TESALIA scales up to models having 2000 features.
Blocking variables	Number of features: 10,20,30,40,50,100,200,300,500,1000,2000 Percentages of cross-tree constraints: 5%, 10%, 15% Number of complex constraints: 1,2,5
Constants	
CSP solver	*ChocoSolver v2*
Heuristic for variable selection in the CSP solver	*Default*

Table 5 Hypotheses and design of experiments.

Table 5 shows the hypothesis of the experiments executed to validate our approach. To make the experiments reproducible, a number of fixed assumptions are made, such as homogeneous feature costs. ChocoSolver [9], with it's default heuristic, is used as the CSP solver for extracting software products from the feature model presented in Figure 8

Experimental platform - Virginia Tech ATAACK Cloud. The experiments were conducted using a version of TESALIA implemented in Java. Further, TESALIA was installed in the Virginia Tech ATAACK Cloud, which is a cluster testbed capable of simultaneously testing many configurations of an Android application on 1,000+ Android Emulator instances. The ATAACK Cloud runs on 34 dual-CPU Dell Blades with Intel Xeon X3470 CPUs running at 2.93GHz, with 16 threads per CPU, and CentOS v6. Each dual-CPU Dell Blade has 36GB of RAM.

5.2.1 Experiment 1: Market-share based prioritization.

In order to analyze TESALIA's market-share optimization capabilities, we designed an experiment to compare the market share of TESALIA-derived configurations with that of the 20 phones with the highest sales volume on Amazon. Our hypothesis is that simply selecting a set of the most-sold phones will not provide the best market-share coverage and that TESALIA's solutions will provide more market-share coverage. For this experiment, we obtained market share data from Amazon [10]. For the mobile platform configurations, we look at the 20 first configurations recommended by TESALIA using screen

[9] `http://www.emn.fr/z-info/choco-solver/`

[10] Amazon provides the list of the top-selling phones in the United States as of December 12 (`www.amazon.com`)

size and density mobile platform features. Note that only screen market share data have been used for this experiment.

Phone Name	Screen size	ddi	dpi category	Screen size category	MS coverage
Samsung Galaxy S III	1280x720	306	XHDPI	LARGE	3,6
Samsung Galaxy Note II	1280x720	267	XHDPI	LARGE	3,6
Samsung Galaxy S5830 Galaxy Ace	480x320	164	MDPI	NORMAL	11
HTC Droid incredible	800x480	252	XHDPI	NORMAL	25,1
Samsung Y Galaxy S-5360	320x240	133	LDPI	SMALL	1,7
Samsung Galaxy Nexus	1280x720	316	XHDPI	LARGE	3,6
Samsung Galaxy SIII mini	800x480	233	HDPI	NORMAL	50,1
Samsung Galaxy sII	800x480	219	HDPI	NORMAL	50,1
HTC A9192 inspire	800x480	217	HDPI	NORMAL	50,1
Motorola Atrix mb860	960x540	275	XHDPI	NORMAL	25,1
Sony xperia U ST25A-BW	854x480	280	XHDPI	NORMAL	25,1
Dell Aero	640x360	210	HDPI	NORMAL	50,1
HTC EVO 4g	1280x720	312	XHDPI	NORMAL	25,1
Samsung Galaxy Y Duos	240x320	127	LDPI	SMALL	1,7
Samsung galaxy gt-s7500 ACE plus	480x320	158	MDPI	NORMAL	11
Google nexus 4	1280x720	318	XHDPI	LARGE	3,6
Samsung Galaxy ace 2	480x800	246	HDPI	NORMAL	50,1
Sony xperia play	480x854	245	HDPI	NORMAL	50,1
HTC freestyle f5151	480x320	180	MDPI	NORMAL	11
Motorola droid 2	480x854	265	XHDPI	NORMAL	25,1

Table 6 Twenty Amazon most sold phones (December'12).

	ldpi	mdpi	hdpi	xhdpi
small	1.7%		1.0%	
normal	0.4%	11%	50.1%	25.1%
large	0.1%	2.4%		3.6%
xlarge		4.6%		

Table 7 Google provided market-share (October'12).

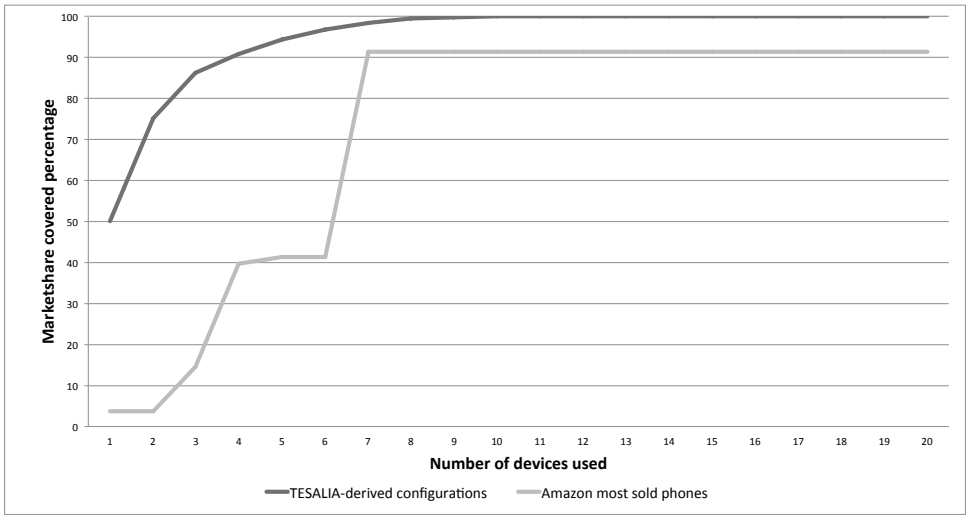

Fig. 10 Comparison of the market-share coverage of the five most sold phones on Amazon and five TESALIA-derived mobile platform configurations.

In the Table 6 we present the raw data for the twenty most sold phones. This table summarizes the screen size, the pixel density, the market share categories and the market-share coverage. Note that

only the prioritization operation has been performed in this experiment. Later in Table 7 we show the market-share data provided by google in October'12, which is the most recent market-share data that is freely available.

The market share coverage refers to the kinds of screen (combinations of dpi and size) covered by a concrete product and its associated market share value. Thus, having one configuration with a small and ldpi configuration will provide the 1.7% market-share coverage.

Results & analysis→ Figure 10 shows the market-share coverage of the first 20 most sold phones with respect to screen size and density compared with the market-share covered when buying the most sold phones in Amazon. We can see in the chart of Figure 10 that is not possible to reach the 100% market share coverage when relying in buying most-sold phones. Also, using TESALIA-derived configurations we reach the 90% of market-share coverage with only four configurations. Moreover, if we only take into account the five first phones we observe that the five TESALIA-derived configurations cover over 53% more of the market than the five most sold phones. The overall market-share covered buying the five most sold devices is 51.4% while the five most-scored TESALIA-derived configurations cover 94.4% of the market. In light of these results we can conclude that the benefits of using market-share prioritization enable developers to test their applications in front of those devices their customers use. Moreover, we observed that buying the flag-phones or most-sold phones is not a good idea when optimizing the testing of Android applications.

5.2.2 Experiment 2: Cloud cost and market-share optimization.

This experiment compares the cost per unit of market-share obtained from TESALIA-derived configurations versus the most sold phone sets from Experiment 5.2.1. Each phone was considered to have a cost of $600 in this experiment, which is typical of unlocked mobile devices. Then we calculated the cost of buying the N first phones and divided by the market-share obtained using them. This is,

$$\frac{600 \times NumberOfConfigurations}{marketShareCovered}$$

For example, for the first Amazon phone we would consider the cost as $^{600}/_{3.6}$, for the third $^{600\times3}/_{14.6}$ and so on. With this experiment we want to show the economical benefit of automatically derive the most valuable configurations in terms of market-share.

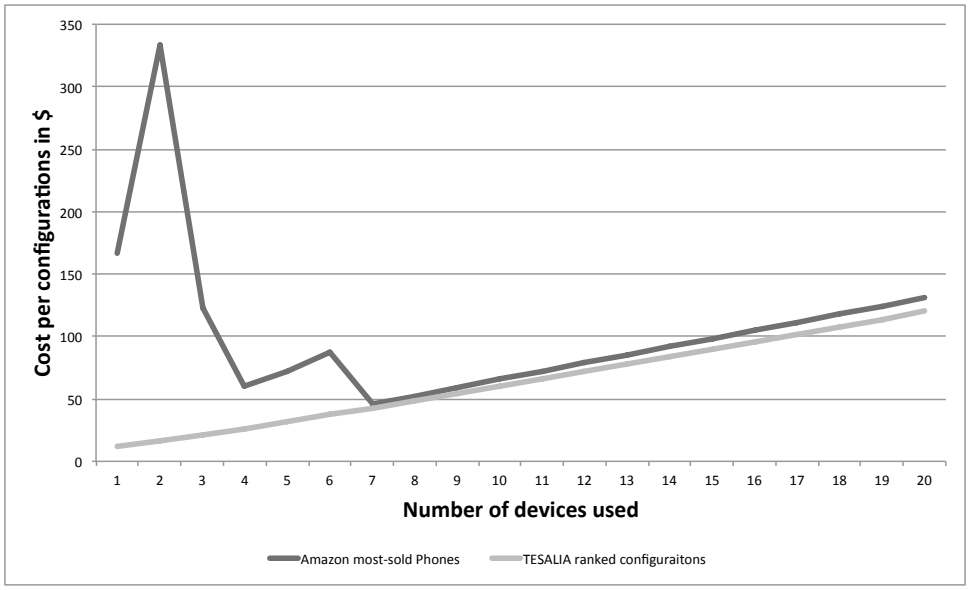

Fig. 11 Market-share coverage per monetary unit.

Experiment results. Figure 11 shows the market-share/cost when using TESALIA versus buying the N most sold phones. As can be seen in the experiments, TESALIA is more cost-effective than selecting

a set of the most popular phones to test on. Moreover, another interesting conclusion is that testing more than 10 of the most sold phones or TESALIA-derived configurations provides no additional benefit. The surplus saved when using the TESALIA approach could be invested either into additional testing on other mobile platform configurations or running extra tests on the same mobile platform configurations.

5.2.3 Experiment 3: TESALIA-packaging validity for the Android scenario.

A key question when using CSPs and solving NP-hard problems, such as Knapsack problems, is the scalability of the approach. The purpose of this experiment is to determine if TESALIA is scalable enough to solve market-share optimization problems for real-world mobile platform configuration spaces and market sizes. This experiment analyzes the time for TESALIA to derive all possible configurations of the Android platform that represent real devices and solve the Knapsack problem to optimally select a set of market-share maximizing configurations to test.

Experimental platform → This experiment was conducted on an Intel Core i7 processor running at 1.8GHz, with 4GB DDR3 RAM and the 6.0 Java Virtual Machine with command line options –Xmx250m and –Xms64m. The operating system was Mac OS v.10.7.1.

Results → Our experiments showed that TESALIA can find 29,510 configurations per second. Based on estimates of the total number of Android devices from OpenSignalMaps, we estimate that there are roughly 4,000-6,000 Android device models. Moreover, we attempted to overestimate the number of possible Android versions that could be on each device as 5, since newer devices probably have fewer versions of Android that can run on them and we consider 5 to be a reasonable upper bound on the number of potential different Android versions available for a given device model. Given this estimate, there could be up to 30,000 different valid platform configurations in the market, all of which could be derived by TESALIA in approximately 1 second.

We also tested the time to derive package the set of tests with 30,000 products. Solving this problem on our testbed took an average of 232 seconds on our test platform. We solved the Knapsack using an exact technique. Substantially faster approximation algorithms are available that give near-optimal results in substantially less time. These algorithms could also be used if desired.

5.2.4 Experiment 4: TESALIA scalability.

The scalability of TESALIA has been validated for its use in current Android feature model. However, is important to verify that it scales for larger problems. We tested TESALIA in front of two different models sets. First, random models and later, models from the SPLOT Mendonca et al (2009) repository.

Testing TESALIA with random models → The random feature models used vary between 10 and 2000 features, 0 to 15 % of cross-tree constraints and 0 to 5 complex constraints (A constraint involving attributes). To generate these models, we used the Betty tool (Segura et al, 2012) provides an implementation of an algorithm (Thum et al, 2009) for the feature model random generation. Later, we configured it to generate models having two integers attributes per feature (value and cost) in a range from 0 to 10. Also, to prevent possible threats affecting the experiments execution, we obtained 10 models per combination of inputs and present the average time. Moreover, non-valid models were discarded for this experiment. Finally, three different functions were defined for each testing scope. For pruning, we discarded the configurations exceeding a total cost greater than the 10 multiplied by the number of features. For prioritization, the function used is the sum of the value attribute. Finally, for the packaging scope, the total cost allowed in the knapsack problem is twice as the maximum cost for a product.

Testing TESALIA with SPLOT models → While the random models allow us to see the impact of the different variables in the execution time, they may not mimic the topology of realistic models such as those used in video-sequence generation domain(Galindo et al, 2014) or cloud computing usage(García-Galán et al, 2013). To determine if TESALIA scales in a realistic scenario we took all SPLOT models(Mendonca et al, 2009) and randomly added a cost and a value attribute per feature so that we could execute our optimization. We also added 0 to 5 complex cross-tree constraints per model. Attribute values ranged from 0 to 10 in the random models. Also, we generated 10 attributed models per combination of inputs and presented the average time. Again, we used the three same objective functions used for the random models.

Experimental platform → This experiment was conducted on of the blades of the ATAACK cloud in one thread. Concretely, the blade was using a dual-CPU Dell Intel Xeon CPU X3470 running at 2.93GHz CentOS v6 and the 6.0 Java Virtual Machine with command line options –Xmx250m and –Xms64m.

Results with random models → Our experiments show that TESALIA scales up to models with 2000 features without requiring more than 20 seconds of CSP computation for any of the analyses. Figure 12 shows the average time required in a logarithmic scale by each permutation of input variables along with maximum and minimum values (marked by points above the bars). In the worst case, the execution of the operations did not exceed 10 seconds, which we consider is sufficient for a wide range of applications. Also, in the figure, we can observe that the time required by TESALIA depends on the testing operation being used. That is, different operations perform differently. For example, summing the execution time for all input parameters combinations and pruning the tests takes 7290.5 milliseconds less than test prioritization.

Results with SPLOT models → After executing the three operations with the models available in the SPLOT repository we noticed that 7.421 milliseconds were required for the pruning operation, 8.446 for the prioritization, and 11.099 for the packaging operation. Figure 13 shows a small set of the models analyzed, which contains the most CPU consuming models, and 0 complex cross-tree constraints. For example, the most complex model in the repository – REAL_ FM_4 – took 524.5 milliseconds for the packaging operation. With these results in hand, we can conclude that our approach scales up to realistic feature model sizes, as well as to large random models. Moreover, we can infer that the real models are less complex for the operations presented in this paper. The data set with the time required for all SPLOT models can be found in the material website. We also noticed that the inclusion of complex cross-tree constraints reduces solving time as there are less configurations to explore.

5.3 Analysis and discussion

In this section we discuss the results we obtained and how the null hypothesis has been rejected, thus accepting the alternative hypothesis.

We performed four different experiments to check the validity of the TESALIA approach. With the first experiment we aimed to check the validity of the market-share metric when selecting smartphones configurations to test. The null hypothesis was that TESALIA does not improve the market share coverage compared to testing on the highest selling phone configurations. To refute the null hypothesis we compared the 20 most sold phone configurations from Amazon with the configurations produced by using TESALIA. Taking a look to the results we accepted the alternative hypothesis which indicates that TESALIA outperforms the traditional approach of testing on the most sold phones.

In the second experiment we compared the cost of the traditional approach of testing on the most sold phones versus the TESALIA approach. We compared the cost per configuration in dollars assuming that each phone costs 600$. In this case, the null hypothesis to refute is that it is cheaper to buy the most sold phones than using TESALIA. In this case, again for the 20 most sold phones, the experiments did not refute the null hypothesis leading us to accept the alternative hypothesis: TESALIA is more cost effective than the traditional approach.

The third and fourth experiments investigated the scalability, in terms of computing cost, of our approach. In each experiment, we determined the execution time of the approach when provided feature models of varying complexity and structure. In both experiments, we have been able to refute the null hypothesis and conclude that TESALIA can properly cope with realistic models described in prior literature. We noticed that the number of constraints is not impacting that much the analysis process. However, the less constraints the model has, a small increment in computing time is observed, specially in the minimum and maximum values. We suspect that this is caused because we are maintaining the same value and cost functions, thus, the less constraints the model has implies more products, making more costly to traverse all solutions.

In these experiments we have shown that TESALIA can cope with the constraints and variability complexity present in Android and in a variety of other models. Moreover, we showed the benefits TESALIA can offer. However, we need to check the viability of the TESALIA approach with industrial partners in future work. In current work, we have begun developing guides for developers to aid in

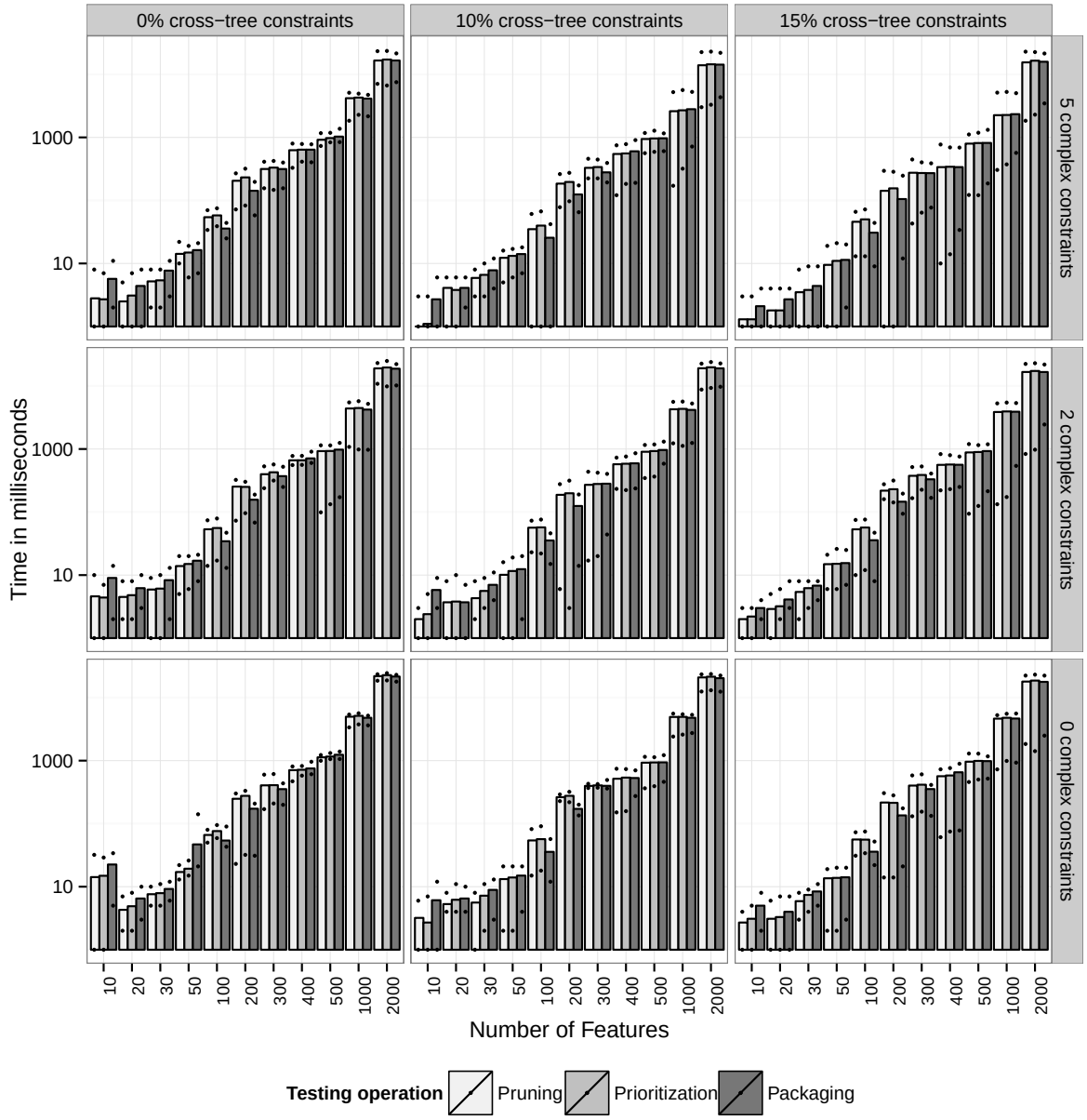

Fig. 12 Time required by TESALIA to prune, prioritize or package products encoded in feature models depending on the number of features, the percentage of cross-tree constraints and the number of extended constraints.

creating optimization functions (Alférez et al, 2014) for a variety of domains. Section 8 discusses our plans to integrate these techniques in TESALIA.

5.4 Threats to validity

Even though the experiments presented in this paper provide evidence that the solution proposed is valid, there are some assumptions that we made that may affect their validity. In this section, we discuss the different threats to validity that affect the evaluation.

External validity. The inputs used for the experiments presented in this paper were either realistic or designed to mimic realistic feature models. The Android feature model is realistic since numerous experts were involved in the design. However, since it was developed using a manual design process, it may have errors and not encode all device configurations. Also, the random feature models may not

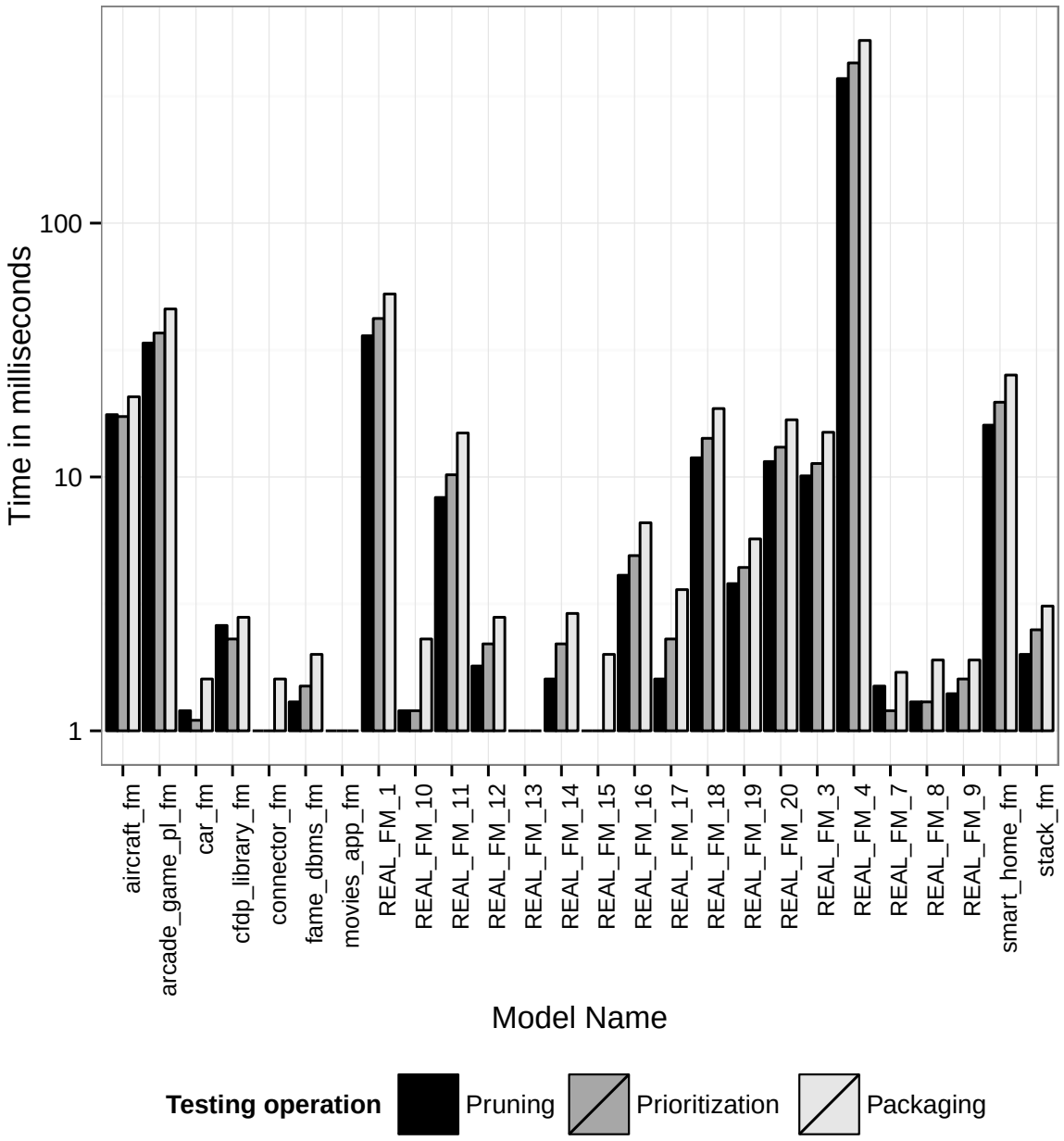

Fig. 13 Time required by TESALIA to prune, prioritize or package products encoded in SPLOT feature models depending on the operation executed.

accurately reflect the structure of real feature models used in industry. Another possible threat is the lack of market-share data affecting more features and the age of the market share data. The major threats to the external validity are:

– *Population validity*, the Android feature model that we used may not represent all valid Android configurations. Also, random models might not have the same structure as real models (e.g. mathematical operators used in the complex constraints), also the use of incomplete market-share data may have introduced errors. To reduce these threats to validity, we generated the models using previously published techniques (Thum et al, 2009) and using existing implementations of these techniques in Betty (Segura et al, 2012). Also, for the attributes generation, as there are no well-known values in

the literature yet, we introduced extended cross-tree constraints and attributes ranges consistent with past studies of feature models (Cordy et al, 2013) and (Passos et al, 2011)
- *Ecological validity*: While external validity, in general, is focused on the generalization of the results to other contexts (e.g. using other models), the ecological validity if focused on possible errors in the experiment materials and tools used. To prevent ecological validity threats, such as third party threads running in the virtual machines and impacting performance, the TESALIA analyses were executed 10 times and then averaged.

Internal validity The CPU resources required to analyze a feature model depend on the number of features and percentage of cross-tree constraints. However, there may be other variables that affect performance, such as the nature of the complex constraints used. To minimize these other possible effects, we introduced a variety of mathematical operators over the integers in the constraints to ensure that we covered a large part of the constraint space.

6 Related Work

This section compares TESALIA with related work, such as pair wise testing, cost related testing, value-driven testing, and other techniques used for mobile testing.

Software product line testing In order to reduce the cost of testing a software product line, researchers have presented several approaches for reducing the number of tests to execute, and therefore the cost of the entire testing phase. For example, (Colanzi et al, 2013) presented an overview of possible strategies for reducing testing costs. (do Carmo Machado et al, 2012) present a literature review of strategies for test software product lines. These works discuss the two principal mechanisms when testing software product lines–domain artifacts testing and application artifacts testing. This paper focuses on application artifacts testing, where combinatorial optimization has proved to be one of the most promising approaches.

Software product lines are an approach for building a configurable software platform that can be adapted to varying requirement sets. A key component of a software product line is that there are a set of common components, as well as points of variability that can be adapted to a given requirement set. Some research (Muccini and Van Der Hoek, 2003; Bertolino and Gnesi, 2003; Kästner et al, 2012) has investigated testing techniques that can be applied to software product lines. These software product line testing techniques provide varying algorithmic approaches to ensuring that the entire configuration space of the software product line is covered. For example, (Kästner et al, 2012) presented a method to reduce the efforts of executing a test over the whole set of products encoded in a product line. This is done by analyzing the model and only execute the test one time in the parts not affected by variability. This paper addresses the related but distinct problem of attempting to derive a set of the underlying software product line configurations to test on top of. Moreover, TESALIA focuses on optimizing the testing based on different value and cost functions.

Pair-wise testing. One of the most appealing approaches has been presented in (Cohen et al, 2006). Cohen et al. first explain the importance of creating good covering arrays for the set of products encoded in a product line, and later in (Cohen et al, 2008) presented specific algorithms to generate the covering arrays and provide empirical results. Several others have utilized this general method(Johansen et al, 2012a; Perrouin et al, 2011, 2010; Lamancha and Usaola, 2010; Cohen et al, 2008). For example in (Oster et al, 2010), authors prosed the use of CSP for generating pair-wise feature permutations. Our proposal is more ambitious than the one presented by Oster et al., by enabling not only the use of t-wise pruning but also more complex functions such as the cost of executing the tests in the cloud. These approaches can be complementary. For example, by using a piecewise function as the TESALIA cost function, we can discard some t-wise pairs with some reasoning over the attributes of the features involved in the product. Moreover, (Johansen et al, 2012b) focuses on value guided testing by generating covering arrays. In that approach, sub-product lines have different weights, thus some products are more critical than others when testing. TESALIA extends this approach by using cost and value encoded in feature model attributes.

Cost guided testing. The cost of software development is one of the main concerns when developing software. There are several proposals that present solutions to address the problem of testing costs (Oster et al, 2010; Binkley and Society, 1997; Perrouin et al, 2011; Withey, 1996; Sneed, 2009; Perrouin et al, 2010; Srikanth et al, 2005; Boehm and Sullivan, 1992; Rothermel and Hall, 1997; Boehm, 2005; Cohen et al, 2008). With the TESALIA proposal we are also able to minimize the testing cost while maximizing the value of our tests.

Configuration management & analysis. She and Lotufo et al. have investigated techniques for modeling the features and configuration rules embedded in the Linux kernel(She et al, 2010; Lotufo et al, 2010). In their work, they use similar feature modeling approaches to manage the variability in the Linux kernel. TESALIA also applies existing feature modeling techniques to manage mobile platform configurations and aid in testing mobile software. TESALIA also focuses on how to model the variability of configurable platforms and derive valid configurations. However, a key difference with TESALIA is that it is specifically focused on combining this configuration data with a market-share model in order to derive platform configurations for testing.

Multi-objective optimization problems. There are circumstances where it is important to balance trade-offs between different testing objectives. These problems are known as "multi-objective" problems. Most solutions for these problems are based on evolutionary algorithms (Deb, 2001; Coello Coello, 2006).

Recently there is a trend of applying these techniques to software product line testing. Because of their similarity with TESALIA, we have developed a deeper comparison between these approaches. The results obtained by this comparison are shown in Table 8. Concretely, we compare; i) if the approach uses multi-objective solving techniques. This is, if multiple functions, maybe conflicting, are used for the same testing operation; ii) the testing operation supported (pruning, prioritization and packaging); iii) if the solution allows user defined functions; iv) the support for quality attributes; v) the completeness of the solving techniques, this is if the approach explores the whole solution space or uses heuristics mechanisms, and vi) if the implementation is available and if supports complex cross-tree constraints (constraints over attributes) and attributes with ranges. Note that the techniques described here are not focused on product prioritization, pruning, or packaging but techniques that can be applied to other multi-objective problems such as the configuration of a product line. ....

(Sayyad et al, 2013) explored the existing evolutionary algorithms to find Pareto optimal solutions. The research found that IBEA algorithm provides the best performance when dealing with feature models. This work presented a comparison between different multi-objective solving techniques based on evolutionary algorithms and enables the prioritization of objective functions over features. The main difference of this approach compared to TESALIA is that instead of heuristic based methods, such as evolutionary algorithms, TESALIA uses a CSP and support attributes.

A different approach was presented by (Henard et al, 2013). In this work, authors proposed the configuration or generation of test-suites by using objectives over features. An interesting point of this approach is the generation of test-suites based on the pair-wise coverage they offer. Again, the main difference with TESALIA is the approach used (Evolutionary or CSP) and the attributes support.

Finally, (Olaechea et al, 2012) used an exact solver to obtain the set of products based on attributes values. The main differences between these approaches and our approach is the use of different solving techniques that explore the whole solution space. Another notable difference is that TESALIA allows for constraints that relate attributes to features. Note that none of the approaches explicitly support prioritization

Knapsack algorithms. Knapsack problems and bin packing problems have been studied for decades (Martello and Toth, 1990; Coffman Jr et al, 1996). In the solution proposed in this paper, an exhaustive solution based on dynamic programming has been used to select the set of hardware configurations that maximize market-share without exceeding a maximum budget when testing. Researchers have proposed other methods to solve the knapsack problem based on heuristics (Akbar et al, 2001). These methods provide a good approximation of the maximum value of the knapsack, but have much better algorithmic time complexity.

Value-driven development. Companies aim to maximize profit when developing software. Value-Driven development is a series of processes that companies can follow to focus development and testing

Approach	(Sayyad et al, 2013)	(Henard et al, 2013)	(Olaechea et al, 2012)	TESALIA
Multi-objective solver	●	●	●	○
Pruning support	○	⊘	○	●
Prioritization support	○	○	○	●
Packaging support	○	○	○	●
User-defined testing functions	○	○	●	●
Attributes support	○	○	●	●
Completeness	○	○	●	●
Implementation available	○	●	○	●
Complex constraints	○	○	○	●
Attributes with Ranges	○	○	○	●
● addressed as goal, ⊘ addressed but with restrictions, ○ not regarded as goal				

Table 8 Related work comparison.

to maximize profitability. There is previous research on this topic (Boehm, 2005; Boehm and Sullivan, 1992; Srikanth et al, 2005). (Sneed, 2009) proposed the use of a Return of investment (ROI) metric in order to guide the different stages of software development. The ROI metric provides a mechanism to quantify the profitability of developing a software product at various stages in the software development lifecycle. To maximize testing ROI, (Srikanth et al, 2005) proposed different ways to prioritize test-case execution. In this paper, the market-share metric has been proposed to maximize the number of users reached by the platform configurations that a mobile application is tested on. This approach is a specific type of ROI analysis for value-driven development and complementary to prior research.

7 Concluding remarks

In this section we present the lessons we learned while developing the TESALIA solution. A key challenge when testing a software product line is the large number of different products that can be encoded into it. In the mobile development industry, a similar problem exists, called platform fragmentation, where developers must deal with different features such as major versions of the mobile OS or varying screen sizes. In most cases, testing all possible platform configurations is not feasible, and developers therefore select a set of popular phones to test on. However, no studies have been performed to determine the effectiveness of this approach over alternative approaches.

This paper presents TESALIA, a framework for selecting SPL products (e.g. complete device descriptions) to test applications on. TESALIA extends prior work on automated analysis of feature models by deriving which products, or groups of products, an application should be tested on to provide the most value. Value is defined by a series of cost functions provided by the developers. We provide an example cost function that balances the desire to test software on platform feature configurations that cover the largest percentage of the mobile market (e.g. the current market) and the desire to keep testing costs low.

TESALIA requires that developers specify the set of configuration rules for a software product line as a feature model. Next, this feature model is used to i) derive all valid products (or Android platforms configurations); ii) prune the products by using a concrete cost function; iii) prioritize the products, by using a value function; and finally; iv) if desired, TESALIA is able to package sets of products that provide the most value for the given set of constraints. These products can then be used for testing. As shown by the empirical results, TESALIA can derive sets of mobile platform configurations to test mobile applications on that provide substantially better market-share coverage than simply selecting a set of the most popular phones. From this research we learned the following important lessons:

1. **Different functions for different scopes.** Depending on the function used to calculate test value, the value of the test-suite may vary. Traditionally, researchers have explored t-wise functions for optimizing SPL testing. However, in this paper we examine the importance of other functions, such as market-share coverage, to reach as many users as possible. Moreover, even using a market-share metric, we will obtain different results depending on the testing targets. For example, in the mobile industry, if a company only targets the Galaxy S, then the market-share indexes should give priority to the features of the Galaxy S. This shows the importance of allowing user input when select the

function to prune and prioritize configurations. In future work, we will explore metrics based on the target audience, such as a comparison between coverage when using market-share indexes or pairwise testing.

2. **Testing in fragmented ecosystems can be expensive.** In a fragmented ecosystem, the large number of mobile platform configurations may cause an exponential growth in testing complexity. In these ecosystems, prioritization of tests is a critical component of a successful testing strategy. In this paper, the use of a market-share metric has been proposed to prioritize test configurations for mobile development. However, to reduce the variability present in the model of the market, users should consider the target market of the application under test. For example, if the application aims to run on every possible mobile device, a global market-share should be used. In another case, if the target audience is the employees of a company, the market breakdown of mobile phone features should be specific to that company.
3. **New and fast selling phones do not guarantee excellent market-share coverage for testing.** Empirical results showed that a set of top selling phones from a given time period may leave substantial features of the mobile platform configuration space untested.
4. **App-specific market-share optimization is possible.** Many mobile apps use a remote reporting framework to send device statistics back to developers. For example, OpenSignalMaps tracks and published information on the number of unique device models that their app is installed on. By combining TESALIA with this type of app-specific market-share data, developers can derive optimized mobile platform configurations to test on for their specific app market characteristics.

8 Future Work

In this section we present the main directions of our future work related to the testing of SPL in the context of mobile applications.

Multi-objective comparison: Recently there is a trend of multi-objective testing based solutions for product lines. We plan to compare these methods with TESALIA and provide an in-deep analysis of the pros and cons of both approaches when testing Android apps.

Knapsack problems: While there are several implementations of knapsack solving algorithms, which ones work best on product-line problems is not known. In the future, we will review and analyze the use of knapsack solving strategies for SPL testing.

Testing on different Android distributions: TESALIA can be used to test applications by using the Android distribution provided by Google. However, there are other distributions such as CyanogenMod (`http://www.cyanogenmod.org/`), REPLICANT(`http://www.replicant.us/`) among other open-source projects. We plan to extend TESALIA analysis techniques to support the emulation of those Android forks. Thus, Android fork developers can detect errors present in their distributions but not in the official Google distribution.

Historical data considerations: We plan to extend TESALIA, so it can consider historical data. That is, we would like TESALIA to track the errors associated with features and use this information to aid in prioritizing future tests.

Obtaining larger market-share data and sales reports: TESALIA validation has been done by using Googles public and free market-share data. However, this data was not complete. Moreover, we also will need to obtain the most-selling phones information to compare. We are planning to obtain larger and more complete market-share data by crawling the Android play store and Amazon.

Qualitative evaluation: While in this paper we have shown that TESALIA offers technical advantages when coping with testing of Android applications, qualitative experimentation is still required for the modeling part of our work. This is, currently the feature model and constraints have been developed by domain experts but determining the difficulty to determine such constructs is still unknown. Others authors have already cope with modeling attributes such as Olaechea et al (2012). We plan to perform future work in the style as (Alférez et al, 2014) and (Acher et al, 2014) did, but with Android developers instead of video-sequence generation images. Moreover, we plan to integrate the facilities of the VM language (`http://mao2013.github.io/VM/`) with TESALIA for the sake of easy function definitions

Material

For reviewing purposes, TESALIA and associated test data is available from `http://tesalia.github.io/`. Please note, that the software is distributed under an LGPLv3 license.

Acknowledgements This work has been partially supported by the European commission (FEDER), by the Spanish government under TAPAS (TIN2012-32273) project and the Andalusian government under Talentia program, THEOS (TIC-5906) projects and COPAS (TIC-1867).

References

Acher M, Alferez M, Galindo JA, Romenteau P, Baudry B (2014) Vivid: A variability-based tool for synthesizing video sequences. In: 18th International Software Product Line Conference (SPLC'14), tool track, Florence, Italie, URL `http://hal.inria.fr/hal-01020933`

Akbar M, Manning E, Shoja G, Khan S (2001) Heuristic solutions for the multiple-choice multi-dimension knapsack problem. Computational Science-ICCS 2001 pp 659–668

Alférez M, Galindo JA, Acher M, Baudry B (2014) Modeling Variability in the Video Domain: Language and Experience Report. Rapport de recherche RR-8576, INRIA, URL `http://hal.inria.fr/hal-01023159`

Batory D, Benavides D, Ruiz-Cortes A (2006) Automated analysis of feature models: challenges ahead. Communications of the ACM 49(12):45–47

Beck K (2003) Test-driven development: by example. Addison-Wesley Professional

Beizer B (2003) Software testing techniques. Dreamtech Press

Benavides D, Segura S, Ruiz-Cortés A (2010) Automated analysis of feature models 20 years later: a literature review. Information Systems 35(6), DOI 10.1016/j.is.2010.01.001, URL `http://dx.doi.org/10.1016/j.is.2010.01.001`

Bertolino A, Gnesi S (2003) Use case-based testing of product lines. ACM SIGSOFT Software Engineering Notes 28(5):355–358

Binkley D, Society IC (1997) Test Cost Reduction 23(8):498–516

Boehm B (2005) Value-Based Software Engineering : Seven Key Elements and Ethical Considerations (February)

Boehm B, Sullivan K (1992) Software Economics : A Roadmap pp 319–343

Coello Coello CA (2006) Evolutionary multi-objective optimization: a historical view of the field. Computational Intelligence Magazine, IEEE 1(1):28–36

Coffman Jr E, Garey M, Johnson D (1996) Approximation algorithms for bin packing: A survey. In: Approximation algorithms for NP-hard problems, PWS Publishing Co., pp 46–93

Cohen MB, Dwyer MB, Shi J (2006) Coverage and adequacy in software product line testing. Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis - ROSATEA '06 pp 53–63, DOI 10.1145/1147249.1147257, URL `http://portal.acm.org/citation.cfm?doid=1147249.1147257`

Cohen MB, Dwyer MB, Society IC (2008) Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints : A Greedy Approach 34(5):633–650

Colanzi TE, Assunção WKG, de Freitas Guilhermino Trindade D, Zorzo CA, Vergilio SR (2013) Evaluating Different Strategies for Testing Software Product Lines. Journal of Electronic Testing 29(1):9–24, DOI 10.1007/s10836-012-5343-y, URL `http://link.springer.com/10.1007/s10836-012-5343-y`

Cordy M, Schobbens PY, Heymans P, Legay A (2013) Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In: Software Engineering (ICSE), 2013 35th International Conference on, pp 472–481, DOI 10.1109/ICSE.2013.6606593

Dalal SR, Jain A, Karunanithi N, Leaton JM, Lott CM, Patton GC, Horowitz BM (1999) Model-based testing in practice. In: PROC. INTL. CONF. ON SOFTWARE ENGINEERING (ICSE '99), pp 285–294

Deb K (2001) Multi-objective optimization. Multi-objective optimization using evolutionary algorithms pp 13–46

do Carmo Machado I, McGregor JD, Santana de Almeida E (2012) Strategies for testing products in software product lines. ACM SIGSOFT Software Engineering Notes 37(6):1, DOI 10.1145/2382756.2382783, URL http://dl.acm.org/citation.cfm?doid=2382756.2382783

Dougherty B, White J, Schmidt DC (2012) Model-driven auto-scaling of green cloud computing infrastructure. Future Generation Comp Syst 28(2):371–378

Galindo JA, Benavides D, Segura S (2010) Debian packages repositories as software product line models. towards automated analysis. In: ACoTA, pp 29–34

Galindo JA, Alférez M, Acher M, Baudry B, Benavides D (2014) A variability-based testing approach for synthesizing video sequences. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2014, pp 293–303, DOI 10.1145/2610384.2610411, URL http://doi.acm.org/10.1145/2610384.2610411

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

Hartman A (2005) Software and hardware testing using combinatorial covering suites. In: Graph Theory, Combinatorics and Algorithms, Springer, pp 237–266

Henard C, Papadakis M, Perrouin G, Klein J, Traon YL (2013) Multi-objective test generation for software product lines. In: Proceedings of the 17th International Software Product Line Conference, ACM, New York, NY, USA, SPLC '13, pp 62–71, DOI 10.1145/2491627.2491635, URL http://doi.acm.org/10.1145/2491627.2491635

Johansen MF, Haugen Oy, Fleurey F (2012a) An algorithm for generating t-wise covering arrays from large feature models. Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -volume 1 p 46, DOI 10.1145/2362536.2362547, URL http://dl.acm.org/citation.cfm?doid=2362536.2362547

Johansen MF, Haugen Oy, Fleurey F, Eldegard AG, Syversen Tr (2012b) Generating Better Partial Covering Arrays by Modeling Weights on Sub-product Lines pp 269–284

Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., DTIC Document

Kästner C, von Rhein A, Erdweg S, Pusch J, Apel S, Rendel T, Ostermann K (2012) Toward variability-aware testing. In: Proceedings of the 4th International Workshop on Feature-Oriented Software Development, ACM, New York, NY, USA, FOSD '12, pp 1–8, DOI 10.1145/2377816.2377817, URL http://doi.acm.org/10.1145/2377816.2377817

Kuhn D, Wallace D, Gallo J AM (2004) Software fault interactions and implications for software testing. Software Engineering, IEEE Transactions on 30(6):418–421, DOI 10.1109/TSE.2004.24

Lamancha BP, Usaola MP (2010) Testing Product Generation in Software Product Lines pp 111–125

van der Linden FJ, Schmid K, Rommes E (2007) Software product lines in action. Springer

Lopez-Herrejon RE, Galindo JA, Benavides D, Segura S, Egyed A (2012) Reverse engineering feature models with evolutionary algorithms: An exploratory study. In: 4th Symposium on Search Based Software Engineering, Springer, Trento, Italy, pp 168–182

Lotufo R, She S, Berger T, Czarnecki K, Wasowski A (2010) Evolution of the linux kernel variability model. Software Product Lines: Going Beyond pp 136–150

Martello S, Toth P (1990) Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc.

Mendonca M, Branco M, Cowan D (2009) S.p.l.o.t.: Software product lines online tools. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, ACM, New York, NY, USA, OOPSLA '09, pp 761–762, DOI 10.1145/1639950.1640002, URL http://doi.acm.org/10.1145/1639950.1640002

Muccini H, Van Der Hoek A (2003) Towards testing product line architectures. Electronic Notes in Theoretical Computer Science 82(6):99–109

Nie C, Leung H (2011) A survey of combinatorial testing. ACM Computing Surveys (CSUR) 43(2):11

Olaechea R, Stewart S, Czarnecki K, Rayside D (2012) Modelling and multi-objective optimization of quality attributes in variability-rich software. In: Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages, ACM, New York, NY, USA, NFPinDSML '12, pp 2:1–2:6, DOI 10.1145/2420942.2420944, URL http://doi.acm.org/10.1145/2420942.2420944

Oster S, Markert F, Ritter P (2010) Automated Incremental Pairwise Testing of Software Product Lines pp 196–210

Passos L, Novakovic M, Xiong Y, Berger T, Czarnecki K, Wasowski A (2011) A study of non-boolean constraints in variability models of an embedded operating system. ACM, Munich, Germany, URL http://fosd.de/2011

Perrouin G, Sen S, Klein J, Baudry B, Traon YL (2010) Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. 2010 Third International Conference on Software Testing, Verification and Validation pp 459–468, DOI 10.1109/ICST.2010.43, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5477055

Perrouin G, Oster S, Sen S, Klein J, Baudry B, Traon Y (2011) Pairwise testing for software product lines: comparison of two approaches. Software Quality Journal pp 605–643, DOI 10.1007/s11219-011-9160-9, URL http://www.springerlink.com/index/10.1007/s11219-011-9160-9

Pohl K, Metzger A (2006) Software product line testing. Commun ACM 49(12):78–81, DOI 10.1145/1183236.1183271, URL http://doi.acm.org/10.1145/1183236.1183271

Roos-Frantz F, Benavides D, Ruiz-Cortés A, Heuer A, Lauenroth K (2012) Quality-aware analysis in product line engineering with the orthogonal variability model. Software Quality Journal 20(3-4):519–565

Rothermel G, Hall D (1997) A Safe , Efcient Regression Test Selection Technique (2):1–35

Sayyad A, Menzies T, Ammar H (2013) On the value of user preferences in search-based software engineering: A case study in software product lines. In: Software Engineering (ICSE), 2013 35th International Conference on, pp 492–501, DOI 10.1109/ICSE.2013.6606595

Segura S, Galindo J, Benavides D, Parejo J, Ruiz-Cortés A (2012) Betty: Benchmarking and testing on the automated analysis of feature models. In: Eisenecker U, Apel S, Gnesi S (eds) Sixth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12), ACM, Leipzig, Germany, pp 63–71

She S, Lotufo R, Berger T, Wasowski A, Czarnecki K (2010) The variability model of the linux kernel. VaMoS 10:45–51

She S, Ryssel U, Andersen N, Wasowski A, Czarnecki K (2014) Efficient synthesis of feature models. Information and Software Technology (0), DOI http://dx.doi.org/10.1016/j.infsof.2014.01.012, URL http://www.sciencedirect.com/science/article/pii/S0950584914000238

Sinnema M, Deelstra S (2007) Classifying variability modeling techniques. Information and Software Technology 49(7):717–739

Smith B, Feather MS (2000) Challenges and methods in testing the remote agent planner. In: In Proc. 5th Int.nl Conf. on Artificial Intelligence Planning and Scheduling (AIPS, pp 254–263

Sneed H (2009) Value Driven Testing. 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques pp 157–166, DOI 10.1109/TAICPART.2009.13, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5381632

Spillner A, Linz T, Schaefer H (2011) Software testing foundations: a study guide for the certified tester exam. O'Reilly Media, Inc.

Srikanth H, Williams L, Osborne J (2005) System test case prioritization of new and regression test cases. In: Empirical Software Engineering, 2005. 2005 International Symposium on, pp 10 pp.–, DOI 10.1109/ISESE.2005.1541815

Tang QY, Friedberg P, Cheng G, Spanos CJ (2007) Circuit size optimization with multiple sources of variation and position dependant correlation. In: Advanced Lithography, International Society for Optics and Photonics, pp 65,210P–65,210P

Thum T, Batory D, Kastner C (2009) Reasoning about edits to feature models. In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, IEEE, pp 254–264

White J, Galindo JA, Saxena T, Dougherty B, Benavides D, Schmidt DC (2014) Evolving feature model configurations in software product lines. Journal of Systems and Software 87(0):119 – 136, DOI http://dx.doi.org/10.1016/j.jss.2013.10.010, URL http://www.sciencedirect.com/science/article/pii/S0164121213002434

Withey J (1996) Investment Analysis of Software Assets for Product Lines

Zhu H, Hall PAV, May JHR (1997) Software unit test coverage and adequacy. ACM Computing Surveys (CSUR) 29(4):366–427