ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ
ΣΧΟΛΗ ΟΙΚΟΝΟΜΙΑΣ ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Διατριβή επί διδακτορία

ΕΙΣΑΓΩΓΗ ΚΑΙΝΟΤΟΜΩΝ ΜΕΘΟΔΩΝ ΣΤΟΝ ΕΛΕΓΧΟ ΚΑΙ ΣΤΗΝ ΑΔΕΙΟΔΟΤΗΣΗ ΛΟΓΙΣΜΙΚΟΥ

Σωτηρόπουλος Παναγιώτης
ΑΜ 2022201599003

Επιβλέπων:
Κώστας Βασιλάκης
Καθηγητής Τμήματος Πληροφορικής και Τηλεπικοινωνιών του
Πανεπιστημίου Πελοποννήσου

Τρίπολη, Μάιος 2025

Ευχαριστίες

Πρώτα απ΄ όλα, θα ήθελα να εκφράσω την ευγνωμοσύνη μου στον επιβλέποντα καθηγητή μου κύριο Κωνσταντίνο Βασιλάκη, για την καθοδήγηση και τη βοήθειά του καθ΄ όλη τη διάρκεια της εκπόνησης της διδακτορικής μου διατριβής. Η συμβολή του κύριου Βασιλάκη, με τις κατάλληλες υποδείξεις και την καθοδήγησή του, ιδιαίτερα κατά τις δύσκολες στιγμές αυτής της πορείας, ήταν καθοριστική, γι΄ αυτό τον ευχαριστώ από καρδιάς.

Επιπλέον, θα ήθελα να ευχαριστήσω τη γυναίκα μου και τα παιδιά μου για την υπομονή τους και τη στήριξή τους σε όλο αυτό το διάστημα.

Τέλος, θα ήθελα να αφιερώσω αυτή την εργασία στους γονείς μου, στη γυναίκα μου και στα παιδιά μου.

Περίληψη

Τα προγράμματα λογισμικού είναι πλέον αναπόσπαστο τμήμα της ζωής μας και την επηρεάζουν ποικιλοτρόπως σε όλους τους τομείς τους: από τον τομέα της επικοινωνίας, μέχρι την ενημέρωση, από την οικονομία έως την επεξεργασία δεδομένων και από την ψυχαγωγία στον ελεύθερο χρόνο μας έως την έρευνα για φάρμακα. Η ποικιλία των προγραμμάτων λογισμικού τα οποία είναι διαθέσιμα προς χρήση για τους σκοπούς που ήδη αναφέραμε είναι τεράστια, ενώ αντίστοιχα σημαντική ποικιλομορφία υφίσταται και στις μεθόδους και διαδικασίες ανάπτυξης των προγραμμάτων. Επιπλέον, τα προγράμματα εξελίσσονται με την πάροδο του χρόνου, οδηγώντας είτε σε καινούργιες εκδόσεις είτε σε νέα προγράμματα με εκτεταμένες λειτουργικότητες.

Η παροχή εγγυήσεων λειτουργίας για τα προγραμμάτων λογισμικού είναι ένας πολύ σημαντικός τομέας, στον οποίο έχουν αναπτυχθεί και εφαρμόζονται πολυάριθμες τεχνικές και μέθοδοι που σκοπό έχουν την επίτευξη της μέγιστη δυνατής διασφάλισης της ποιότητας των προγραμμάτων, τόσο ως προς τις λειτουργικές τους απαιτήσεις (ιδίως αναφορικά με τις διαστάσεις της ορθότητας και της πληρότητας), αλλά και ως προς τις μη λειτουργικές τους απαιτήσεις, συμπεριλαμβάνοντας μεταξύ άλλων τις διαστάσεις των επιδόσεων, της αξιοπιστίας, της συντηρησιμότητας, της ασφάλειας, της διαλειτουργικότητας κ.ο.κ. Μία σημαντική μέθοδος για την παροχή των εγγυήσεων, είναι ο έλεγχος των προγραμμάτων.

Τα προγράμματα μπορούν να ελεγχθούν είτε στατικά είτε δυναμικά. Στις στατικές δοκιμές, τα προγράμματα δεν εκτελούνται αλλά αναλύονται τα χαρακτηριστικά του περιεχομένου τους (εντολές προγραμματισμού) και η δομή τους, ενώ στις δυναμικές δοκιμές πραγματοποιείται εκτέλεση των προγραμμάτων και εξετάζονται η συμπεριφορά τους και οι καταγραφές της εκτέλεσής της. Μερικές από τις κατηγορίες/μέθοδους δοκιμής λογισμικού είναι:

- λειτουργικός έλεγχος, δηλαδή έλεγχος εάν οι λειτουργίες εκτελούνται σωστά και με ακρίβεια. Ο λειτουργικός έλεγχος περιλαμβάνει δοκιμές μονάδας, δοκιμές ενοποίησης, δοκιμές συστήματος, δοκιμές αποδοχής κ.λπ.

- μη λειτουργικές δοκιμές, δηλαδή δοκιμές για να προσδιοριστεί εάν πληρούνται οι μη λειτουργικές απαιτήσεις του λογισμικού, όπως ασφάλεια, απόδοση, χρηστικότητα, συμβατότητα κ.λπ.

Επιπρόσθετα, τα προγράμματα λογισμικού τυπικά συνοδεύονται από άδειες χρήσης, οι οποίες προσδιορίζουν με ποιους τρόπους μπορεί να χρησιμοποιηθεί το ίδιο το πρόγραμμα ή/και ο κώδικάς του (στην περίπτωση που ο κώδικας του προγράμματος είναι διαθέσιμος). Η παροχή κατάλληλου πλαισίου αδειοδότησης είναι σημαντική, τόσο αναφορικά με την ανάπτυξη της καινοτομίας, όσο και αναφορικά με τις εγγυήσεις που μπορούν να έχουν οι χρήστες για τη χρήση του λογισμικού, εμπεδώνοντας αίσθηση ασφάλειας.

Η παρούσα διατριβή εξετάζει αυτές τις δύο πτυχές προτείνοντας νέες, καινοτόμες μέθοδους για (α) τον έλεγχο και (β) το πλαίσιο αδειοδότησης του λογισμικού. Ειδικότερα:

Στο Κεφάλαιο 1 εισάγεται ένα νέο πλαίσιο διαχείρισης του κώδικα ελέγχου του λογισμικού, το οποίο μπορεί να εφαρμοστεί σε συνδυασμό με όλες τις υπάρχουσες μεθόδους δοκιμών ελέγχου και για όλες τις γλώσσες προγραμματισμού. Η κεντρική φιλοσοφία του νέου πλαισίου είναι ότι όλα τα διαθέσιμα τεστ/δοκιμές διατηρούνται

κεντρικά, σε ένα ή περισσότερα αποθετήρια, και κάθε έλεγχος επισημειώνεται με μεταπληροφορίες σχετικά με τα στοιχεία του κώδικα, τις εκδόσεις λογισμικού και τα χαρακτηριστικά των εκδόσεων για τις οποίες προορίζεται. Κατόπιν, κατά τη φάση της δημιουργίας και του ελέγχου ενός εκτελέσιμου προγράμματος, ανάλογα με τον κώδικα και την έκδοση που ελέγχεται, εντοπίζονται οι κατάλληλες δοκιμές ελέγχου, οι οποίες εξάγονται και χρησιμοποιούνται. Μερικά από τα πλεονεκτήματα της νέας μεθόδου είναι ότι (α) οι δοκιμές γράφονται μόνο μία φορά και μπορούν στη συνέχεια να χρησιμοποιηθούν για οποιοδήποτε πλήθος προγραμμάτων και και εκδόσεων, (β) οι έλεγχοι μπορούν να χρησιμοποιηθούν για δοκιμές προγραμμάτων με δυναμικά χαρακτηριστικά (γ) υποστηρίζει και προάγει την ανάπτυξη προγραμμάτων με βάση τις δοκιμές (test-driven development), (δ) υποστηρίζει τις δοκιμές προγραμμάτων λογισμικού με διαφορετικές ρυθμίσεις, (ε) υποστηρίζει τη μεταγλώττιση και σύνδεση προγραμμάτων με βάση τα χαρακτηριστικά (feature-based builds) και (στ) υποστηρίζει τη σύγκριση και τον συνδυασμό των λειτουργιών των διαφορετικών εκδόσεων των προγραμμάτων.

Στο Κεφάλαιο 2 η μέθοδος που περιγράφηκε στο Κεφάλαιο 1, επεκτείνεται σε εφαρμογές κινητών συσκευών, καλύπτοντας χαρακτηριστικά που σχετίζονται με τις ιδιαιτερότητες του οικοσυστήματος των κινητών εφαρμογών, όπως η ποικιλομορφία των συσκευών και των λειτουργικών συστημάτων, η ανάγκη για εξοικονόμηση ενέργειας, οι μεταβολές στη δικτυακή συνδεσιμότητα κ.ο.κ.

Στα Κεφάλαια 3-4 παρουσιάζονται αναλυτικά δύο περιπτώσεις χρήσης του πλαισίου διαχείρισης του κώδικα ελέγχου του λογισμικού για επαύξηση της ασφάλειας του λογισμικού. Ειδικότερα, στο Κεφάλαιο 3 παρουσιάζουμε ένα πλαίσιο διαχείρισης ευπάθειας λογισμικού για την ελαχιστοποίηση του κινδύνου επίθεσης στα συστήματα Internet of Things (IoT). Το πλαίσιο αυτό διευκολύνει (α) τη διαμόρφωση του λογισμικού ώστε να περιλαμβάνει μόνο τις απαραίτητες δυνατότητες, (β) την εκτέλεση δοκιμών που σχετίζονται με την ασφάλεια και τη σύνταξη λιστών ευπάθειας λογισμικού σε όλη την πλατφόρμα, και γ) την ιεράρχηση της αντιμετώπισης των ευπαθειών, λαμβάνοντας υπόψη τον αντίκτυπο κάθε ευπάθειας, το σχετικό τεχνικό χρέος (technical debt) για την αποκατάστασή του και τον διαθέσιμο προϋπολογισμό ασφαλείας. Το προτεινόμενο πλαίσιο μπορεί να χρησιμοποιηθεί ως βοήθημα στην υλοποίηση της πλατφόρμας IoT από αρχιτέκτονες λογισμικού, προγραμματιστές και υπεύθυνους ασφάλειας.

Στο Κεφάλαιο 4, προτείνεται μια μέθοδος για την ελαχιστοποίηση του κινδύνου που οφείλεται στο λογισμικό βιβλιοθηκών. Η προτεινόμενη μέθοδος οποία βασίζεται στο το πλαίσιο που παρουσιάζεται στα Κεφάλαια 1- 2. Στα σύγχρονα συστήματα Internet of Things (IoT), αλλά και στα συστήματα λογισμικού ευρύτερα, αξιοποιούνται βιβλιοθήκες για την υλοποίηση των απαιτούμενων λειτουργιών. Οι βιβλιοθήκες, ωστόσο, συχνά εμπεριέχουν τρωτά σημεία, τα οποία μπορεί να αποτελέσουν αντικείμενο εκμετάλλευσης από επιτιθέμενους, και να οδηγήσουν σε περιστατικά ασφάλειας. Σε πολλές περιπτώσεις, η λειτουργικότητα που απαιτείται από μια εφαρμογή υλοποιείται από μια σειρά εναλλακτικών βιβλιοθηκών, με κάθε βιβλιοθήκη να έχει τη δική της λίστα ευπαθειών, ενώ διαφοροποιήσεις μπορεί να υφίστανται και σε άλλες μη λειτουργικές ιδιότητες (π.χ. απόδοση εκτέλεσης, αποτύπωμα μνήμης κ.λπ.). Η προτεινόμενη προσέγγιση μπορεί να αυτοματοποιήσει την εργασία ελαχιστοποίησης του επιπέδου κινδύνου των συστημάτων IoT (ή εν γένει του λογισμικού) που οφείλεται στα τρωτά σημεία των βιβλιοθηκών που απαιτούνται από το υπό δοκιμή λογισμικό. Η προτεινόμενη προσέγγιση εκμεταλλεύεται τη γνώση σχετικά με τις βιβλιοθήκες που παρέχουν ισο-

δύναμη λειτουργικότητα, και αυτόματα αξιολογεί το επίπεδο κινδύνου των υποψήφιων συνδυασμών βιβλιοθηκών, και τελικά επιλέγει τον συνδυασμό βιβλιοθηκών που παρουσιάζει το ελάχιστο επίπεδο κινδύνου προκειμένου να τον ενσωματώσει στο εκτελέσιμο. Επιπλέον, το επίπεδο κινδύνου των υποψήφιων εφαρμογών παρακολουθείται συνεχώς για εντοπισμούς νέων τρωτών σημείων ή διορθώσεις στις υλοποιήσεις, ενεργοποιώντας νέες αξιολογήσεις κινδύνου και παράγοντας νέα εκτελέσιμα αρχεία, ανάλογα με την περίπτωση.

Στα Κεφάλαια 5-6 της διατριβής, μελετάται μία μέθοδος δυναμικών δοκιμών, κατά την οποία εκτελούνται όλες οι δυνατές διαδρομές εκτέλεσης για συγκεκριμένες εισόδους. Οι μέθοδοι αυτές βοηθούν στην εύρεση διαλειπόντων σφαλμάτων (intermittent faults) στον κώδικα, σφαλμάτων δηλαδή που παρουσιάζονται σπάνια, μόνο όταν η εκτέλεση του προγράμματος ακολουθεί συγκεκριμένα μονοπάτια στον κώδικα λόγω συγκεκριμένων εισόδων, απρόσμενων αλληλεπιδράσεων, συνθηκών στο περιβάλλον της εφαρμογής κ.λπ. Σε αυτές τις μεθόδους εντοπίζονται συγκεκριμένα μοτίβα πρόσβασης σε διαμοιραζόμενες μεταβλητές που με μεγάλη πιθανότητα υποδεικνύουν ύπαρξη διαλειπόντων σφαλμάτων στον κώδικα. Στη συνέχεια ο χρήστης καλείται να ελέγξει τα αντίστοιχα σημεία του κώδικα προκειμένου να αποφασίσει αν όντως υπάρχει σφάλμα. Ειδικότερα, ο αλγόριθμος που παρουσιάζεται στο Κεφάλαιο 5 εξετάζει τα μοτίβα πρόσβασης θεωρώντας μόνο τις άμεσες εξαρτήσεις μεταξύ των διαμοιραζόμενων μεταβλητών, ενώ στο Κεφάλαιο 6 παρουσιάζεται μία επέκταση του αλγόριθμου η οποία θεωρεί και έμμεσες εξαρτήσεις μεταξύ των διαμοιραζόμενων μεταβλητών.

Στο Κεφάλαιο 7, προτείνεται μία μέθοδος *εν δυνάμει ευρεσιτεχνίας* για το λογισμικό, η οποία επιτρέπει τη διασφάλιση της διανοητικής ιδιοκτησίας με περιορισμένο κόστος, καθιστώντας έτσι τη διαδικασία απόκτησης πατέντας προσιτή σε οποιονδήποτε ενδιαφερόμενο. Επιπλέον, γίνεται αναφορά στην υπάρχουσες άδειες χρήσης λογισμικού και προτείνεται μία νέα διαδικασία προσωρινής αδειοδότησης που δίνει μεγαλύτερη ευελιξία στην επιλογή της άδειας, και παρέχει πληροφορίες για τη διάρκειά της στους χρήστες.

Τέλος, στο Κεφάλαιο 8 παρατίθεται μία σύνοψη της διδακτορικής διατριβής και σκιαγραφούνται μελλοντικές κατευθύνσεις έρευνας.

Λέξεις-κλειδιά: Έλεγχος λογισμικού, Διαλείποντα σφάλματα, Ανίχνευση σφαλμάτων, Διαμοιραζόμενες μεταβλητές, Έλεγχος βάσει μοντέλων, Πλαίσιο Πρόσθετων Ελέγχων, Διαχείριση κώδικα ελέγχων, Δυναμικός έλεχος, Ανάπτυξη καθοδηγούμενη από τους ελέγχους, Έλεγχος λογισμικού πολλαπλών εκδόσεων, Έλεγχος σε πολλαπλά περιβάλλοντα, Αποθετήριο προγραμμάτων ελέγχου, Έλεγχος με βάση χαρακτηριστικά, Έλεγχος εφαρμογών διαδικτύου, Έλεγχος κινητών εφαρμογών, Πατέντες λογισμικού, Άδειες χρήσης λογισμικού

Abstract

Software programs are now an indispensable part of our lives, affecting them in a multiple ways across a number of sectors:, from communication to information, from economy to data processing and from entertainment in leisure time to drug research. The variety of software programs that are available for use in the context of the above mentioned domains is vast, while a correspondingly wide divergence exists in the area of methods and procedures of software development. Moreover, programs evolve along the passage of time, leading either to new versions, or to new programs with extended functionalities.

The provision of guarantees concerning the correct operation of the software programs is a very important topic, where various techniques and methods have been developed and applied to achieve the highest possible quality of the programs, both in terms of functionality and interoperability with other programs. The execution of tests is one of the main methods for providing guarantees.

Programs can be tested either statically or dynamically. In static testing, programs are not executed but rather traits of their content (programming instructions) and structure are analyzed, while in dynamic testing, programs are executed and their behavior and traces are examined. Some of the software testing categories/methods are:

- functional testing i.e. testing whether functionalities are executed correctly and accurately; functional testing is subdivided into unit testing, integration testing, system testing, acceptance testing.

- non-functional testing, i.e. tests to determine whether non-functional requirements of the software, such as security, performance, usability, compatibility etc. are met.

Additionally, software programs are typically shipped with licenses, which specify how the program itself and/or its code (if the program code is available) can be used. The provision of a suitable licensing framework is important, both concerning the development of innovations and regarding the guarantees offered to users for the use of the software, instigating a sense of safety and stability.

This thesis examines these two aspects proposing new, innovative methods for (a) software program testing and (b) the software licensing framework. More specifically:

In Chapter 1 a new framework for managing the software testing code is introduced, which can be used in combination with all existing control testing methods, and for all programming languages. The overarching idea of the new framework is that all available tests are maintained centrally, in one or more repositories, and each test is annotated with meta-information about the code elements and the versions for which it is intended. Then, during the phase of creating and testing an executable program, depending on the code and the version being tested, the appropriate/corresponding tests are identified, extracted, and used. Some of the advantages of the new method are that (a) tests are written only once and can then be used for unlimited programs and versions, (b) tests can be used to validate programs with dynamic features, (c) it supports and promotes test-driven development, (d) it supports testing software

programs with different configurations, (e) it supports feature-based builds, and (f) it supports comparing and combining the functions of different versions of programs.

In Chapter 2 the method described in Chapter 1 is extended to the area of mobile device applications, covering features related to the intricacies and particularities of the mobile application ecosystem, such as the diversity of devices and operating systems, the need for energy conservation, changes in network connectivity, and so on.

Chapters 3-4 present two use cases for the software control code management framework that aim to enhance software security. Specifically, in Chapter 3 we present a software vulnerability management framework to minimize the risk associated with attacks on Internet of Things (IoT) systems. This framework facilitates (a) configuring software to include only the necessary capabilities, (b) performing security-related testing and compiling software vulnerability lists across the platform, and (c) prioritizing vulnerability remediation, taking into account the impact of each vulnerability, the associated technical debt for its remediation, and the available security budget. The proposed framework can be used as an aid in the implementation of the IoT platform by software architects, developers as well as information security specialists.

In Chapter 4, a method for minimizing the risk due to software libraries is proposed, which is based on the framework presented in Chapters 1-2. In modern Internet of Things (IoT) systems, but also more broadly in software systems, libraries are used to implement the required functions. Libraries, however, often contain vulnerabilities, which can be exploited by attackers, and lead to security incidents. In many cases, the functionality required by an application is implemented by a series of alternative libraries, with each library having its own list of vulnerabilities, while variations may also exist in other non-functional properties (e.g. execution performance, memory footprint, etc.). The proposed approach can automate the task of minimizing the risk level of IoT systems (or software in general) due to vulnerabilities in the libraries required by the software under test. The proposed approach exploits knowledge about the libraries that provide equivalent functionality to automatically evaluate the risk level of candidate library combinations, and finally selects the library combination that presents the minimum risk level to integrate into the executable. In addition, the risk level of candidate applications is continuously monitored for new vulnerabilities or implementation fixes, triggering new risk assessments and generating new executables, as appropriate.

In Chapters 5-6 of the thesis, a dynamic testing method is studied, in which all possible execution paths for specific inputs are executed. These methods help in locating intermittent faults in the code, i.e. faults that occur rarely, and only when the execution of the program follows specific paths in the code due to specific inputs, unexpected interactions, conditions in the application environment, etc. In these methods, specific access patterns to shared variables are identified that indicate with a high probability the existence of intermittent faults in the code. The user is then asked to check the corresponding points in the code, in order to decide whether an error actually exists. In particular, the algorithm presented in Chapter 5 examines access patterns considering only direct dependencies between shared variables, while in Chapter 6 an extension of the algorithm is presented which additionally considers

indirect dependencies between shared variables.

In Chapter 7, a method for *tentatively patenting* software is proposed, which allows for safeguarding intellectual property with reduced cost, thus making the patent acquisition process accessible to anyone interested. Furthermore, reference is made to existing software licenses and a new versatile licensing framework is proposed that gives elevated flexibility in the choice of license, and provides information about its duration and additional guarantees to software artifact users.

Finally, Chapter 8 summarizes the doctoral thesis and outlines future research directions.

Keywords: Software testing, Intermittent faults, Fault detection, Shared variables, Model-based checking, Additional Testsuite Framework, Testsuite management, Dynamic testing, Test-driven development, Multiversion testing, Multiple environment testing, Test program repository, Feature-based testing, Internet application testing, Mobile application testing, Software patents, Software licenses

Contents

List of Figures

List of Tables

Chapter 1

The Additional Testsuite Framework: Facilitating Software Testing and Test Management

Abstract

In this chapter, we present the Additional Testsuite Framework, a novel test suite management approach, which provides structures and instrumentation for the creation, maintenance, evolution and use of test suites for software programs. In particular, the tests can be maintained in a centralized repository, and are developed and maintained independently of specific versions of the associated software. Through the use of annotations, tests are categorized and distributed to the desired versions of the software. The presented framework also supports test-based development, dynamic/selective program builds, feature-based builds, testing in different environments and source code analysis. The Additional Testsuite Framework concept has been implemented and extensively evaluated, with the test cases notably including the JBoss EAP CE and OpenLiberty servers.

1.1 Introduction

1.1.1 Software testing: context and challenges

Software testing is a major phase of the software development and maintenance process. It consumes a significant amount of effort in order to ensure that the developed software meets both its functional and non-functional requirements [1]. Testing promotes software reliability, which is defined as the probability that a software system will not cause system failure for a specified time under specified conditions [2].

Testing can be performed at varying levels (unit testing, integration testing, system testing) [3] and using different methods, tools and techniques (e.g. whitebox, graybox or blackbox testing [4]). Research in the field of software testing is continuing to keep in pace with evolving system requirements, development methodologies and techniques, leveraging comprehensiveness and enhancing automation. As asserted in

[5], the quality of the testing procedure results is bound by the quality of the test cases employed in this procedure, hence it is important that test cases are properly developed and maintained. Moreover, it is important that software programs can efficiently specify the tests they need to run; the fact that software programs may include different versions and/or may be deployed under different configurations, with each configuration possibly targeting particular hardware/software environments and integrating diverse subcomponents and/or libraries further perplexes the testing procedure and its management, since each program version, environment, subcomponent or library may necessitate the inclusion or exclusion of certain tests. [6] recognizes the selection of tests as a significant issue, transcending the dimensions of *how* (which tests are selected to be performed on a system) and *what* (decision of the actual code that is executed).

Similar concerns pertain to the software development phase, when developers specify program configurations, to be used for compiling the executable programs: in this context, a software module may have dependencies on various components, which may in turn have dependencies on other components, and all supporting components must be fetched and appropriately linked or packaged with the software module under development. Supporting components are either stored locally or are fetched from repositories. Moreover, dependencies between components that are more loosely coupled may be present, such as dependencies between web applications and containers, or applications and database systems. Special packaging, deployment and publication procedures may be needed to be followed to create executable software artifacts, make them operational or register them into software artifact repositories. To tackle the complexity arising from such dependencies, a multitude of tools is available for software developers to use: dependencies between components are declaratively specified using notations such as Makefiles [7] or the Project Object Model (POM) [8]. In all cases, each needed component may specify its own dependencies, and dependency managers such as Maven [8], Composer [9], Make [7] and NuGet [10] transitively resolve all dependencies ensuring that all necessary subcomponents are identified and finally integrated into the resulting software artifacts. Centralized or hierarchically structured repositories, such as Maven [8] and NuGet [10] can be used to underpin searching, downloading or publishing software artifacts. Dependencies between components that are more loosely coupled are managed by additional tools and supporting configurations, such as Hibernate [11] and .NET Persistence API [12]. Finally, software artifact assembly, deployment and publishing procedures are specified using suitable means and tools, such as Makefile instructions [7], Maven plugins (e.g. the Apache Maven Assembly plugin [13] and the Apache Tomcat Maven plugin [14]), or continuous integration tools such as Jenkins [15].

However, the automations and facilities that are available for easing the management of software components and their integration are not available for the software testing phase. Current testing methodologies, practices and tools require that developers and testers create the tests needed for each individual program version and subsequently couple program versions with tests in a manual fashion. Furthermore, the code realizing the tests is typically copied and pasted across versions, introducing the need for synchronized updates and the possibility for test misconfiguration. Simi-

larly, testers must manually identify and arrange for the execution of tests specifically crafted for validating program behaviour in specific environments (e.g. application containers or cloud deployments), further encumbering the testing process.

1.1.2 The Additional Testuite Framework

In this chapter, the concept of the Additional Testuite Framework (AT Framework) is presented, which provides a holistic management for tests, covering the test development, maintenance and delivery/use phases.

The AT Framework upgrades the classic one-to-one mapping between a program version and a statically specified corresponding testsuite into a dynamic testsuite specification, where tests to be included are designated through declarative specifications that are evaluated on-the-fly to derive the set of tests that need to be applied on the particular case. Tests are developed and annotated to designate the software artifact/version to which they apply, the deployment environment in which they will be used, or any other pertinent aspect that may be related to testing; then upon test execution, the AT framework instrumentation extracts the tests annotations and the test execution specification to automatically determine which tests should be applied in the particular case. The AT Framework can support all the types and levels of testing, such as unit testing, component testing or integration testing.

For instance, consider the case that a feature addition is delivered in a subcomponent of a multi-version software program and applied by a component upgrade. Under such a circumstance, one integration test for each version of the software would need to be developed (or be copy/pasted across versions) and integrated into the test suite for the particular version. Using however the AT Framework, it is possible to develop a single integration test, which would be added to the AT Framework repository and from there be automatically integrated into all version-specific sets of tests.

The AT Framework has the following advantages :

1. It introduces and formalizes the concept of a test program repository, allowing all test programs to be stored and maintained in one place.

2. Each test is written and stored once, and can be integrated into the testsuites of any number of software program or program versions.

3. Through the use of annotations, tests can be flexibly associated to multiple programs, versions and environments, reducing the complexity and effort needed for test case maintenance.

4. It supports a multitude of testing paradigms, including dynamic feature testing, test-driven development, and source code analysis.

5. It facilitates tester's tasks within testing paradigms, including the localization of regression bugs, testing within different. configurations and compilation of feature-based documentation through the management of tests documentation [16].

The main concept behind the AT Framework is the storage and maintenance of tests within one or multiple repositories, and indexing them using annotations. Subsequently, upon test execution, the test annotations are matched against the test specification to select the appropriate ones and run them against the software under test. This approach facilitates the maintenance of tests, eases the association of tests with the appropriate software artifacts and allows for the dynamic assembly of tests, taking into account the software artifact under construction, the features to be included therein and the environment within which the test procedure will be conducted.

1.1.3 Research contribution and chapter structure

The AT framework proposed in this chapter is a framework that provides test developers and testers with additional capabilities concerning the management and applications of the testsuites, while it additionally integrates standard testing procedures under a comprehensive environment. It advances the state-of-the-art in software testing by delivering the advantages (1)-(5) listed in the previous section in the software testing process. It is also possible to combine the proposed framework with other state of the art testing methods, such as automated test case generation [17, 18] or test case selection/prioritization [19], while it can be used with any software programming language. To substantiate the chapter contribution to the state-of-the-art and quantify the reaped benefits, the proposed approach is evaluated under the following research questions:

RQ1) to what extent does the AT Framework contribute in reducing the number of test code bases in multi-version software?

RQ2) to what extent does the AT Framework contribute in reducing the number of test code bases in feature-based software compositions?

RQ3) to what extent does the AT Framework reduce the number of test method implementations present in the test code bases?

RQ4) is the workload of adding the necessary annotations manageable and less than the workload needed to manage the separate test code bases?

The proposed framework is of generic nature and can be applied to any application type, including traditional"/desktop applications and internet applications as well. In the latter context, the following use cases of the proposed framework can be identified:

1. managing tests for multi-version web software, both concerning web application containers (e.g. WildFly, JBoss and OpenLiberty), and individual web applications, which can be versioned either for the purpose of dynamic application update [20] or for delivering different features to different customer installations.

2. managing tests for different execution environments in which a web application needs to be tested; this includes testing within different application containers (e.g. WildFly vs. Apache Tomcat), testing within different container versions (e.g. JBoss EAP 7.4 vs. JBoss EAP 7.0), testing under varying virtualization setups (e.g. "bare machine"/no virtualization vs docker containers[1] setup vs. virtual machines[2]) or testing under different integrations (e.g. using SQL server vs. MySQL to implement the web application's data layer).

3. following a test-based approach to the development of internet applications.

4. integrating a number of conventional testing techniques that are applied on internet applications, e.g. source code analysis, unit tests and integration tests, under a single, comprehensive testing environment.

The rest of the chapter is structured as follows: section 1.2 overviews related work. Section 1.3 introduces the proposed AT Framework while section 1.4 describes the use cases for the AT framework, some of which cover only testing, while other use cases expand cover related tasks, such as test-driven development. Finally, section 1.5 introduces some tools to complement the operation of the framework, section 1.6 presents an evaluation of the AT framework and section 1.7 concludes the chapter.

1.2 Related work

In the current testing practice, testsuites are, in most cases, attached to the software program and are updated along with the version changes of the software program. This approach though, assumes that older versions will not be developed any more, and this assumption is not always true: in the current era we observe that multiple versions of operating systems [21, 22] or other software [23, 24, 25] are maintained in parallel, where old versions may either receive maintenance support only (i.e. receive bug fixes), or be extended to accommodate additional features through backports [26]. In all cases, when the codebase of the current version is modified to either fix a bug or introduce a new feature, the associated tests should be copied to the test bases of affected old versions as well. Regression testing tools [27, 28] provide support for maintaining tests pertaining to multiple versions and associating specific tests with versions of the software to be tested, however they do not provide comprehensive support for a number of other features, including feature-based testing, test-driven development, or dynamic program builds.

According to [29], software testing involves the execution of a software component or system component to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test:

1. meets the requirements that guided its design and development,

2. responds correctly to all kinds of inputs,

[1] `https://www.docker.com/resources/what-container/`

[2] `https://azure.microsoft.com/en-us/overview/what-is-a-virtual-machine/`

3. performs its functions within an acceptable time,
4. it is sufficiently usable,
5. can be installed and run in its intended environments, and
6. achieves the general result its stakeholders desire.

Testing can be conducted following a number of different types of test approaches as well as at different levels. Regarding the test types, notable testing paradigms include the following:

1. Static [30] vs. dynamic testing: Static testing performs the checks without executing the software program (it can be done manually or by a set of tools in order to check the code, requirement documents and design documents and add review comments), while dynamic testing executes the program in order to detect faults.
2. The exploratory approach: This type of testing approach [31], where testers continuously gain experience and knowledge, and combine them with their creativity to expand and improve the tests.
3. The "box" approach [32]: The box approach considers the software as an artifact of some degree of opaqueness, and tests are correspondingly formulated to exploit the available level of knowledge on the internal software structure. Three major variations of the box approach which are typically used are:
    a. *White-box testing*: This type of testing, assumes that all the code of the software program to be tested is known; the tests target all aspects of the code, including its externally observed behavior (e.g. API calls) as well as its internal operation.
    b. *Black-box testing*: Black box testing assumes that the software code is not known; therefore tests may target only the externally observable behavior of the code.
    c. *Grey-box testing*: Grey box testing assumes some amount of knowledge regarding the code of the software program is available; consequently the tests can target both the externally observable behavior of the code and those aspects of the internal code structure that are known (e.g. APIs of subsystems).

Tests may also be classified according to the focus of the testing procedure:

1. Unit testing [33]: The level of software testing where individual components are tested.
2. Integration testing [34]: The level of software testing where individual units are combined and tested all together.
3. System testing [35]: The level of software testing where the complete software is tested.

4. Operational acceptance testing [36]: The level of software testing where a system is tested for acceptability of delivery.

while many testing types, techniques and tactics can be identified, e.g *functional* (verification of requirement specification, etc.) vs non-functional (performance, reliability, etc.) testing, regression testing [37], and so forth. We note here that the proposed AT Framework cannot be classified under any of the above categories, since it is not a testing methodology, but a comprehensive test management and exploitation framework, complete with the required instrumentation, to facilitate the work of testers, especially in the context of multiversion program testing, while additionally programs sharing parts of their code bases are also supported. Our approach considers versions of a software program or parts of code bases as independent artifacts, as far as the testing procedure is concerned, and each such independent artifact can be associated with any number of tests, independent of other artifacts. The declarative, version-aware specification of the association between tests and software artifacts, coupled with independent management and evolution of tests are the key features of the AT Framework approach, which allow testers to harvest a number of important benefits, as discussed in section 1.1.

Table 1.1 summarizes the different types of testing that are reported in the literature, correlating each type with the AT Framework use cases that may be used to support each type of testing and listing relevant method advantage.

Testing type / level	Description	Methods	Advantages	Limitations	AT framework use cases
Dynamic testing [38]	Executes the program in order to detect faults.	All execution paths are tested extensively	Can find all possible errors for specific input	Time and resource consuming	C.f. sections 1.4.1, 1.4.2, 1.4.3, 1.4.4, 1.4.5 and 1.4.6.
Static testing [30]	Performs the checks without executing the software program.	Performed manually (code audits or peer reviews) or using automated tools.	Automated tools provide a fast and easy way to find and fix the errors. Errors can be found at an early stage of development life cycle. Code audits and peer reviews capitalize on the expertise of reviewers.	Demands a lot of time and has a high cost when done manually. Automated tools may miss faults that may manifest at run-time.	C.f. sections 1.4.4, 1.4.5 and 1.4.7.
Exploratory testing [31]	Continuously expands and improves the tests.	Repeated iterations of analysis and testing development.	Supports focused testing and improves tester creativity.	Limited by the tester's skills. Determination of the test cases is tester-dependent.	C.f. sections 1.4.1, 1.4.2, 1.4.3, 1.4.4, 1.4.5 and 1.4.6.

White-box testing	Assumes that all the code of the software program to be tested is known and can be used in the tests.	Internal structure of the components is exposed to the tester. The tester builds tests exploiting the internal functions of the code.	All the aspects of the internal code can be tested and used for the creation of test cases.	Test cases need to be changed whenever the implementation is modified.	All use cases in section 1.4.
Black-box testing	Assumes that the main part of the software code in not known and relevant knowledge cannot be used in the tests.	Internal structure of the components is not exposed to the tester. The tester specifies the input and checks the output results.	Tests are less affected by changes in the implementation.	The internal code functions cannot be tested atomically.	All use cases in section 1.4.
Grey-box testing	Assumes that the main part of the software code is partially known, and relevant knowledge can be used in the tests.	Tester has partial knowledge of the internal structure of the components. The tests are developed based on the knowledge of the general functionality and the partial internal component knowledge.	Combines the advantages of white and black box testing (depending on the amount of internal structure knowledge utilized).	Combines the disadvantages of white and black box testing (depending on the amount of internal structure knowledge utilized).	All use cases in section 1.4.

Unit testing [33]	Testing of individual software components are tested.	Unit testing covers the test of the functionality of specific components.	Bugs can be located at early stages of development. Only the knowledge of the specific component is needed to write the tests.	Unit testing can't find all faults, since testing is confined to the functionality of the specific component.	C.f. sections 1.4.1, 1.4.3, 1.4.4, 1.4.5 and 1.4.6.
Integration testing [34]	The level of software testing where individual units are combined and tested as an integral entity.	Creation and execution of tests that examine functionalities delivered through the interaction of multiple components.	Finds faults that are not possible to be identified via unit testing.	Knowledge of all involved components and their interactions is needed by the testers. Change to a specific component, and especially to its contracts, may lead to change of the integration tests.	C.f. sections 1.4.1, 1.4.2, 1.4.3, 1.4.4, 1.4.5 and 1.4.6.

System testing [35]	The level of software testing where the complete software is tested.	Black-box technique for functional, non-functional, security, performance, end-to-end and other testing types.	Involves end-to-end testing. Can be done in production environment. Tests both the application architecture and business requirements.	Can be complicated, especially in distributed systems.	C.f. sections 1.4.1, 1.4.2, 1.4.3, 1.4.4, 1.4.5 and 1.4.6
Operational acceptance testing [36]	The system is tested for acceptability for delivery.	Operational testing is a type of non-functional acceptance testing in an environment similar to the production one.	It reassures performance, security, reliability, accessibility, stability, maintainability, accessibility, interoperability, backup and recovery.	Limited mainly to testing for non-functional aspects, therefore functional aspects are not considered.	C.f. sections 1.4.1, 1.4.2, 1.4.3, 1.4.4, 1.4.5 and 1.4.6.

Table 1.1: Support of AT Framework Structures for different testing types and levels

Source code version control tools such as git [39] or SVN [40] may offer a partial solution to the test case management problem: when a new version is forked, the test code is made available to the new version but effectively shared between the new version and the previous one(s). Source code version control tools may keep the test code files to a minimum when test code evolves linearly with the source code, i.e. when an addition, modification or deletion of a test method m made at software version v_i holds for all software versions $v_j : j > i$. However, if a modification to the test suite should only apply to a specific set of intermediate versions and needs subsequently to be rolled back, no direct mechanism for maintaining a single copy of the respective method is available. Moreover, source code version control tools have cannot effectively manage the maintenance of the test case pool: for instance the addition of a new test case to all versions or the need for a modification of a test case that needs to be reflected across all versions, necessitate a distinct commit action for each version. Additionally, source code version control tools can only be used to (partially) manage test code files, but do not support additional functionalities such as feature-based testing, testing against different environments, test-driven development [41, 42, 43] and so forth.

A different approach to providing support to developers with the software testing task is the automated generation of test cases. [17] presents a state-based model which extracts the behaviors of the system specified in use cases, and uses these behaviors to generate test case. [44] recognizes that automated processing of generic-purpose and arbitrarily written use cases for automated test case generation yields suboptimal results due to the diversity of the ways that use cases are expressed and the limitations of natural language processing, and proposes Restricted Use Case Modeling, an approach for writing system use cases in a fashion that enables their automated processing and the formulation of test cases. The search-based test case generation approach employs meta-heuristic optimization algorithms, including genetic algorithms (GA), to create additional test cases by exploiting knowledge obtained from the existing ones, in order to reach the desired objectives for the test case base [45, 46, 18]. A survey of automated test generation is presented in [47]. However, the efficiency of automated test case generation has not reached insofar the desired levels, and still the software testing phase necessitates considerable resources and cost, ranging from 15% to 45% of the overall software development cost [48, 49, 50]. This stems from:

- the need to manually craft test cases that guarantee full code coverage and testing efficiency.
- once the test case code base has been crafted, it needs to be maintained across multiple versions of the software and/or contexts of its use, such as its inclusion as a component in other software, to drive the development process (as in the case of test-driven development [41]), to accommodate testing in varying and evolving environments and so forth.

The AT Framework responds to these challenges by underpinning and facilitating the management and maintenance of test case bases, while additionally providing comprehensive support for a multitude of testing paradigms, and supporting tester's tasks within testing paradigms.

1.3 The AT Framework Architecture

The novel concept of the AT Framework is the ability to create the test once and use it for testing any application version of the pertinent software, with applicability being defined in a dynamic fashion, using rules. As stated above, the AT Framework can be used for creating tests for any software with multiple versions or for multiple software programs which share parts of their code base, e.g. some modules, and have a part of the testsuite in common.

The tests of all software program versions exist at some place; this can be realized using a single repository or multiple repositories. Each program version then includes rules, in the form of annotations, which specify the exact tests that should be extracted and applied on the particular software version. Annotations are integrated with the code, and are syntactically placed before the test classes or test methods, and each annotation indicates the location from which testsuite elements should be fetched, as well as criteria regarding the selection of tests according to the version of the software program being tested. Effectively, annotations dynamically create a *tests taxonomy*, and each test is classified under one or more taxonomy branches; provisionally, tags can be associated with these classifications to designate whether some test is applicable to a particular range of versions. Figure 1.1 illustrates the classification of tests under the dynamically created taxonomy and tag assignment; this process is analyzed in more detail in section 1.4.

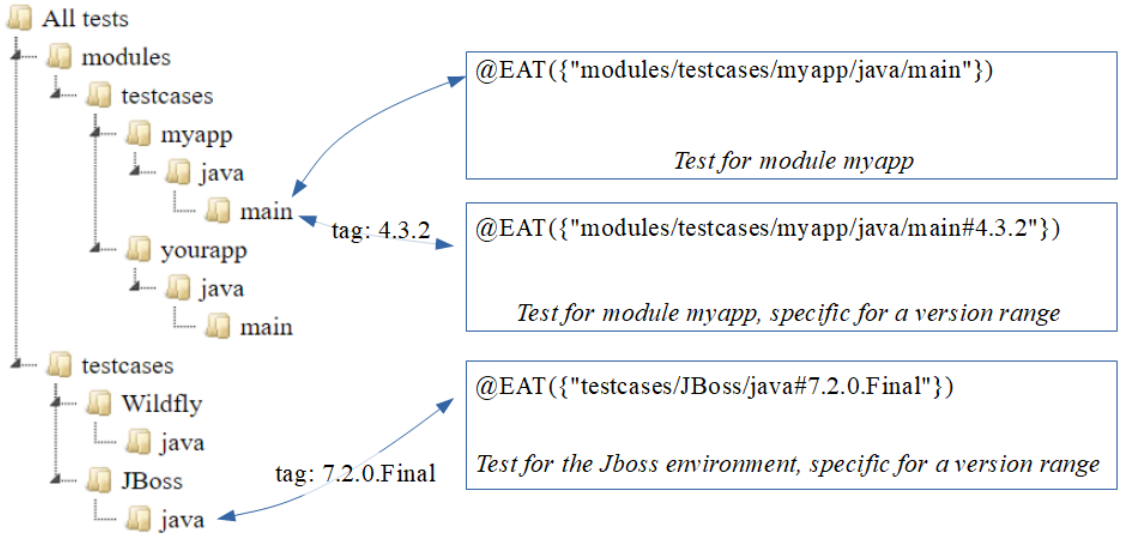

Figure 1.1: The dynamically created test taxonomy, test classification and tags.

When the executable for a specific version of the program is created, according to the relevant specification (e.g. a POM file), the annotations present in the relevant source files are processed by the *AT Framework annotation processor*, which arranges for analyzing the annotation content and gathering the tests that fulfill the rules, to assemble the testsuite of the software version currently being processed. Finally, the tests can be run using the standard test execution methods, such as a `gradle` build or an `mvn` target. The testing process workflow, according to the AT Framework, is depicted in Figure 1.2.

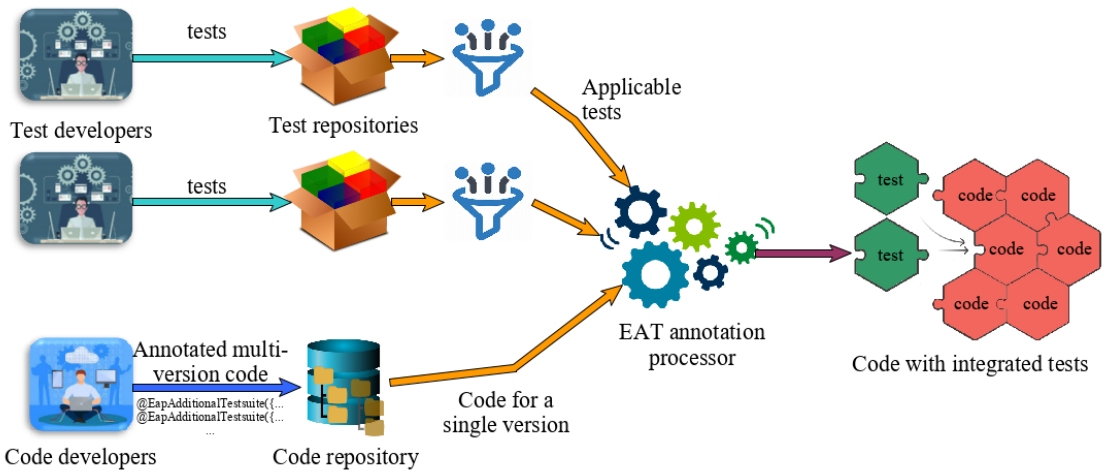

Figure 1.2: The testing process workflow, according to the AT Framework.

The architecture presented in Figure 1.2 can be also perceived as a layered approach, with each layer providing a number of functionalities on top of lower-level layers, and the top layer orchestrating the execution of the tests. These layers are as follows:

1. *The repositories and management layer*: at this layer the repositories to be used are defined. Each repository may be either local or remote. Code and tests are pushed into the repositories by developers and testers, and are then fetched when the versions to be tested are assembled. In addition, management functionalities regarding the gathering of the remote tests and the distribution of the appropriate tests, resources or configurations can be performed.

2. *The modules layer*: This layer effectively provides a comprehensive project staging area, where all elements needed to assemble and instrument the software programs to be tested are fetched and stored. These elements include the source code (`src/main/java`), the test code (`src/test/java`), the configuration files (`src/config`), as well as some scripts used for the deployment and running of test programs (e.g. `standalone.xml`, server directories, automation scripts in `src/test/scripts` and so forth). If the application runs within an application server (e.g. within the Apache Tomcat[3], the WildFly[4] or other JBoss[5] server), this layer also arranges for fetching and storing the relevant files from the respective application servers.

3. *The filtering layer*: This layer processes specifications regarding the particular version and configuration to be tested, and selectively retrieves from the staging area the elements necessary to test the designated program assembly. In particular, according to the specifications, tests and sources may be filtered according to (a) the version of the program (b) its configuration (c) the

[3] `http://tomcat.apache.org/`
[4] `https://wildfly.org/`
[5] `http://www.jboss.org/`

JVM version and its parameters (d) the environment on which it will be deployed. This layer can be considered as a set of sub-layers, with each sub-layer performing filtering according to a specific type of criteria.

4. *The assembly layer*: In this layer, a project specification containing all sources, tests, configurations and deployment descriptors pertinent to the particular version and configuration to be tested is assembled.

5. *The test execution layer*: This layer creates a fully instrumented executable for the software to be tested and arranges for executing the selected tests, using an appropriate plugin (e.g. the SureFire plugin[6]). Deployment descriptors, selected by the filtering layer, are used to select physical machines on which the instrumented program version will be run, servers on which the program will be deployed and/or resources to be made available to the executing application, as applicable.

In the next section we will describe in detail the different categories of tests supported by the AT Framework and how the AT Framework approach can be applied in the context of different use cases. The current implementation of the AT Framework can be found at [51].

1.4 Use cases for the AT Framework

In this section we present the use cases for the AT framework. Some use cases entail only testing-related functionalities, however the AT framework can be exploited to facilitate additional tasks, such as dynamic program builds. For each use case, the AT instrumentation and the implementation methodology are described.

1.4.1 Applying AT Testsuites to multiversion programs and programs that share part of their code base

Typically, the tester creates tests for the artifacts that should be tested (modules, components, full applications) and stores these tests in some repository. This repository may be either local (e.g. the `src/test/java` folder in a Java project) or a remote one (e.g. a git repository). The tester then injects into the module/application code annotations to designate the particular tests that should be applied to the code; the annotations effectively define conditions on which tests to include depending on (a) the program being built (b) the version of the program being built (c) the deployment environment of the program , e.g. the application server within which the program will be deployed and its version. These annotations are dynamically processed during the *test / sources generation* phase of the *build* lifecycle of the software program AT, and arrange for the appropriate tests to be fetched and integrated into the test locations of the project, making them thus available for execution within the *test* phase of the *build* lifecycle. Table 1.2 presents examples of the annotations used in AT Testsuites.

[6] `https://maven.apache.org/surefire/maven-surefire-plugin/`

Annotation	Explanation
@EAT({"modules/test/myapp/java/main"})[7]	Classify the test under the `modules/test/myapp/java/main` taxonomy branch
@EAT({"modules/test/.../java/main"})	Classify the test under *all* taxonomy branches that are located under `modules/test/` and ending with `java/main`
@EAT({"modules/test/myapp/java/main#4.3.2"})	Classify the test under taxonomy branch `modules/test/myapp/java/main` assigning to the "myapp" path component the tag *"4.3.2"* to designate that the test is applicable only for application versions equal to or greater than 4.3.2
@EAT({"modules/test/myapp/java/main#4.3.2*4.5.7"})	Classify the test under taxonomy branch `modules/test/myapp/java/main` assigning to the "myapp" path component the tag *"4.3.2*4.5.7"* to designate that the test is applicable only for application versions between 4.3.2 and 4.5.7
@EAT({"&/myapp/java/main")})	Same as the first case; when the & symbol is used at the beginning of an EAT path, it is substituted by a standard user-defined prefix string (in this cases `modules/test/`)
@EAT({"modules/test/myapp/&"})	Same as the first case; when the & symbol is used at the end of an EAT path, it is substituted by a standard user-defined suffix string (in this case, `java/main`)

Table 1.2: Examples of the annotations used in the AT Framework

Since in multi-version programs, different versions of a program include different sources in their build tree, the differentiation in tests between different versions is directly supported through the placement of the appropriate `@EAT` annotations in the version-specific source files.

Fig. 1.3 depicts the application of the AT framework on a multi-version program: as illustrated in the image, some tests may apply to all versions (tests in the centre of the image), some may apply to a number of versions (overlapping “slice" segments), while some tests may be version-specific. When a specific version of the software is tested, the tests corresponding to the pertinent “slice" are extracted and applied. Fig 1.4 illustrates the application of the AT framework on software artifacts with a shared code base: in order to test a version of CS1 comprising versions A.V1, B.V4 and C.V8, the relevant tests for components A, B and C will be used; correspondingly, in order to test a version of CS2 comprising versions A.V4 and B.V3, the respective tests for components A and B will be retrieved from the test repository and executed accordingly.

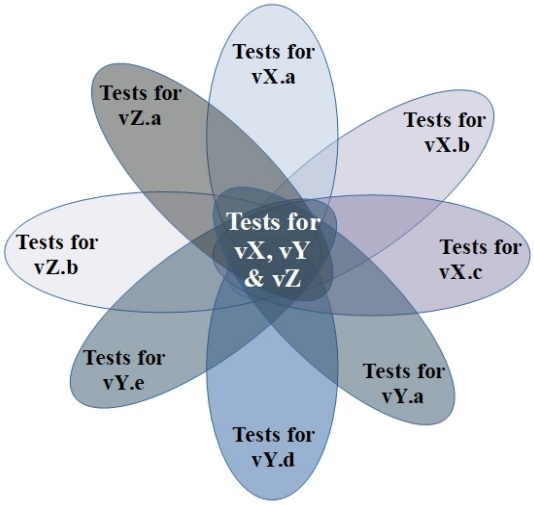

Figure 1.3: Applying the AT framework to a multiversioned software program

1.4.2 Supporting tests in different configurations

Programs may be deployed using different configurations -and probably on diverse hardware and software environments-, and appropriately adapt their code or behavior to the particular characteristics of the environment. For example, applications deployed on Weblogic servers should practice official JDBC API for database connectivity, whereas when an application is deployed on JBoss servers should use jca-JDBC wrappers [52]; similarly, when an application is deployed on Apache Tomcat 8.0 it may use the `StandardContext.getServlets()` internal API to retrieve the available servlets, however this API is not supported in Apache Tomcat 8.5 or higher [53]. The differences in the execution paths followed on when programs

[7] in the implementation, the annotation is spelled out as *@EapAdditionalTestsuite*; here it is abbreviated as *@EAT* for space and readability purposes

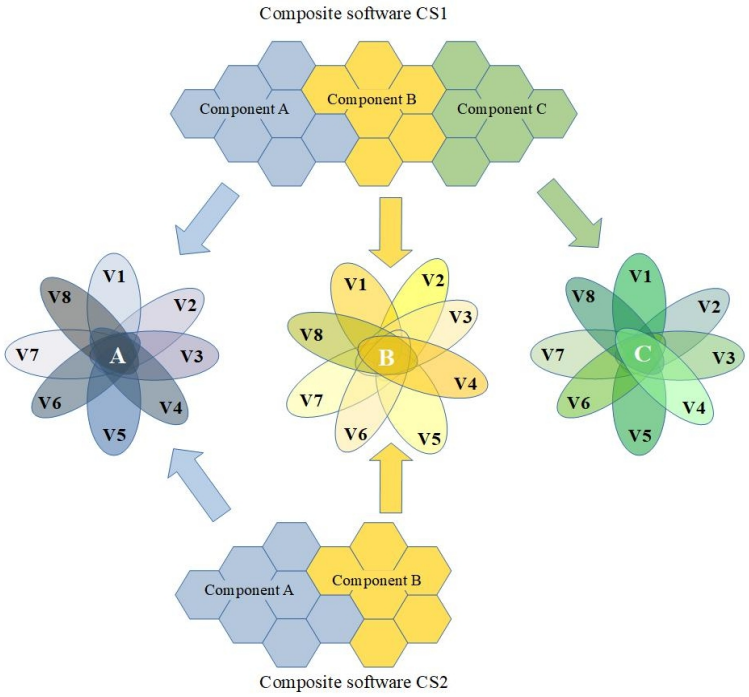

Figure 1.4: Applying the AT framework to a software program sourcing from multiple code bases

are deployed on different configurations necessitate the use of appropriately tailored tests, each suited for a set of particular configuration. Configuration-specific tests may complement configuration-independent tests that apply on the tested software version. The AT framework supports specification of tests that should be applied in accordance to the configuration under which the software program will be deployed; Fig. 1.5 illustrates the application of the AT framework to support testing in different configurations.

The specification of configuration-specific tests is again realized using the *@EAT* annotation. Firstly, the developer organizes the tests under different folders of the test tree, according to configuration to which they pertain to; for instance test cases for running a web application under the Wildfly platform can be placed in the `testcases/Wildfly/java` folder, whereas test cases for running the same web application under the JBoss EAP platform can be placed under the `testcases/JBoss/java` folder. Then appropriate annotations are used in the source code, which -besides referencing the test folders- may reference a specific version or a version range of the environment. Table 1.3 provides examples of *@EAT* annotations used to include tests specific to designated environments, while figure 1.6 graphically illustrates the organization of the tests. As shown in Figure 1.6, some tests may be common to all versions of both application containers, other tests may apply to multiple application container/version combinations, while some others may be specific for a particular version of an application container.

The AT Framework can also accommodate specifications of tests that are tailored to validate software behavior on different cloud environments. In this case, tests are defined similarly to the cases listed in Table 1.3, while *deployment descriptors* are

Annotation	Explanation
@EAT({"testcases/Wildfly/java#19.0.0.Final"})	Classify the test under the `testcases/Wildfly/java` taxonomy branch, also assigning to it the tag "19.0.0.Final". The classification designates that the test is applicable when the application is deployed for testing on the Wildfly server, while the tag limits the applicability of the test to the cases that the Wildfly server version is equal to or greater than *19.0.0.Final*
@EAT({"testcases/JBoss/java#7.2.0.Final"})	Classify the test under the `testcases/JBoss/java` taxonomy branch, also assigning to it the tag "7.2.0.Final". The classification designates that the test is applicable when the application is deployed for testing on the JBoss server, while the tag limits the applicability of the test to the cases that the JBoss server version is equal to or greater than *7.2.0.Final*
@EAT({"testcases/Wildfly/java# 18.0.0.Final*19.1.0.Beta1"})	Classify the test under the `testcases/Wildfly/java` taxonomy branch, also assigning to it the tag "18.0.0.Final*19.1.0.Beta1". The classification designates that the test is applicable when the application is deployed for testing on the Wildfly server, while the tag limits the applicability of the test to the cases that the Wildfly server version ranges between *18.0.0.Final* and *19.1.0.Beta1*

Table 1.3: Examples of the annotations used in AT framework referencing the deployment configuration

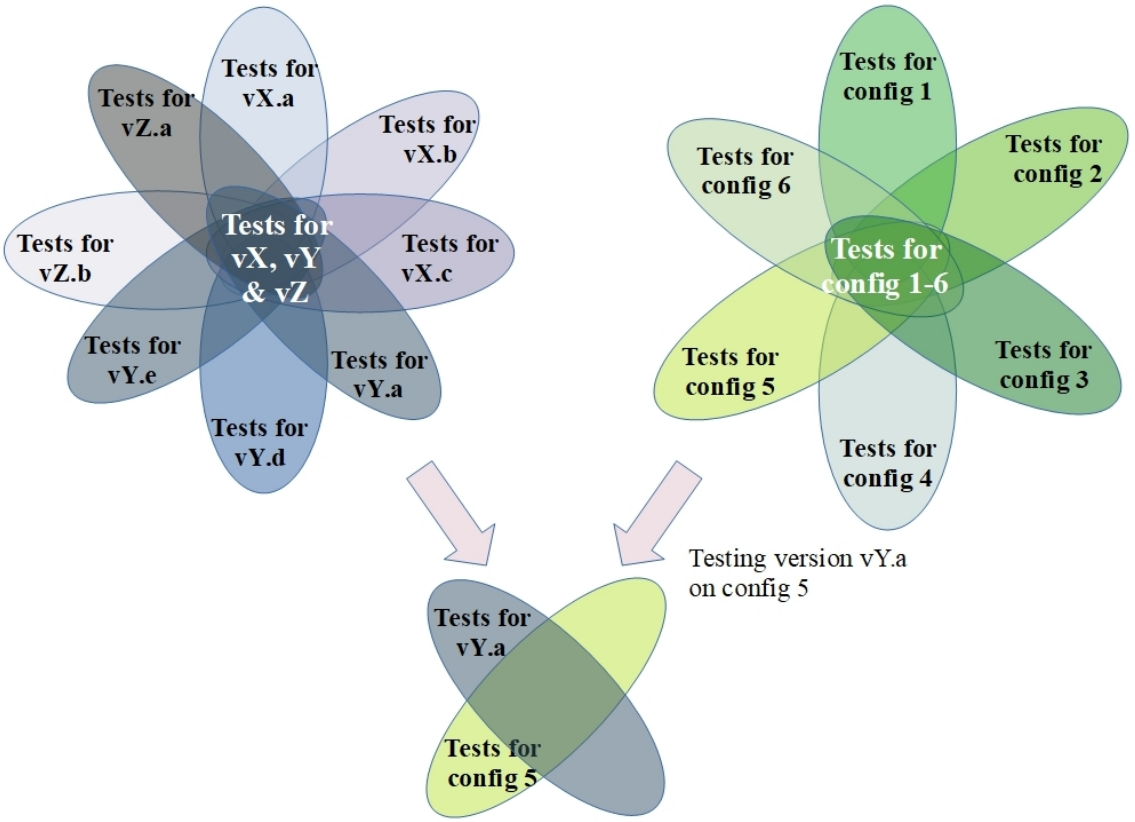

Figure 1.5: Applying the AT framework to support testing in different configurations

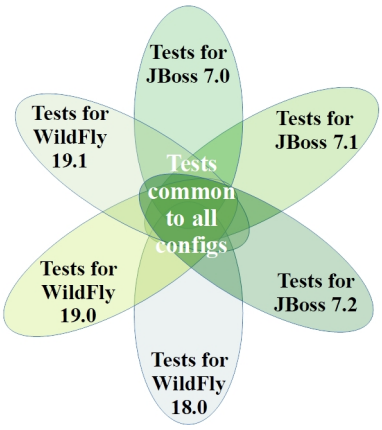

Figure 1.6: Applying the AT framework to support tests under different application containers.

used to automate the rollout of the software on the target cloud environment, against which the pertinent tests will be run. For instance, software components may be deployed to the Microsoft Azure cloud using deploy tasks [54], while the deployment on the AWS platform may be performed through AWS CodeDeploy [55] using AppSpec specifications [56]. Similar provisions exist for containerized environments, where -for instance- deployment specifications can be used to rollout applications on Kubernetes-managed clusters [57, 58]. Fig. 1.7 illustrates how the AT framework

Use case	Subsection
Applying AT Testsuites to multiversion programs and programs that share part of their code base	1.4.1
Supporting tests in different configurations	1.4.2
Applying the AT Framework for Feature-Based Program Builds	1.4.3
Using the AT Framework to Support Dynamic Program Builds	1.4.4
Supporting Test-based Software Development	1.4.5
Testing Custom Software Builds	1.4.6
Supporting Source Code Analysis	1.4.7

Table 1.4: Use cases of applying the AT framework

can be used to support cloud environment-specific tests. Cloud environment-specific tests may simply specify different API endpoints or accommodate testing of more complex functionalities that adapt to the particular cloud environment.

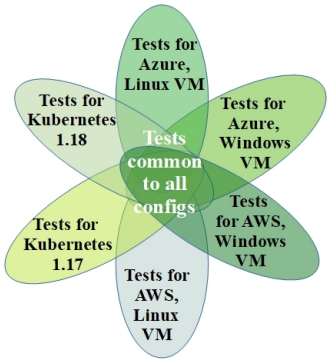

Figure 1.7: Applying the AT framework to support tests under different cloud environments

In the following subsection, the use cases of applying the AT framework on software in general, and internet software in particular are detailed; table 1.4 summarizes the use cases.

1.4.3 Applying the AT Framework for Feature-Based Program Builds

Most programs have a static configuration, in the sense that the modules to be compiled and linked to form the executable are specified statically, using notations such as Makefiles [7] or the Project Object Model (POM) [8]. Program composition may follow however different models, such as the case of software product lines, where

software products are defined by means of a unique composition of features [59], which are developed and maintained independently. Accordingly, when an executable is built for the testing process, the test selection tooling should ascertain that tests compiled in the test program are those that are pertinent to the feature selection on the basis of which the program is built.

The AT Testuites supports the testing of dynamically built programs, by providing the `@ATFeature` annotation, which determines whether a particular test will be distributed to the composition of the software program to be tested, depending on whether the attributes are included in the list of available features of the program. The `@ATFeature` annotation should designate the features that are needed to be included in the dynamic software program, in order for the test to be included in the compiled test version; for each feature, the minimum and maximum versions of each feature can be specified. Note that the specification of multiple features effectively designates a conjunction, i.e. *all* features listed in the `@ATFeature` annotation (with qualifying version numbers) should be enabled for the target program, in order that the test to be compiled into the target executable.

A notable example of a feature-based setup is the OpenLiberty server, where features may be included or not, according to the needs of the particular installation [60]. Listing 1.1 illustrates the use of the `@ATFeature` annotation in the case of the OpenLiberty server. The `@ATFeature` annotation designates that the particular test should only be used if *all* of the features *jaxrs*, *jaxb*, *jsonp*, *cdi*, *localConnector*, and *servlet* are included, while the *minversion* clause specifies the minimum version of each feature.

Listing 1.1: Using the @ATFeature annotation for feature-based test inclusion

```
1  @Test
2  @ATFeature(feature={"jaxrs,jaxb,jsonp,cdi,localConnector,servlet"},
3  minVersion={"2.1,2.2,1.1,2.0,1.0,4.0"},
4  maxVersion={"null,null,null,null,null,null"})
5  public void testJaxRs() throws Exception {
6     String result = httpCall("rest/resource");
7     Assert.assertEquals("<?xml version=\"1.0\" encoding=\"UTF-8\"
          standalone=\"yes\"?><model><name>Niki</name></model>", result);
8  }
```

The AT framework extracts the features that are enabled from the `FEATURE_LIST` environment variable, which should be set to point to a text file, where each line designates a feature that is included in the software built, along with the version of the feature; Listing 1.2 illustrates a sample file for the OpenLiberty server. Note that the contents of this file only serve the purpose of providing the information about enabled features to the AT framework, while the software itself may employ any method of its choice to handle feature activation (e.g. a configuration file). Naturally, the feature list passed to the AT framework should match the feature list that is actually enabled for the software. Listing 1.3 depicts an OpenLiberty server configuration file which matches the feature list specified by the file shown in 1.2.

Listing 1.2: Indicative list of enabled features for OpenLiberty

```
jaxrs, 2.1
jaxb, 2.2
jsonp, 1.1
cdi, 2.0
localConnector, 1.0
servlet, 4.0
```

Listing 1.3: OpenLiberty Server configuration file

```
<server description="new server">
<featureManager>
<feature>jaxrs-2.1</feature>
<feature>jaxb-2.2</feature>
<feature>jsonp-1.1</feature>
<feature>cdi-2.0</feature>

<!-- Enable the following features to run tests with Arquillian managed
    container -->
<!-- tag::localConnector[] -->
<feature>localConnector-1.0</feature>
<!-- end::localConnector[] -->
<!-- tag::Servlet[] -->
<feature>servlet-4.0</feature>
<!-- end::Servlet[] -->
</featureManager>

<httpEndpoint id="defaultHttpEndpoint" httpPort="9080" httpsPort="9443"
    />
<webApplication location="guide-arquillian-managed.war" context-root="/
   "/>
</server>
```

A more complete example of feature-based testing applied on the OpenLiberty server can be found at [61].

1.4.4 Using the AT Framework to Support Dynamic Program Builds

The feature-based software builds presented in section 1.4.3 presuppose that the functionalities needed to be included in a specific program build are organized into features; however software program builds may be custom-made to include arbitrary sets of functionalities, in order to match the requirements of software use in a particular environment. Especially for extensive, functionality-rich software artifacts that evolve through continuous development, only a small subset of the functionalities may be pertinent to particular users in specific environments: for instance, a web server typically includes a multitude of authentication methods, only few (if any) of which are used in any particular installation.

The AT Framework allows the building of program executables to include a

specific set of arbitrarily selected functionalities, to match the requirements of program usage in specific environments: to exploit this feature, developers compose a test-set in which the desired functionalities are invoked. The tests within this test-set may be already existing in the program's test code base, or custom-made for this purpose. The test can be developed using either the information of the `ATFeature` annotation, which specify all the software features / modules that are needed for the specific tests to run and succeed, or using test source dependency analysis, in combination with the known lists of software program (e.g. server and client) artifacts / modules, in order to build the specific software distribution only from the modules (along with their dependencies) that are needed for the specified test set. Effectively, instead of manually composing the program object model (POM) to be used in the generation of the executable, developers create a "POM-by-example". When this test-set is available, the AT framework applies dependency analysis to identify the code artifacts that should be included in the executable, generating thus a minimal executable, within which only the modules delivering the specified functionalities and their dependencies are included.

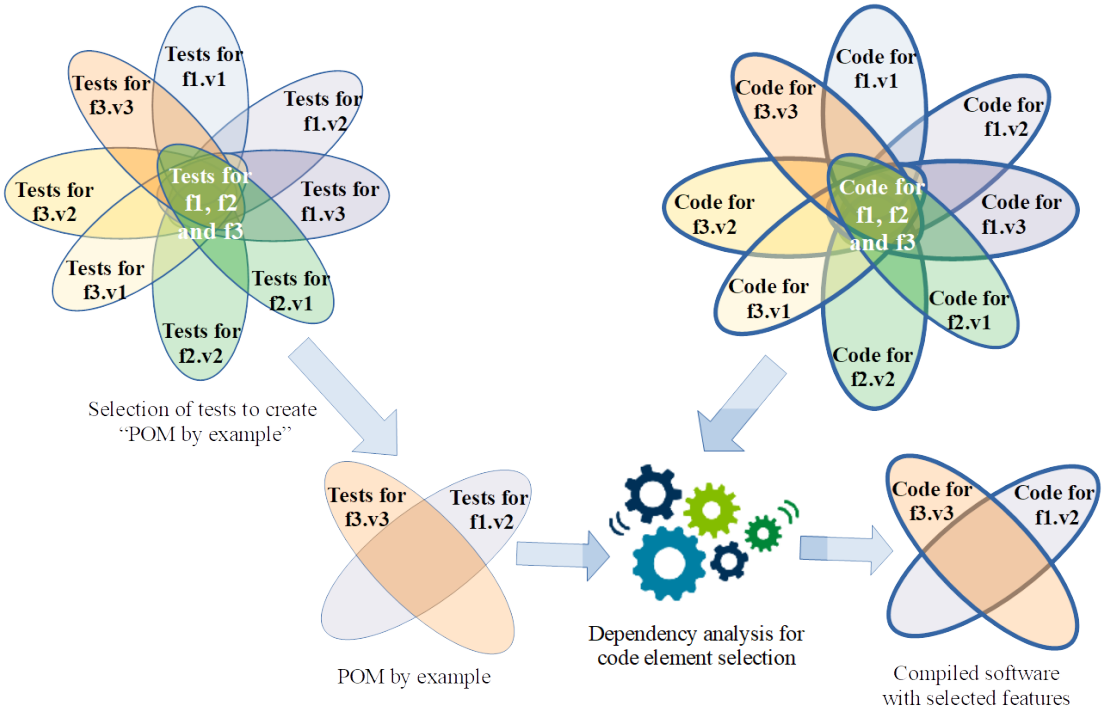

Figure 1.8: Applying the AT Framework to support dynamic program builds

Furthermore, using this AT feature, statistics could be created in order to detect the modules that have the maximum usage among the tests of the AT Framework. For example, a specific module that is used from the majority of the tests, should be included almost in all the distributions, while a module that is used only for

few tests, should not be included except if some of these few tests are needed for specific usage. Other parameters, besides percentage of usage, could also be taken into account when choosing the modules that will be used to create the software to be distributed. Some of them could be security, source code footprint, available APIs (Application Programming Interfaces) exposed, etc.

The AT Framework operation for creating dynamic programs based on a test set is depicted in Figure 1.8.

1.4.5 Supporting Test-based Software Development

According to the software engineering methodologies, acceptance tests can be developed directly from the use cases [62]; therefore, test development may proceed in parallel with the development of the main software. This aspect is stressed in the test-driven development process, where automated tests are always created prior to developing functional code in small, rapid iterations [63, 64, 65, 66]; subsequently, the code is developed, tested and debugged, and the code is considered complete when all tests are satisfied. Test-driven development is considered an integral aspect of many agile development processes, with tests being crafted either on the basis of acceptance tests [41] or from tests spanning the whole development process, including acceptance and functional tests [42, 43]. Test-driven development is reported to exhibit a number of benefits over other approaches, including a smaller defect rate after Functional Verification Test (FVT), simplification of integration and easier adaptation to changes [63].

The AT Testuites supports the test-based software development process, by providing a pipeline where the tests available for a software program, considering the environment/context that they will be tested in (c.f. 1.4.1) and the feature list to be incorporated in the executable (c.f. 1.4.3) are automatically gathered and tested against the target software program, leading to the compilation of comprehensive testing reports. More specifically, this pipeline proceeds as follows:

1. Firstly, the sources needed to compile the program executable are retrieved and staged.

2. Subsequently, the tests pertaining to the particular version of the program, taking into account the features that will be compiled in the executable (c.f. subsection 1.4.3), as well as the environment within the program will be tested (c.f. subsection 1.4.2) are extracted and staged.

3. The program is compiled to produce the executable format, e.g. a `jar` archive.

4. Each of the tests is compiled using the `jar` archive produced in the previous step; if the compilation fails, then the current stage of the program development has not appropriately realized the contracts needed by the test, and therefore it is included in the final report as a failing test. If the test is successfully compiled, then it is run and the test results (either success or failure) are included in the final report.

While the procedure described above is tailored to the Java language, which is supported by the current implementation of the AT Testuites framework, it can be directly applied to any language or case where the program to be tested, excluding the code realizing the entry point, can be built in a form that can be used as a library. This applies to all interpreted languages (e.g. Python, PHP etc). For cases where the program to be tested is not built as a library *and* the executable cannot be used as such (this case notably includes Windows executables), still the methodology can be directly used, with the worst case requiring that each of the tests is separately compiled together with the source code of the program to be tested.

1.4.6 Testing Custom Software Builds

As discussed in section 1.4.4, feature-rich programs may be customly built for particular environments, in order to include in the executable file only those features that are necessary for their operation within their operational context, and the AT framework can support a “POM-by-example" approach to facilitate the specification of these builds. Conversely, it is possible that developers have carefully crafted the POM for specific environments, and under such a context the testing procedure should adapt itself to the developer-provided POM, separating the tests that fail to compile due to the fact that the POM in effect does not include the modules necessary for the tests to successfully compile and later execute from the tests that *do compile* but fail. This separation is important for the test result interpretation, since the former class of test failures does not signify some software bug but rather the fact that the failed test does not apply to the specific constrained software configuration, while the latter class of test failures corresponds to some software bug.

The AT framework supports testing of custom software builds, by applying the procedure described for supporting test-based software development (c.f. subsection 1.4.5): each applicable test is firstly checked whether it can be compiled against the software executable that is built by the POM in effect; only tests that pass this stage successfully are then executed and accounted in the final test results. Tests that fail to compile can be separately reported.

1.4.7 Supporting Source Code Analysis

Source code analysis has evolved as an important aspect of the code testing and debugging process. Static analysis can be used to detect a multitude of issues in the source code, including memory leaks [67] or -more generally- resource leaks [68], security flaws [69] and so forth. While a number of source code problems are localized in specific procedures and can be detected at development time through code smells [70] within the development environments [71], a considerable number of code smells are manifested at the integration phase [72, 73].

The Additional Testsuite framework provides support for assembling the source code of the software programs and conducting source code analysis on the whole code bulk, facilitating the identification and correction of bugs in the interoperation of code modules that are compiled into the executable program. Effectively, after the Additional Testsuite framework has determined the components that will be built in

the software executable, either by consulting the executable program's object model [8] and/or by considering the modules needed for the realization of the features to be included in it (c.f. subsection 1.4.3), it creates a bundle containing the identified source modules and submits them for checking to appropriate static analysis software (e.g. SonarQube [74]). Then, the results of the static analysis are collected and presented to the developer team for further perusal.

1.5 Tooling for the AT framework

The developer's/tester's work with the AT framework can be supported with a number of tools that underpin and facilitate various aspects of the compilation and maintenance of the test suite. In the following paragraphs, we present some important tool functionalities.

The test suite refactoring tool. Existing projects have been developed using the standard development practices, i.e. versions are created and maintained in an independent fashion, and each branch contains a copy of the test suite, appropriately customized for the particular version. In order to apply the AT framework methodology, unique test methods must be discerned, and each such method should be tagged with the identifiers of the versions that it should be applied to. Since this work is tedious and error-prone, a test suite refactoring tool can transform the dispersed version-oriented test suite code branches to the format suitable for direct use with the AT framework.

The tool processes the individual branch test code branches, and within each branch it identifies the test methods that are executed. Then, it check whether test methods belonging to different branches and having same names are identical regarding their implementation or not. A simple comparison can be performed at source code level, while a future implementation may compile the sources and then use intermediate code-level comparison (e.g. through the Javassist library [75, 76] for Java or through the Mono.Cecil library [77] for the .Net framework), to more effectively identify identical implementations, e.g. cases where variables have been renamed. When all comparisons have been made, the test suite refactoring tool compiles a list of distinct method implementations, and tags each method implementation with appropriate `@EAT` annotations listing the identifiers of the versions that it appears in, and finally it generates the source code tree to be imported to the AT framework.

Inspecting the test code for a particular configuration. Testers and developers may want to inspect the specific tests that apply to some particular software build, which may designate specific version usage, feature list, configuration parameters etc. The AT framework includes the provision to write the source code of all tests that would be executed in the currently considered software build to a specific location at the file system, and developers & testers may then examine the code from there, using standard development and testing tools.

Flagging of annotation discontinuities. While maintaining the test source

code base, developers and testers may enter incorrect `@EAT` annotations, resulting in cases where, in some software configuration, not all needed tests are applied, or some tests are erroneously utilized. The AT framework may flag cases where the provided annotations lead to discontinuities regarding the versions on which tests apply, and subsequently the developers/testers can examine whether these discontinuities are intended or not. For instance, if the `@EAP` annotations indicate that a specific test should be applied in EAP versions 6.1.x, 6.2.x and 6.4.x, but not in the intermediate version 6.3.x, it may be the case that the developer/tester has not appropriately entered the `@EAP` annotation that would associate the test with version 6.3.x, but it is also possible that the test is not applicable to version 6.3.x. The annotation discontinuity checker tool flags such cases for further scrutiny by the developers/testers. Similarly, the annotation discontinuity checker tool flags cases where

- some test appears only in some specific version and does not appear in any direct predecessor or successor, since the version number may be misspelled (e.g. 7.3.x instead of 6.3.x), the test may have been incorrectly omitted from predecessor or successor versions, or the test may be inappropriately applied altogether.

- cases where a specific version is referenced multiple times, both for controlling redundancy and to cover cases where the second occurrence should have been modified to correspond to some other version.

1.6 Evaluation

The AT Framework has already been applied on the testsuites of a number of real-world software cases, notably including the JBOSS community edition server, Spring Boot, Active MQ and so forth. A list of cases in which the AT Framework has been successfully applied can be found at [78]. Two characteristic cases are the JBoss Enterprise Application Platform server CE (cf. Fig.1.6), which focuses on multiversion and shared code aspects, and the Openliberty server (cf. Listings 1.1 1.2 1.3), which focuses on the feature-based testing aspects. In the next subsections we elaborate on the insights gained from applying the AT Framework on these software projects. In particular, the evaluation addresses the following research questions:

RQ1) to what extent does the AT Framework contribute in reducing the number of test code bases in multi-version software?

RQ2) to what extent does the AT Framework contribute in reducing the number of test code bases in feature-based software compositions?

RQ3) to what extent does the AT Framework reduce the number of test method implementations present in the test code bases?

RQ4) is the workload of adding the necessary `@EAT` annotations manageable and less than the workload needed to manage the separate test code bases?

	Shared code base	Non-Shared code base
Single version	Tests developed to verify the functionality of modules and apply to a specific version of a project	Tests developed to verify the functionality of server-specific code and apply to a specific version of the server
Multiple versions	Tests developed to verify the functionality of modules and apply to a multiple version of either a single project or both projects	Tests developed to verify the functionality of server-specific code and apply to multiple versions of a single server

Table 1.5: Typology of tests according to the dimensions of (a) shared vs. non-shared code and (b) single-version vs. multiple version tests

1.6.1 Applying the AT Framework to the JBoss Enterprise Application Platform (Community edition)

The JBoss Enterprise Application Platform Community Edition (JBoss EAP-CE) is an open-source Java based application server used for building, deploying, and hosting highly-transactional Java applications and services. The technology developed and used in JBoss is also available under the WildFly branding [79], and therefore the two projects use a considerable amount of shared code base. Furthermore, each of the JBoss EAP and WildFly servers has insofar released a number of releases, therefore these two projects clearly fall in the case of "multiversion programs and programs that share part of their code base" described in subsection 1.4.1.

JBoss EAP-CE and WildFly include a vast number of tests; some tests pertain to the shared code base (JBoss EAP modules), and are thus applicable to both projects, while other tests apply to the non-shared code base (i.e. the code developed specifically for one of the servers). In both cases, a test may apply to a single version or to multiple versions. Table 1.5 depicts the typology of tests according to the dimensions of (a) shared vs. non-shared code and (b) single-version vs. multiple version tests.

To assess the effectiveness of the AT Framework in relation to the research questions RQ1-RQ4, we initially extracted the test suites of the JBoss Servers (JBoss EAP 6.1 to 7.2 and WildFly 10.1, 13.0, 17.0 and 20.0) and extracted statistics on the number of test files and their distribution across versions. Subsequently we manually refactored according to the AT Framework and recomputed the statistic measures. In order to perform refactoring, the distinct test cases from all versions of each software module were first discerned, and, subsequently, appropriate AT framework annotations were added to each one of the distinct test cases to designate the software versions to which the test case applies. Please note that two test cases are considered *distinct* if their code is not identical: therefore if -for instance-

	Original test suite	Refactored test suite
Number of test code bases	17	1
Number of test method instances within the test code bases	9344	1150
Number of @EAT annotations	-	1150
Avg. number of taxonomy branch associations per annotation	-	8.12

Table 1.6: Statistics of the original and the refactored testsuites for JBoss EAP

some software has n versions, $SV_1, SV_2, ..., SV_n$ and the test code base of each of the versions SV_1, SV_2, SV_k contains some test case TC_1, while the test code base of each of the versions $SV_{k+1}, ..., SV_n$ contains an updated version of TC_1 which will be denoted as TC_2, then the refactored test code base will contain exactly one instance of TC_1 and one instance of TC_2, with the instance of TC_1 containing `@EAT` annotations linking TC_1 to versions $SV_1, SV_2, ..., SV_k$, and similarly the instance of TC_2 containing `@EAT` annotations linking TC_2 to versions $SV_{k+1}, ..., SV_n$. Finally, we examined the effort of creating and modifying the test suite, both in the original test base and in the refactored test base. The refactored test suite is available in [51]. The statistics of the original testsuites and the refactored testsuites are depicted in table 1.6. Figure 1.9 illustrates statistical information regarding the number of taxonomy branches associated with the tests of the refactored code base. As shown in the figure, approximately 55 % of the refactored tests are associated with a number of taxonomy branches ranging from 6 to 10, while 24% of the tests are associated with more than 10 taxonomy branches.

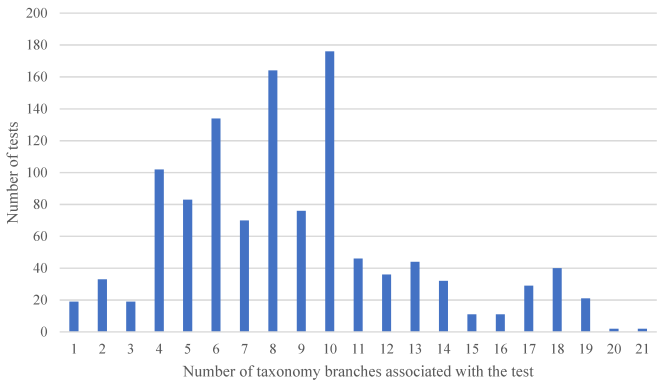

Figure 1.9: Statistics on the number of taxonomy branches associated with the tests

The 17 test code bases listed in table 1.6 to comprise the original test suite, correspond to the different versions of the EAT servers (11 distinct versions for

EAP, and four versions for WildFly, which each one of the brandings also contains a “version independent" test code base, applying to all versions). In the refactored test suite, only one code base exists and tests are correlated to server brandings and versions through `@EAT` annotations. Since a single annotation may reference multiple branches, a distinct test method implementation need only appear once in the refactored test code base and is associated with all pertinent versions by listing these versions in the `@EAT` annotation placed within the file; on the contrary, in the original test suite distinct test method implementation should be replicated in all test code bases to which it applies, hence the overall number of test method implementations is significantly higher. As shown in table 1.6, the average number of version specifications per annotation is 8.12: this means that a single test applies approximately to eight versions of the JBoss servers, on average. Figure 1.10 depicts the distribution of tests across the different brandings and versions of the JBoss Servers.

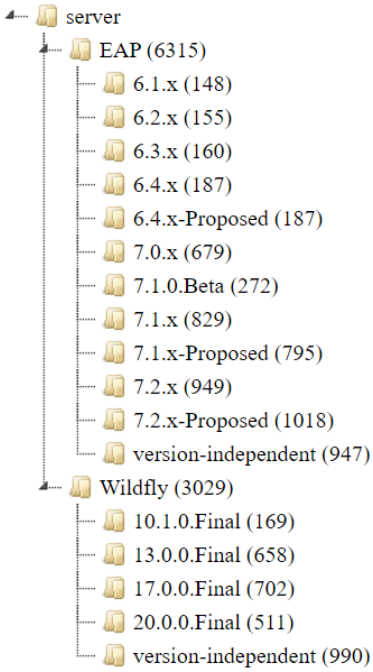

Figure 1.10: Distribution of tests across the different brandings and versions of the JBoss Servers

Many tests are indirectly included in the tests suite applicable to a specific JBoss server version, through (a) their inclusion in the test suite of a module and (b) the inclusion of the module to the specific JBoss server version. Similarly to the case of main applications, tests may be associated with a specific version or a version range of a specific module; in such cases, the test is included in main applications versions that utilize a module version that contains the test. Figure 1.11 illustrates the number of tests assigned to the different modules utilized by all considered versions of the JBoss servers. For conciseness purposes, the distribution is depicted limited to the module level, while the module version level is contracted.

The ability to list each test method once and associate it with all pertinent brandings and versions of the JBoss servers, has a significant on the maintainability

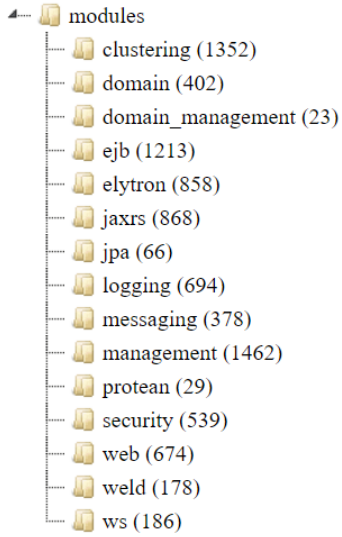

Figure 1.11: Distribution of tests across different modules utilized by the range of JBoss Servers

of the test code base. Table 1.7 lists the actions that should be performed to maintain the test code base, taking also into account the evolution of the application code base.

Additionally, the capability to easily associate a single test against a number of versions may be exploited to pinpoint the location of bugs in the code: if a test is successfully executed against some version, while its execution fails against another, the difference in the source code of these two versions could be analyzed, in order to detect the specific part(s) of the code that constitute the root cause of the bug, and therefore should be corrected.

1.6.2 Applying the AT Framework to the OpenLiberty Server

OpenLiberty is a lightweight open framework for building fast and efficient cloud-native Java microservices [80], which supports some of the most popular Java standards and uses features [60]. Features are the discrete units of functionality by which the developer controls the elements of the runtime environment that are loaded into a particular server. By adding or removing features from a server configuration, the developer can control what functions the server can perform. Features provide the programming models and services that applications require. The features can be specified in the server configuration files as shown in Listing 1.3. Some features include other features within them, and the same feature may be included in one or more other features.

Using the `@ATFeature` annotation (c.f. Listing 1.1), it is possible to associate a test with configurations where a set of specific features are all enabled, and additionally their versions fall within specified ranges. Once appropriate `@ATFeature` annotations are inserted in the test code base, the tests are properly executed when relevant configurations of the software are tested.

The code base of the OpenLiberty server used in this evaluation utilizes seven

Maintenance use case	Without the EAT Framework	Using the EAT framework
Add a new test method	Copy the test method file into each pertinent test code base	Create an appropriate `@EAT` annotation in the test method file, referencing all pertinent versions
Modify a test method	Modify each copy of the file into all test code bases	Modify the single copy of the file
Modify a test method to better suite one or more versions	Modify the test method in the test code bases of all pertinent versions	Create a single copy of the test method file and adjust the `@EAT` annotations in both files
Drop a test method	Remove the test file from all code bases	Remove the test file from the single code base
Create a new program version	Create a new test code base, by copying all pertinent test method files	Adjust the pertinent `@EAT` annotations in the test code base. If the new version is covered by version range specifications in the existing annotations, no action needs to be performed.

Table 1.7: Actions required to maintain the test code base according to different use cases

features, (*cdi*, *ejb*, *jaxb*, *jaxrs*, *jsonp*, *localConnector* and *servlet*), and a test suite demonstrating the use of the `@ATFeature` for the OpenLiberty server has been compiled and is available in [61]. Different configurations of the OpenLiberty server may utilize specific subsets of features, and correspondingly different sets of tests need to be run against each separate configurations. Indicative sets of features that could be enabled are as follows:

1. The most basic feature-based tests are performed when at least the *localConnector* and *servlet* features are simultaneously enabled.

2. When the *ejb* feature is additionally enabled (i.e. all features *localConnector* and *servlet* and *ejb* are all enabled), the *sfsbServlet* module may deliver caching functionality, and therefore relevant test should be executed.

3. When features *cdi*, *jaxb*, *jaxrs*, *jsonp*, *localConnector* and *servlet* are enabled, many additional tests are enabled, including serialization/deserialization, security-specific tests for serialization/deserialization, as well as tests for fully-fledged Jakarta RESTful Web Services.

According to the above, four feature-based configurations were considered in this test, one for each of the feature sets listed above, plus one where all features are enabled.

The feature-based tests supported by the Additional Testsuites framework facilitate the distribution of tests to the appropriate configurations. Indeed, we can notice that the when the second configuration is used, the basic tests of the first configuration should be run alongside the cache-oriented tests, since the features enabled in the second configuration are a superset of the features enabled in the first one. Similarly, when the third configuration is used, the basic tests of the first configuration should be again run. In general, when two configurations C_i and C_j exist where $features(C_i) \subset features(C_j)$, all tests specified for configuration C_i should be executed when C_j is activated. Consequently, if the test suite configurations are maintained without the use of the Additional Testsuite framework, any modification in the test suite of configuration C_i must be manually copied to the test suite of configuration C_j. The greater the number of configurations, the more work that needs to be duplicated, while version specifications introduce an additional factor or complexity. The feature-based test distribution of the Additional Testsuites framework localizes the effect of changes to one configuration only, facilitating test suite maintenance tasks.

Table 1.8 depicts statistics on test code bases for the OpenLiberty server, (a) for the original and (b) the refactored test suites. Practically, the number of test code bases equals the count of target feature combinations (4, in this case). Notably, this evaluation experiment assumed fixed versions for all used submodules, to focus on the feature-based aspect and offer insight on the maintenance gains offered by the Additional Testsuite Framework for feature-based programs. If feature-based configurations allowing multiple versions of submodules are used, for each such feature-based configuration a number of distinct test code bases are needed, as illustrated in section 1.6.1.

	Original test suite	Refactored test suite
Number of test code bases	4	1
Number of test method instances within the test code bases	70	38
Number of @EAT annotations	-	28
Number of @ATFeature annotations	-	10
Avg. number of feature combinations on which an `@ATFeature` annotation applies	-	2.3

Table 1.8: Statistics of the original and the refactored testsuites for the OpenLiberty server

1.6.3 Additional validation experiments

To gain further insight on the effect of the Additional Testsuite Framework on the test maintenance process, we refactored the test suites of five additional open-source projects, namely Spring Boot [81] (versions 2.1.0-2.4.0), 2nd chance[8] [82] (versions 1.0.0-2.1.1), ActiveMQ (versions 5.10.0-5.17.0 for broker tests [83] and versions 2.6.3-2.16.0 for client tests [84]; broker tests and client tests are maintained in different projects in the distribution are reported here separately too) and Eclipse Vertx [85] (versions 3.6.0-4.0.3). Since none of these projects entails feature-based builds, only the multi-version aspects were explored in these experiments. The statistics of the original and the refactored test suites are depicted in table 1.9.

1.6.4 Discussion

Taking into account the results of the experiments presented in sections 1.6.1-1.6.3, in this section we elaborate on the research questions listed in section 1.6.

- *to what extent does the AT Framework contribute in reducing the number of test code bases in multi-version software?*
  The AT framework always uses a single test code base in multi-version software, whereas without the AT framework each distinct version utilizes a separate code base. The gains in the number of code bases are proportional to the number of versions of the software.

- to what extent does the AT Framework contribute in reducing the number of test code bases in feature-based software compositions?
  The AT framework always uses a single test code base to accommodate all

[8]An open source mentor-to-university student association application

	Spring Boot		2nd chance		ActiveMQ (broker tests)		ActiveMQ (client tests)		Eclipse Vertx	
	original	refactored	original	refactored	original	refactored	original	refactored	original	refactored
Number of test code bases	12	1	3	1	45	1	14	1	16	1
Number of test method instances within the test code bases	1716	146	7	4	598	18	300	22	3301	430
Number of @EAT annotations	-	183	-	4	-	18	-	22	-	430
Average number of versions to which a test method applies	1	11.75	1	1.75	1	33.22	1	13.64	1	7.68

Table 1.9: Statistics of the original and the refactored testsuites for the additional validation experiments

tests in featured-based composition. When the AT framework is not employed, the number of test code bases that need to be created and maintained equals the number of distinct feature combinations that are desired. The worst-case scenario is that any feature combination may be used, in which case the number of required test code bases equals $n!$, where n is the number of features.

- to what extent does the AT Framework reduce the number of test method implementations present in the test code bases? When multiple test code bases are maintained, test methods that apply to multiple versions or multiple feature combinations need to be duplicated in all pertinent test code bases. Since the AT framework utilizes a single test code base, the need for duplication is eliminated. The experiments reported in sections 1.6.1-1.6.3 indicate that the elimination of method duplication reduces the number of test methods in the test code bases by a percentage varying from 43% (for the 2nd Chance software) to 97% (for the ActiveMQ broker tests). The amount of reduction depends on the number of versions/feature combinations of the software, and the average number of versions/feature combinations to which a test method applies on average: when test methods are highly version-specific, the gains are smaller, whereas when test methods have a broad version coverage, the gains are more considerable.

- is the workload of adding the necessary `@EAT` annotations manageable and less than the workload needed to manage the separate test code bases? The number of `@EAT` annotations that need to be inserted in the code is typically less than or equal to the number of test methods within the refactored code base, and these annotations can be inserted upon the creation of the method. Tests that are not version/feature combination-specific (e.g. testing of basic HTTP functionality for web servers) will not be version-tagged and thus will directly apply to all versions of the software, without necessitating any maintenance of the `@EAT` annotations when new versions or feature combinations are introduced. Version-specific tests may require a varying amount of `@EAT` annotation maintenance, depending on the range of versions to which the test is applicable. Typically, a version-specific test starts to be applicable at some specific version of the software, at which point the initial version-tagged `@EAT` annotation is inserted, and may cease to be applicable from some subsequent version and onwards, at which point the ending version designation would be added to the pertinent `@EAT` annotation. In this respect, at most one additional modification per `@EAT` annotation would be typically applied to an `@EAT` annotation. On the contrary, if the AT framework is not used, the introduction of a new version or a new feature combination would require the creation of an additional copy of the test code base, and its appropriate customization to retain only the test methods required for this particular version. The existence of copies of test method implementation copies introduces the need to consistently update all copies when the implementation of a test method is modified. Conclusively, the use of the AT framework significantly reduces the effort needed to maintain the test code base.

1.7 Conclusions

In this chapter we have presented the Additional Testsuite Framework, which upgrades the classic one-to-one mapping between a program version and a statically specified corresponding testsuite into a dynamic testsuite specification, where tests to be included are designated through declarative specifications that are evaluated on-the-fly to derive the set of tests that need to be applied on the particular case. Developers create tests and annotate them appropriately to designate the software artifact/version to which they apply, the deployment environment in which they will be used, or any other pertinent aspect that may be related to testing; then upon test execution, the AT framework instrumentation extracts the test annotations and the test execution specification (which includes information on the deployment environment, software version, features included etc.) to automatically determine which tests should be applied in the particular case. The AT Framework can support all the types and levels of testing, such as unit testing, component testing and integration testing. Furthermore, the Additional Testsuite Framework offers a number of features, including multi-configuration testing, dynamic program builds, test-base software development, custom software builds and source code analysis support.

The AT Framework undertakes a different point of view in the area of testsuites. This approach upgrades the status of testsuites to “first class citizens" in the software engineering domain, rather than considering them as “subordinates" of software artifacts. The AT Framework is generic, and can be applied for any software program written in any software programming language, and can accommodate any type of testing procedure.

The AT Framework has been implemented and has been applied in a number of real-world software cases, notably including the JBOSS server, Spring Boot, Active MQ and so forth. The results are reported in section 1.6 and demonstrate that the AT Framework improves the test code base maintenance process, eliminating the need for multiple code bases and test method duplication, as well as the probability of inconsistencies due to imperfect update of copies and the extra effort that these updates incur. A list of cases in which the AT Framework has been successfully applied can be found at [78]. A tutorial covering the installation and use of the AT framework is available at [86].

In our future work we will explore additional potential of the AT Framework, including the implementation of support for model-based testing methods [87] and techniques for intermittent fault detection [88], as well as the implementation of tools for cross-analyzing test annotations and code versions to check for mis-annotated tests, the existence of which may lead to the omission of some tests in certain versions, deployment environments or features. Combination of model based testing and test based source code generation iteratively will be studied. Generation of statistics regarding code usage in relation to the available tests and exploitation of these statistics in the context of software engineering tasks, such as code refactoring and optimization, will be also considered. Finally, the combination of the AT framework with works on test case prioritization [89] will be examined.

Chapter 2

The Additional Testsuite Framework for Mobile Applications: Facilitating Software Testing and Test Management for Mobile Apps

Abstract

Mobile apps operate on a constantly evolving ecosystem, where new devices with novel capabilities are introduced, new operating system versions emerge, whereas operational environment parameters change, including network types and bandwidths. Therefore, mobile applications need to be correspondingly updated with new versions, to guarantee seamless operation over the full spectrum of ecosystem elements. Additionally, mobile applications may be released in multiple flavours, for example with different functionalities, customized user interfaces or special features. In this chapter, we present the Additional Testsuite Framework for Mobile Applications, which aims to facilitate the mobile application testing process through the provision of suitable concepts and processes for accommodating the special requirements of mobile application testing. Following the Additional Testsuite Framework paradigm [61], source code and tests are written once and are categorized and distributed to the desired versions of the mobile application software through the use of annotations.

2.1 Introduction

Mobile devices have become very popular in the last decades, and alongside with their proliferation, the usage of mobile applications has increased dramatically. In order to guarantee the seamless operation of mobile applications, they need to be tested thoroughly in different software and hardware environments, and ascertain that the desired quality requirements are met.

Mobile applications, similarly to all software, should satisfy a number of quality characteristics, the most important of which are usability, maintainability, reliability,

security, efficiency, compatibility, and functionality [90, 91]. Testing for each of these characteristics may require the application of tailored methods or the execution of specialized tests. In particular:

1. **Usability**, which refers to how a product can be used to reach a specified goal. This shows to what extent the application is understandable, easy to learn, easy to use, has a minimum error rate, and overall satisfaction with the application. These are generic factors of usability that a given mobile app should fulfill. While some phases of usability testing need to be conducted manually, a number of methods have been proposed to (partly) automate the usability testing process [92, 93, 94, 95, 96].

2. **Maintainability:** According to the ISO/IEC/IEEE International Standard for Systems and software engineering vocabulary [97], maintainability is defined as "the ease with which a software system or component can be modified to change or add capabilities, correct faults or defects, improve performance or other attributes, or adapt to a changed environment". Maintainability is therefore a multifaceted aspect, with each facet being assessed using a wide range of criteria that may include statistics on the code (e.g. size/lines of code and cyclomatic complexity), aspects that depend on the business or technological environment (e.g. change proneness, technology evolution) and aspects related to the lifecycle methodology management method, personnel skills and tooling (e.g. effort spent on each phase of the corrective or adaptive maintenance process) [98]. Maintainability metrics may be computed automatically using statistical measures or machine learning techniques [98], or be collected through observations of the relevant processes.

3. **Reliability**, which is defined as the probability of the software to function in a failure-free fashion for a given time period within a given environment and specified operating conditions [99]. A multitude of techniques for assessing software reliability is available, ranging from software defect prediction based on various code attributes and traits [100, 101] to testing different reliability dimensions including feature testing (software functionalities are correctly delivered), regression testing (evolution and maintenance activities do not introduce new bugs), load and stress testing (the software is able to sustain the usual load and very high loads) and so forth [102, 103, 104, 105, 106]. Reliability testing may be performed at architectural level [107], unit level [108] or it may be applied to refactoring processes [109].

4. **Security:** Software security is a multi-faceted aspect, including aspects of (a) the design process (e.g. security by design [110] and security by default [111]) (b) the implementation phase and (c) the deployment and operational phase [112]. In the implementation phase, faults and vulnerabilities are detected and repaired using static analysis [113], possibly complemented by manual code review. In the deployment and operational phases, different methods including reconnaissance [114] and penetration testing [115] are used to ascertain different aspects of the application, including authentication and authorization, business logic data validation, weak cryptography usage and so forth [115].

5. **Efficiency**: the capability of the software product to deliver suitable performance, taking into account the amount of utilized resources, when operating under given conditions [116]. Efficiency includes the aspects of *performance* (minimization of time needed to perform the operations), *resource efficiency* (minimization of resources needed, including memory, CPU, storage, energy, network bandwidth etc.) and capacity (the degree to which the maximum limits of a product or system parameter meets requirements [117]).

6. **Compatibility**: A compatible mobile app is one that properly works across different mobile devices, OS platforms, and browsers, or -more generally- an application that can be built and operates according to its specification across a wide range of divergent field environments [118]. Mobile app compatibility testing is challenged by the large number of target mobile devices with diverse features [119], and different approaches are used to automate and optimize the mobile app compatibility tests [120, 119, 121, 122].

7. **Functionality**: any piece of software should be able to perform the tasks specified in the relevant requirements, and functional testing is the process that measures the extent to which this goal is met. For mobile applications in particular, a number of methods have been proposed to automate functionality testing, which include test case generation, test case execution, test oracle definition, test oracle evaluation and combinations of these approaches towards full automation [123].

The need to thoroughly test mobile applications for all the aforementioned properties creates challenges, complicating the test process. A variety of automated testing methods have been developed but still, because of the continuous updates of operating system versions, software and devices, mobile applications must be updated to both (a) exploit new features that are made available and (b) maintain compatibility and smooth operation. Application evolution necessitates the corresponding update of tests, which should be extended to cover the new or enriched functionalities and the expanded hardware/software ecosystem into which the mobile applications operate. In this context, the following challenges of automated testing for mobile applications are encountered [124, 125]:

- **Fragmentation**, which is met both in software and hardware, and constitutes one of the major issues related to application testing. At the hardware level, large variety of devices is nowadays available with diverse hardware components, which in turn may have different characteristics; for instance, some devices may be equipped with an accelerometer while others may not, whereas accelerometers (where available) may vary in accuracy, response time and so forth. Moreover, while Android and iOS nowadays account for more than 99% of the total number of mobile devices [126], at least 7 versions of Android and 3 iOS versions hold a market share of at least 3% [127, 128], while the versions of tablet operating systems, such as iPadOS should be accounted for (e.g., at least 3 versions of iPadOS with market share over 3% [129]). Application developers need to cater for differences at hardware level, while they should

also exploit the features offered by the latest operating system releases, while maintaining compatibility for the older versions, since applications behave differently on each device, and usability and performance are affected [124]. This results in a massive number of tests that need to be executed to ascertain correct functionality across all hardware and software environments. However, the diversity of mobile devices and platforms is reducing the reusability and maintainability of test cases [130].

- **Changing mobile ecosystem:** Mobile applications offer access to different services through an end-to-end mobile ecosystem, composed of mobile devices, mobile networks and application servers, and each element of this ecosystem vary in time regarding their version used [124, 125]. In addition, mobile networks have different characteristics including connectivity, bandwidth, latency, congestion control and cost. Mobile applications should be tested with all the different combinations of component versions and characteristics, in order to ascertain their intended functionality. The omission of certain combinations from the testing phase exposes the production system to the risk of malfunctions or erratic behaviors.

- **Resource scarcity in mobile device:** Resource limitation is also a topic of consideration regarding the quality of mobile applications; in general mobile devices have a constraint amount of resources, as compared to stationary devices (desktops or servers); these constraints concern CPU speed, memory and disk space, screen size, power availability, network communication (to avoid excessive charging on metered connections) and so forth. Limited resources on a mobile device can degrade the performance of the application or even make it not functional.

In this chapter, we present the Additional Testsuite Framework for Mobile Applications (ATF/MA), which aims to facilitate the mobile application testing process through the provision of suitable concepts and processes for accommodating the special requirements of mobile application testing; additionally, suitable instrumentation is provided to support the execution of the processes and underpin the work of testers. ATF/MA can be used with any software programming language. More specifically, ATF/MA furnishes mobile application developers and testers with methods and tools that facilitate the management and execution of mobile application testsuites, providing support for the following mobile application testing use cases:

- Multi-version mobile application testing
- Multi-configuration mobile application testing
- Multi-environment mobile application testing
- Multi-device mobile application testing
- Using ATF/MA for feature-based mobile application development
- Testing of dynamic mobile applications

These use cases will be further elaborated on in section 2.4.

ATF/MA contributes to the current state-of-the-art in mobile application testing by introducing specialized procedures, recipes and tools that specifically address the needs and complexities of mobile application testing. It extends the Additional Testsuite Framework (ATF) [131], a generic test management framework, by introducing specialized procedures, recipes and tools that specifically address the needs and complexities of mobile application testing. Besides supporting the use cases listed above, ATF/MA can be used to underpin test-driven development or source code analysis. ATF/MA, following the paradigm introduced by ATF, uses annotations to assign properties to tests; then, during the test lifecycle of the software, ATF/MA matches the properties of the tests against the relevant properties and context of (a) the build environment, (b) the features that are included in the build and (c) the execution environment, and arranges so that the appropriate tests, modules, libraries and configuration are included in the executable program that will be used in the testing phase. The AT Framework can be applied on any testsuite of any type or level of testing, such as unit testing, component testing or integration testing [3].

To substantiate the chapter contribution to the state-of-the-art and quantify the benefits derived through the use of the proposed approach, ATF/MA is assessed considering the workload that testers need to carry out for common software engineering activities, (a) without the use of the ATF/MA framework and (b) when employing the ATF/MA framework. The experiments conducted demonstrate that the use of the ATF/MA framework reduces tester workload, while additionally the need to maintain multiple repositories of test code is eliminated, promoting consistency and maintainability.

The rest of the chapter is structured as follows: section 2.2 overviews related work. Section 2.3 introduces the proposed ATF/MA, while section 2.4 describes the use cases for ATF/MA. Finally, section 2.5 reports on experimental findings regarding the benefits reaped from the use of ATF/MA, while in section 2.6 conclusions are drawn and future work is outlined.

2.2 Related work

During the recent years, the widespread of mobile technology and the proliferation of mobile apps has fueled research and technological developments in mobile application testing. In this section, we overview work in this area.

The issue of testing mobile applications with different versions of mobile Android devices is discussed in [92]. The authors propose an online platform which supports the testing of any Android application on multiple diverse Android devices, allowing tests to be run in parallel and identifying the device models where errors occur during the execution of the tested mobile application.

Natnael Gonfa Berihun et al. [90] examine the applicability of testing automation frameworks in mobile application testing, considering the following objectives: reusability, efficiency, functionality, reliability, compatibility, scalability, and performance. This study identifies concerns and challenges related to the following aspects of mobile application testing: complexity, maintenance cost, time consumption, and fragmentation. Mobile application testing challenges are also reviewed in [132],

focusing on the issues arising from (i) mobile device diversity, (ii) mobile platform/OS diversity, (iii) mobile network operators and (iv) scripting; different types of testing for mobile applications are also reviewed.

In [124], the authors identify the main trends in Android application testing, catalogue the most widely employed methodologies and highlight the most important open issues, outlining research topics and directions.

In a wider scope, [133] discusses the current challenges faced by mobile developers in practice, such as developing applications across multiple platforms, lack of robust monitoring, analysis, and testing tools, and emulators that are slow or miss many features of mobile devices.

A comprehensive review of studies concerning mobile application testing is presented in [134], where a systematic mapping study is performed, categorizing and structuring the research evidence that has been published in the area of mobile application testing techniques, and reviewing challenges that they have reported using 79 empirical studies. Finally, [135] maps the research agenda for mobile application testing, presenting a categorization of 21 secondary studies on the basis of (i) their primary and specific topics of research (e.g. performance testing; fragmentation; test case generation; etc.), (ii) objectives of the tests and (iii) platforms used to conduct the testing. Furthermore [135] catalogues and discusses 87 research gaps highlighted in the surveyed literature, categorizing these gaps in 16 broad categories.

2.3 The Architecture of the Additional Testsuite Framework for Mobile Applications

The ATF/MA can be used for testing mobile applications, allowing testers to create each test once and subsequently use it for any version of the application that the particular test is applicable to. The tests can be stored either in a single or in a set of repositories, and the instrumentation of the Mobile AT framework automatically arranges so that pertinent tests are gathered and applied on the current test scenario, considering the application version, the features enabled, the deployment environment or any other relevant parametrization. Supporting this testing procedure paradigm is particularly important for mobile application development and testing, due to the fact that mobile devices, their hardware, their operating systems and their operating environment exhibit significantly higher variability than desktop/stationary computing devices. Additionally, mobile application developers strive to ensure compatibility with a wide range of deployment and operating configurations, including previous versions of the operating systems or network connections, to broaden the potential client base and warrant seamless application operation. The design of ATF/MA alleviates the complexity of defining and maintaining test scenarios for each individual configuration, allowing the declarative specification of the conditions under which a particular test should be included in a test scenario and the necessary tooling and instrumentation for compiling and executing the test suite that is appropriate for any particular configuration.

Declarative specifications of the associations between individual tests and test scenarios are realized using annotations; this practice is in line with widely used

practices for annotating tests [136] or for annotating code in general [137]. When the executable for a specific version of the program is being created, the annotations present in the relevant source files are processed by the *AT Framework annotation processor*, which classifies the tests according to the annotation information provided and chooses the ones that are intended for the test scenario. Finally, the tests are executed and the results are presented to the user. The testing process workflow, according to the AT Framework, is depicted in Figure 2.1.

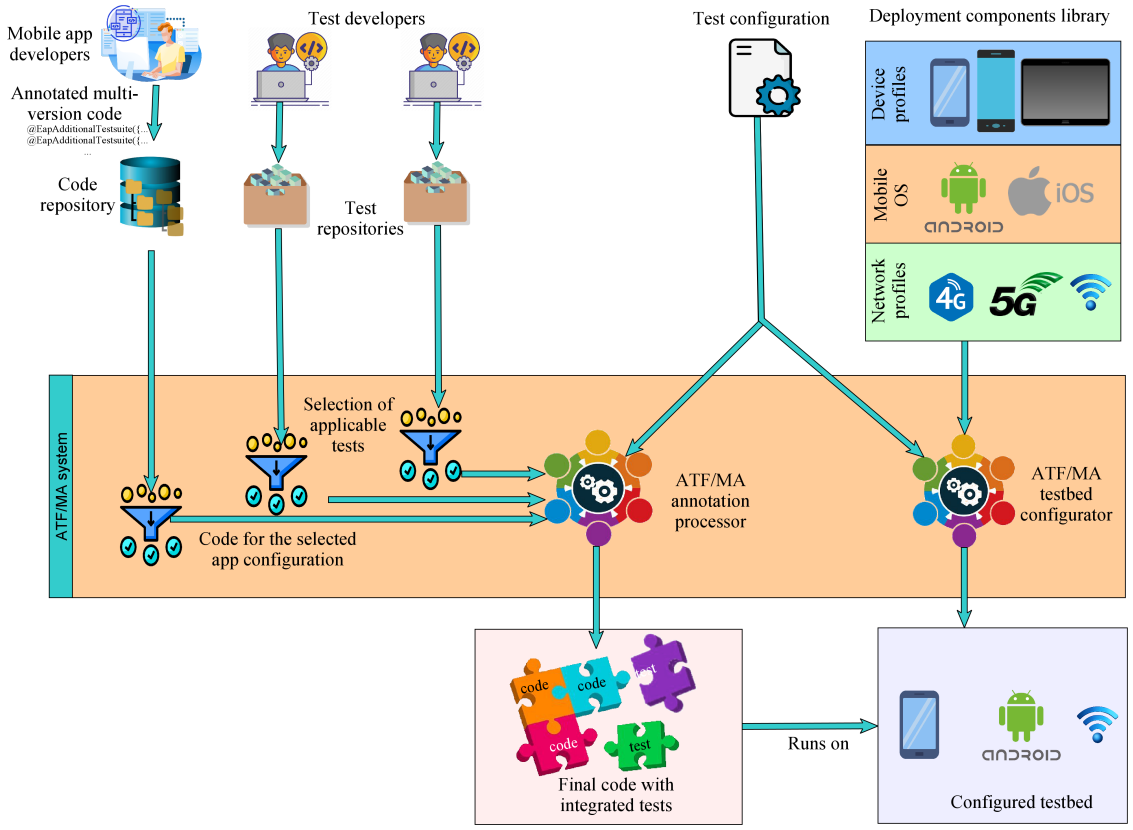

Figure 2.1: The testing process workflow, according to the ATF/MA approach.

We note here that ATF/MA does not only choose the tests pertinent to the test scenario at hand, but additionally extracts the relevant application configuration parameters needed to build the app executable dynamically, including the version of the application and the features to be included.

The overall architecture of the ATF/MA is illustrated in Figure 2.1; the components of the architecture are described in the following paragraphs.

1. *The code and test repositories*: This component hosts the repositories in which application code and tests are defined and maintained. Each repository may be either local or remote. The mobile application code and tests are pushed into the repositories, and are then retrieved to be used, according to the version to be tested. Code and test modifications are pushed into the repositories as the development and testing process evolves.

2. *The deployment components library*: This element hosts components that will be used as infrastructure and/or operating environment for the testing processes; these include executable images (physical or emulated) of mobile devices and mobile operating systems as well as network profiles regulating the connectivity mode and network speed of the mobile application during the test. During the test process, the ATF/MA system retrieves the appropriate components from this repository to create the testbed on which the test will be performed.

3. *The ATF/MA system*: this component arranges for the execution of the test, according to the *test configuration* that will be provided by the tester. Based on the test specification, the two main subcomponents of the ATF/MA system proceed as follows:

    - *The ATF/MA annotation processor* retrieves the application code and test sources from the repositories and the annotations therein are matched against the test configuration specifications. Only the tests and code pertinent to the test configuration are retained, and are used to create the final test executable, which integrates the tests that need to be performed.
    - *the ATF/MA testbed configurator* which processes the test configuration specification to identify the pertinent testbed configuration. The relevant elements are extracted from the *deployment components library module* and used to create the testbed configuration. The executable code created by the *ATF/MA annotation processor* is deployed on the testbed, and the test scenario is then executed.

The complete testing procedure, with detailed instructions on how the tests are described and executed in the current implementation of the ATF/MA are described in [51].

The ATF/MA can be used to support multiple testing needs, including multiversion mobile application testing, multiple configurations of the mobile applications, multi-environment testing (e.g. different versions of the mobile operating system), multi-device testing, multi-network testing and feature-based testing. These testing scenarios are presented in detail in Section 2.4.

2.4 Use cases for the ATF/MA

In the following paragraphs we elaborate on the testing scenarios that can be served by ATF/MA.

2.4.1 Multi-version mobile application testing

Similarly to all applications, mobile applications evolve and new versions are created, to accommodate improvements, new features and bug fixes. Different versions of the same application are expected to have some code in common, while some other code portions, corresponding to updates and new features will differ. Similarly, different versions will share some tests, both at unit level and integration level.

Developers and testers populate the code and test repositories with the relevant artifacts. Subsequently, during the testing phase, the ATF/MA extracts from the test configuration the version(s) to be tested, builds the corresponding executable(s), configures the target platform and conducts the test(s) on the designated platform configuration, including the deployment and running of the executables, and the production of results. The testing procedure is depicted in 2.2.

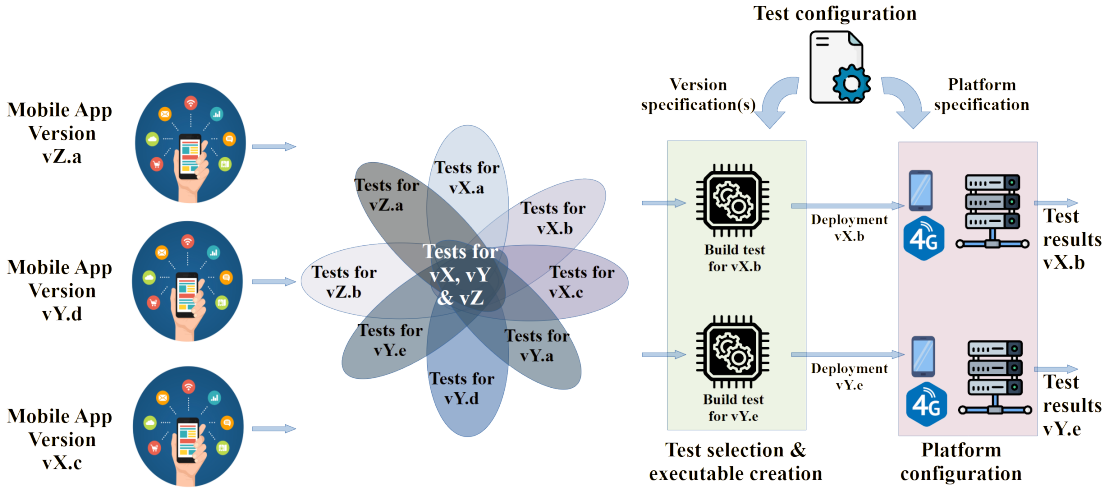

Figure 2.2: Multiversion application testing using the ATF/MA.

For example, let us consider a mobile app where in version 2 a registration facility is introduced that entails the validation of the user's email. To this end, the second version includes code to verify that the email of the user is syntactically correct, and this code is complemented with the relevant tests, which need to be executed only against version 2 of the application. Listing 2.1 illustrates how ATF/MA accommodates this use case, by employing the appropriate annotation (`@EAT`) in the test code; in this annotation, the anchor `#2.0.0` designates that the test is applicable only to versions 2.0.0 and onward; a variant of this syntax `#v1*v2` can be used to specify that the test is applicable to versions from *v1* up to (and including) *v2*.

Listing 2.1: A test that needs to be executed for a specific mobile app version

```
1
2          /**
3          * Unit tests for the EmailValidator logic.
4          */
5          @EAT({"modules/testcases/jdkAll/Android/junitbasicsample/testing-
               samples/unit/BasicSample/app/src/test/java#2.0.0"})
6          public class EmailValidatorTest {
7             ...
8             @Test
9             public void EmailValidatorInvalidEmailTwoDotsReturnsFalse() {
10               assertFalse(EmailValidator.isValidEmail("test@email..com"));
11            }
12
13            ...
```

```
14      }
```

When the Mobile AT Framework is executed for version 1, it will not include the above listed test, because the version does not satisfy the criterion specified in the annotation. Similarly, version 1 may include tests that are not executed in version 2 (e.g. for deprecated/removed features), while tests are allowed to be associated with multiple versions.

2.4.2 Multi-configuration mobile application testing

When a mobile application is executed, it utilises resources from the user mobile device, the access and backbone networks, the servers that provide functionalities related to the application and so forth. Different configurations may vary in one or more of these elements, and the application needs to be tested on a sufficient number of configurations to warrant smooth execution for users. Notably, a change concerning the user mobile device element (e.g. Android vs. IoS) may signify that a completely different code base is used for building the executable (for instance when the application is developed as a native app for each of the target platforms), however the source code may be identical, if the application is build using cross-platform tools such as Flutter [138].

Depending on the configuration used, the equivalent tests are extracted and used for the testing process. In addition to the tests, the relevant sources are extracted from the repositories, the executable files are compiled, the appropriate platforms are configured as specified in the configuration file, and each executable is deployed on the corresponding platform; finally, the executables are run to have the tests executed. Fig. 2.3 illustrates the testing process when multiple configurations are specified.

For example, let us consider the case where an application needs to access different servers providing back-end services, due to legal requirements imposed in different regions. The base endpoint of the server utilised by the application may be specified in a properties file, and two configurations designating different resources may be included in the test configuration, as shown in listing 2.2, which illustrates a sample test configuration file, hosting two different tests corresponding to distinct builds; in each build, a different `application.properties` file is bundled.

Listing 2.2: Multiple configurations specification

```
1       <tests>
2       <test name="testConfig1">
3       <build>
4       <testResources>
5       <testResource>
6       <directory>src/test/properties.server1</directory>
7       </testResource>
8       </testResources>
9       ...
10      </build>
11      </test>
```

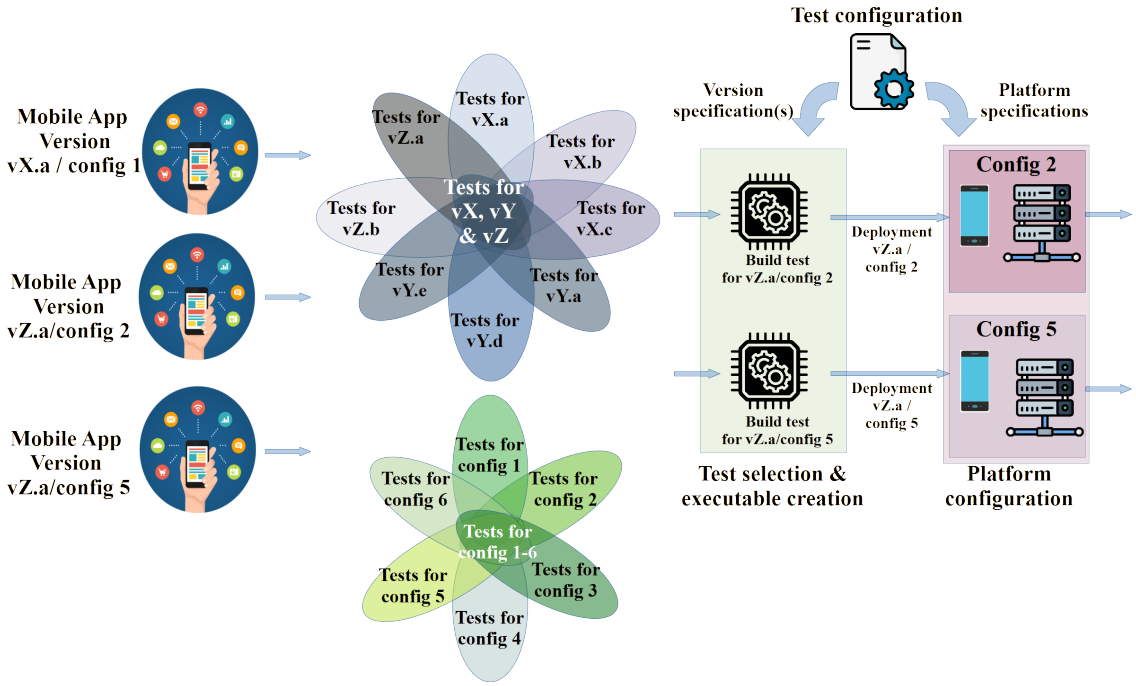

Figure 2.3: Multi-configuration mobile application testing using the ATF/MA Framework.

```
        <test name="testConfig2">
        <build>
        <testResources>
        <testResource>
        <directory>src/test/properties.server2</directory>
        </testResource>
        </testResources>
        ...
        </build>
        </test>
        </tests>
```

Besides different base endpoints, the individual properties files may designate additional variations that are taken into account by the code and validated in the test phase, e.g. timeouts, default image resolutions and so forth.

2.4.3 Multi-environment mobile application testing

User devices executing the mobile applications may run different version of the mobile operation system; each version offers a different API level [139], and each API level may offer varying degrees of support for certain features realized by the mobile applications (e.g. Android 11 offers improved support for conversations, i.e. real-time, bidirectional communications between two or more people [140]). The level of support provided by the mobile operating system may affect among others

the availability of features, the code compiled to realize their implementation, and the code needed to perform tests.

Furthermore, multiple aspects of the execution environment of an application may vary, including network availability and speed (no connection; 3G; 4G; 5G; wireless LAN; etc.), power state (plugged in; power saving mode; normal battery operation mode; etc.), available RAM, and so forth.

The ATF/MA supports the multi-environment mobile application testing, by allowing multiple platform specifications to be accommodated in the test configuration. For each such specification, the ATF/MA extracts the appropriate code and tests from the repositories, compiles the test executable, configures the specified platform and deploys the respective executable on it, and finally runs the tests and collects the results. Again, the tests that are pertinent for each configuration are specified in the repository declaratively, using annotations.

Fig. 2.4 illustrates the testing process when multiple configurations are specified.

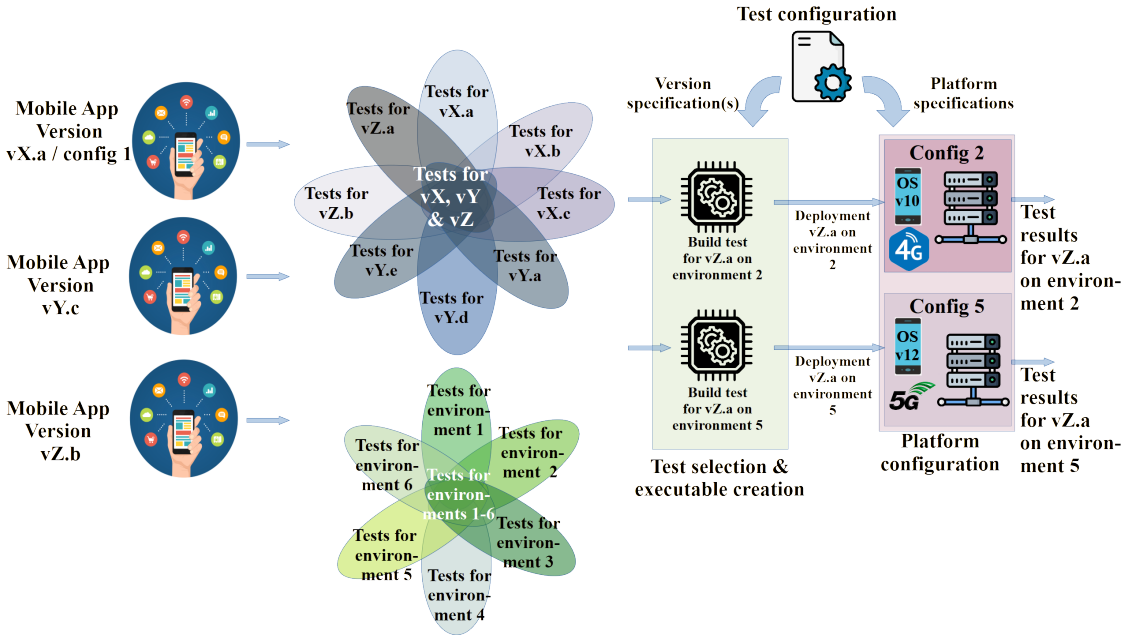

Figure 2.4: Multi-environment mobile application testing using the ATF/MA Framework.

For example, in order to test an application using a WiFi network connection, the test set that is executed can be configured to include a specific test that arranges for connecting to a wireless network using the WiFi suggestion API [141]; the `@Order` annotation of JUnit will be exploited to arrange that this test is run first, thus ensuring WiFi availability to all other tests (c.f. Listing 2.3). If the particular test is not included, then the application will be tested without network connectivity. Other network speeds (e.g. 3G, 4G) may be also tested by appropriately throttling the WiFi network connection, i.e. setting a maximum network rate [142].

Listing 2.3: Network conditioning using a test

```
1
2           /**
```

```
3       Unit tests for ensuring network connectivity. Applicable to all
            versions, inclusion is subject to the test configuration.
4       */
5       @EAT({"modules/testcases/jdkAll/Android/junitbasicsample/testing-
            samples/unit/BasicSample/app/src/test/java#0.0.0"})
6       public class NetworkConditionerTest {
7          ...
8          @Test
9          @Order(1)
10         public void NetworkConditionerTestRet() {
11            final WifiNetworkSuggestion wiFiConnection =
12            new WifiNetworkSuggestion.Builder()
13            .setSsid("testnetwork")
14            .build();
15            final WifiManager wifiManager = (WifiManager) context.
                  getSystemService(Context.WIFI_SERVICE);
16            final int status = wifiManager.addNetworkSuggestions(
                  ArrayList(wiFiConnection));
17            if (status != WifiManager.STATUS_NETWORK_SUGGESTIONS_SUCCESS
                  ) {
18               System.exit(1);
19            }
20         }
21
22      }
```

2.4.4 Multi-device mobile application testing

Mobile applications may be available on a multitude of devices, with quite diverse characteristics, both in terms of hardware equipment (processor type, screen size, available memory, network devices, sensors such as accelerometer or gyroscope etc.) and software (different operating systems or operating system versions). The variations in hardware and software necessitate corresponding differentiations on the tests that need to be conducted for each case, and possibly the code and libraries that need to be built into the respective executable, as well as the build path that needs to be applied.

The Mobile AT Framework can be used to manage the testing of the application on different devices (device Fragmentation). Initially the specifications present in test configuration are extracted to determine the different device types on which the mobile application needs to be tested. For each such device, the pertinent code and tests are extracted from the repositories, utilizing the build path instructions in repositories and the declarative annotations in the tests, which designate the applicability of tests for the specific device configuration; once these elements are available, the executable is built. Subsequently, the target test platforms are configured, the executable is deployed on the mobile device image and finally tests are executed and results are collected.

Fig. 2.5 illustrates the testing process when multiple device types are specified.

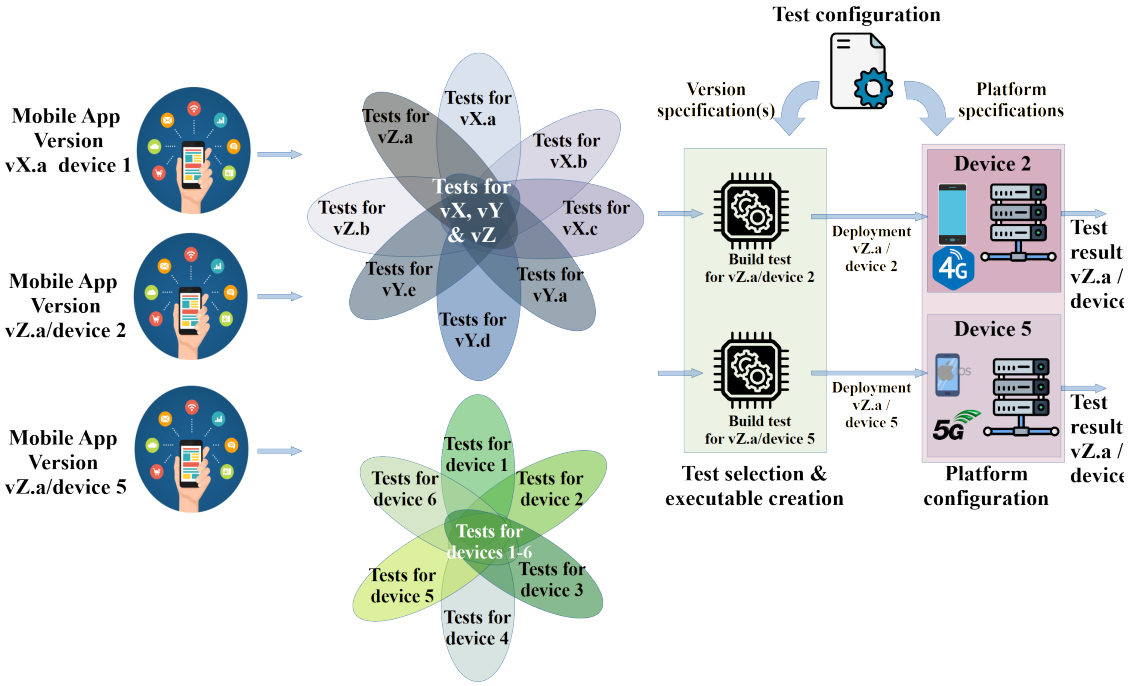

Figure 2.5: Multi-device mobile application testing using the ATF/MA Framework.

The ATF/MA approach to multi-device application testing is illustrated in Listing 2.4. The testing procedure targets two devices, the first of which runs Android 9 (API level 28), while the second one runs Android 11 (API level 30). The main activity *MainActivity.java* of the application is different for each device, since on the Android 11 device it imports *android.telephony.emergency.EmergencyNumber* which has been introduced at API level 29, while it is unavailable for previous API versions.

The test category of the ATF/MA repository is *the jkslinks*[9] and the source differences of *version 1* and *version 2* can be found at sources of the modules layer [10].

Listing 2.4: Source differences in file MainActivity.java of a mobile application example

```
1         /* VERSION 1 */
2
```

[9] https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/testcases/jdkAll/Android/jkslinks

[10] https://github.com/EAT-JBCOMMUNITY/EAT/blob/master/modules/src/main/java/org/example/jkslinks/version1/MainActivity.java and https://github.com/EAT-JBCOMMUNITY/EAT/blob/master/modules/src/main/java/org/example/jkslinks/version2/MainActivity.java

```
3          package com.fari.jkslinks;
4
5          import android.content.Intent;
6          import android.net.Uri;
7          import androidx.appcompat.app.AppCompatActivity;
8          import android.os.Bundle;
9          import android.view.View;
10
11         //@EAT({"modules/testcases/jdkAll/Android/jkslinks/JKSLinks/app/
               src/main/java#2.8.0*2.8.1"})
12         public class MainActivity extends AppCompatActivity {
13
14
15            /* VERSION 2 : import android.telephony.emergency.
                  EmergencyNumber; is added */
16
17            import android.content.Intent;
18            import android.net.Uri;
19            import androidx.appcompat.app.AppCompatActivity;
20            import android.os.Bundle;
21            import android.view.View;
22            import android.telephony.emergency.EmergencyNumber;
23
24            //@EAT({"modules/testcases/jdkAll/Android/jkslinks/JKSLinks/app
                  /src/main/java#2.9.0"})
25            public class MainActivity extends AppCompatActivity {
```

2.4.5 Using ATF/MA for feature-based mobile application development

Feature-based mobile application development, also referred to as *Feature-First Architecture*, is a relatively new trend in mobile application development which is gaining acceptance, since it underpins scalability and maintainability, improves code organization, while it additionally streamlines development and promotes team collaboration [143]. In feature-based mobile application development, features constitute units of functionalities; each feature encompasses the relevant model (data), the pertinent business logic and the user interface elements needed for the user interaction. Each feature is typically mapped to a distinct module, potentially encompassing feature-specific widgets, while dependency injection is utilised to promote composability without sacrificing loose coupling [143]. Once features have been developed, executables (both testing and production ones) can be assembled by synthesizing sets of features; different executables may be produced in this process, where some executables may correspond to versions of the application with different number of features (e.g. free, professional and enterprise versions), or versions of the application corresponding to different actors in some business process, sharing portions of the model, the business logic and the user interface (e.g. customers and suppliers in order processing sharing retrieval and display of orders).

ATF/MA can be used to facilitate testing in the context of feature-based mobile application development. ATF/MA provides a specific annotation namely `ATFeature`, which can be used to designate that the annotated test is pertinent to builds incorporating the designated feature(s). Therefore, when a build encompassing a specific set of features is built, the pertinent tests are retrieved from the test repositories and bundled into the executable, which is then deployed on the platform and run, in order to produce the test results. It is worth noting that some features may necessitate support by hardware and/or software modules of the target platform (e.g. NFC hardware components of specific operating system API levels), hence, in order to test executables with different feature sets, the ATF/MA may need to configure different test platforms.

Fig. 2.6 illustrates the application of ATF/MA for feature-based testing. Note that feature `F3.v2` is assumed to require a newer version of the operating system (v12 instead of v10 needed by the previous feature version, F3.v1); this aspect is catered for by the ATF/MA framework and reflected in the configured platforms.

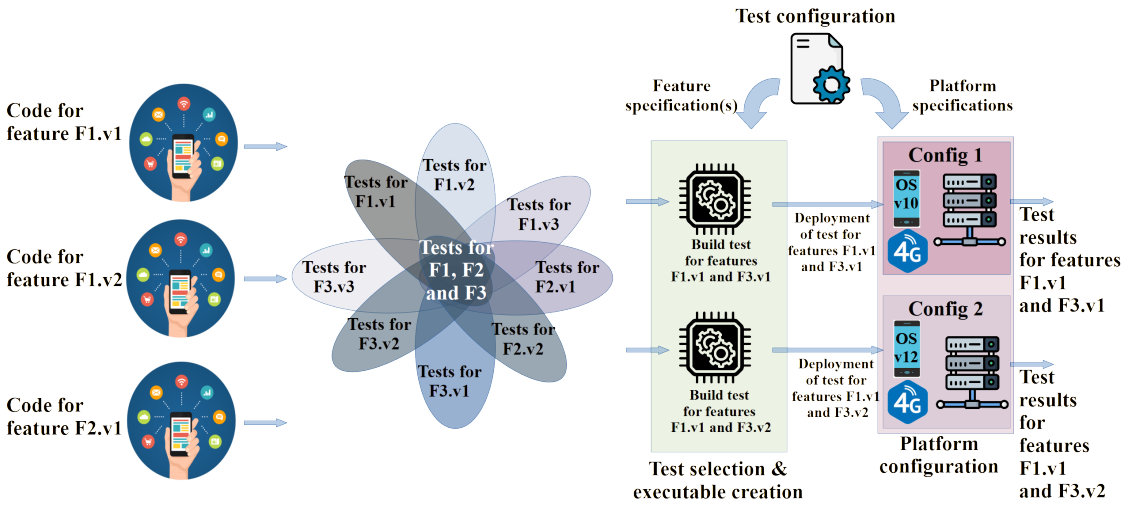

Figure 2.6: Dynamic multifeatured mobile application testing using the AT Framework.

For mobile applications involving multiple or alternate features, the ATF/MA provides the `@ATFeature` annotation, in order to enable or disable the tests related to a feature, depending on whether the specific feature is included in the application build currently considered. The features included in the current application build and the versions of the features are specified to the ATF/MA through the use of the `FEATURE_LIST` environment variable. For instance the annotation in Listing 2.5

Listing 2.5: Example use of the `@ATfeature` annotation

```
1 //@ATFeature(feature={"addFirstParamTwice"},minVersion={"1"},
      maxVersion={"null"})
```

designates that the test is specific test is applicable when the `addFirstParamTwice` feature is enabled, and the respective version is 1 or higher. The respective setting for the `FEATURE_LIST` variable to designate that the `addFirstParamTwice` is included

in the current build would be

```
export FEATURE_LIST="addFirstParamTwice,2"
```

The ATF/MA framework allows for specifications of tests that are executed only when a combination of features is included in the current build. This is accomplished by specifying all pertinent features in the `@ATFeature` annotation, as shown in Listing 2.6.

Listing 2.6: Example use of the `@ATfeature` annotation with multiple features

```
1 //@ATFeature(feature={"feature1,feature2,feature3"},
    minVersion={"1.2.3,1,1"},maxVersion={"2.3.4,12,null"})
```

The annotation in Listing 2.6 specifies that the relevant test should be executed only when *all features* `feature1`, `feature2` and `feature3` are included in the build, and the respective versions are from 1.2.3 to 2.3.4 for `feature1`; 1 to 12 for `feature2`; and 1 and upwards for `feature3`.

An example of feature-based testing is provided in the repository *the testorchestratorwithtestcoveragesample*[11]; the source and test differences of *version 2* can be found at sources of the modules layer [12], while the FEATURE_LIST used in this example is `export FEATURE_LIST="addFirstParamTwice,2"`.

2.4.6 Dynamic mobile applications testing

While the behaviour of mobile applications is –to a large extent–determined by the installed build, mobile apps may need to dynamically adjust their operation to respond to environmental changes. For instance, network connection may switch from cellular internet to wireless LAN or vice versa, the device may switch from normal operation to power save mode, or the display device may change from the smartphone/tablet screen to an external display. Changes in the mode of operation may also be owing to business logic aspects: for example, when the network connectivity switches from cellular internet to corporate wireless internet, an enterprise mobile application may enable an additional set of features, which were previously unavailable for security purposes (the ability to connect establish an authenticated connection to the corporate network is considered as an additional security guarantee).

The ATF/MA framework supports the testing of applications that exhibit dynamic behaviour. In these cases, the test configuration should specify the use of all configurations (cf. Section 2.4.2), environment parameters (cf. Section 2.4.3) and features (cf. Section 2.4.5) that are bound to be activated during the execution of the application. Additionally, the testing protocol should accommodate provisions for the conditions that trigger switching between different modes; for instance the testing

[11] `https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/testcases/jdkAll/Android/testorchestratorwithtestcoveragesample`

[12] `https://github.com/EAT-JBCOMMUNITY/EAT/blob/master/modules/src/main/java/org/example/runner/version2/Calculator.java#L33`, `https://github.com/EAT-JBCOMMUNITY/EAT/blob/master/modules/src/main/java/org/example/runner/version2/CalculatorInstrumentationTest.java` and `https://github.com/EAT-JBCOMMUNITY/EAT/blob/master/modules/src/main/java/org/example/runner/version2/CalculatorAddParameterizedTest.java`

code may include invocations to power management system calls [144] to emulate the behaviour of power saving, as shown in Listing 2.7. The tests included in the test configuration of app `com.company.mobile.testApplication` should include invocations to suitable operations, and the results for the first five seconds (a tunable duration) relate to the behaviour of the application under normal power, whereas results for later timepoints relate to the behaviour of the application under low power. Similarly, the ADB WiFi Setting Manager [145] can be used to switch between network connections, and so forth.

Listing 2.7: Testing under different power conditions

```
1
2               REM This is run on a Windows machine connected to the test
                    device (physical or VM)
3               REM Start the test application
4               adb shell am start -n com.company.mobile/com.company.mobile.
                    testApplication
5               REM wait for five seconds
6               timeout /t 5 /nobreak >nul
7               REM put the mobile device in low power mode
8               adb shell settings put global low_power 1
```

2.5 Evaluation

In this section we evaluate the utility of the ATF/MA framework for the effective management of tests. In order to quantify the benefits of the ATF/MA approach and gain deeper insight on the developer experience from its use, a number of mobile application projects were developed or refactored as listed in Table 2.1; for each project, references to the GitHub locations for the source code and the tests are given.

Table 2.1: List of mobile application projects used for the evaluation

Project	Source code	Test code
Basic mobile application with e-mail address validation checks	[146]	[147]
Multi-configuration mobile application	[148]	[149]
Multi-environment mobile application (adapted from [150])	[151]	[152]
Multi-feature mobile application (adapted from [153])	[154]	[155]
Mobile media player (adapted from [156])	[157]	[158]

Utilizing these projects, we applied common software engineering activities, recording the tasks that need to be carried out for test code and test configuration maintenance for each activity (a) without the use of the ATF/MA framework and (b) when employing the ATF/MA framework. Some tasks need to be performed in both cases, i.e. either when the ATF/MA framework is employed or not; in order to promote readability and facilitate the comparison of the approaches, these are

factored out and listed under column "Common tasks". The results of our experiments are listed in Table 2.2.

Our experiments concur the findings of [131] regarding the gains reaped regarding the use of the ATF/MA framework for test management for multi-version, multi-configuration, feature-based and dynamically built applications. In the context of mobile app development, these benefits are further amplified, due to the existence of additional complexity factors, such as device fragmentation, changing mobile ecosystem and resource scarcity in mobile devices. In more detail:

- When using the ATF/MA framework, only a single test code base needs to be maintained. Without the use of the ATF/MA framework, a separate test code base needs to be maintained for each version; if versioning is combined with feature-based builds, the number of test code bases may scale up to $(\#versions) * (\#features!)$, if all possible feature combinations are pertinent, and each distinct feature combination is applicable to all versions of the app.
- When using the ATF/MA framework, tests are associated with the environment/device setups they pertain to through declarative specifications. Without the use of the ATF/MA framework, test associations with environment/device setups must be performed separately for each environment/device setup.
- When using the ATF/MA framework, introduction of new tests and modification of existing ones necessitate considerably less workload.

Table 2.2: Tasks needed for test code and test configuration maintenance for common software engineering activities (a) without the use of the ATF/MA framework and (b) when employing the ATF/MA framework

Project activity	Common tasks	With ATF/MA	Without ATF/MA
Creation of a new version	-	For tests whose annotations do not designate explicitly the last version on which they are applicable, no task needs to be performed; for tests designating explicitly the last applicable version, the new version needs to be added to the annotation.	A separate code base of the project, corresponding to the new version, needs to be added. Test that cease to be valid for the new version need to be removed from the new code base.
Addition of a configuration	A new configuration file needs to be created, specifying the properties of the new configuration. Tests that are particular for the new configuration need to be created.	Existing tests need to be executed differently under the new configuration need to be modified. If new tests are created for the particular configuration, for each of them an annotation, declaratively specifying all the configurations, versions and environments on which the test is applicable.	All tests to be executed under the new configuration should be explicitly listed. If new tests are created for the particular configuration, each test needs to be manually added to all version code bases, configuration and environment setups where each of them is applicable [159].

Table 2.2: Tasks needed for test code and test configuration maintenance for common software engineering activities (a) without the use of the ATF/MA framework and (b) when employing the ATF/MA framework

Project activity	Common tasks	With ATF/MA	Without ATF/MA
Consideration of additional environments	A new configuration file needs to be created, specifying the new environment setup. Tests that are particular for the new environment need to be created.	Existing tests need to be executed differently under the new environment need to be modified. If new tests are created for the particular configuration, for each of them an annotation, declaratively specifying all the configurations, versions and environments on which the test is applicable.	All tests to be executed under the new environment should be explicitly listed. If new tests are created for the particular environment, each test needs to be manually added to all version code bases, configuration and environment setups where each of them is applicable [159].
Addition of a new device type	A new configuration file needs to be created, specifying the setup for the new device type . Tests that are particular for the new device type need to be created.	If new tests are created for the particular device type, for each of them an annotation, declaratively specifying all the configurations, versions and environments on which the test is applicable.	All tests to be executed for the new device type should be explicitly listed in the new configuration. If new tests are created for the particular environment, each test needs to be manually added to all version code bases, configuration and environment setups where each of them is applicable [159].

Table 2.2: Tasks needed for test code and test configuration maintenance for common software engineering activities (a) without the use of the ATF/MA framework and (b) when employing the ATF/MA framework

Project activity	Common tasks	With ATF/MA	Without ATF/MA
Addition of new features	Tests associated with the new feature need to be developed.	An `ATFeature` annotation designating that the test should be executed when the newly introduced feature is included in the distribution is added to each new test.	A separate test code base is introduced for each build involving the new feature (multiple builds may be required, if the new feature is combined with other features). Newly developed tests pertinent to the new feature are added to each of these test code bases; other tests are added to these test code bases as needed.
Testing dynamic applications	Tests associated with the dynamic features of the applications need to be developed.	Dynamic features pertain to the adaptation of the application's behavior to changing environment, device and configuration characteristics as well as dynamic feature enablement/disablement. Consequently, the relevant tasks listed above for configurations, environments, devices and features apply for dynamic application testing.	
Introduction of a new test	The new test needs to be developed.	An annotation, declaratively specifying all the configurations, environments and devices on which the test is applicable needs to be added to the test.	The test needs to be added manually to each project code base and configuration.
Modification of an existing test	The test code needs to be updated.	None, since only one instance of the test exists.	Modifications to the test need to be propagated in all test code bases where it is present.

2.6 Conclusions

In this chapter we introduced the ATF/MA framework, which can be used to support the testing of mobile applications, meeting the real-world requirements for testing multi-version, multi-configuration, multi-device apps, and furthermore underpinning the process of testing feature-based builds and mobile apps exhibiting dynamic behavior. The ATF/MA framework extends the capabilities of the Additional Testsuite Framework [131] considering the particularities of the mobile application domain, effectively tackles the issues related with device fragmentation and change of the mobile ecosystem and reduces the complexity of test management and the workload of developers and testers.

The ATF/MA framework provides the necessary instrumentation and procedures to assist testers in setting up, developing and maintaining the test code base. In utilizes an annotation-based approach to enable a declarative specification of the association between individual tests and the pertinent builds or environment and deployment configurations. The effectiveness of the ATF/MA framework has been demonstrated through its application on a number of test cases. Similarly to the Additional Testsuite Framework [131], ATF/MA is applicable on any app written in any programming language, including both native languages such as Java and Objective-C, as well as cross-platform languages such as Flutter [138]. Moreover, the ATF/MA framework may serve all types of tests, including unit testing, integration testing, functional testing, performance testing etc.

In our future work we will explore the incorporation of model-based testing methods [87] and techniques for intermittent fault detection [88]; this is particularly important for mobile apps, since conditions which trigger the manifestation of intermittent faults may be correlated with aspects of the mobile app environment, such as change of network connectivity or switching between power consumption modes. Taking into account that for mobile apps, the number of tests that need to be conducted to ensure seamless operation is higher than traditional applications, due to the complexities of the mobile ecosystem and environment, the incorporation of test case prioritization [89] will be considered. Finally, in order to assist testers for identifying tests that are executed for each particular build and validating test annotations, a relevant tool will be implemented.

Chapter 3

A Software Vulnerability Management Framework for the Minimization of System Attack Surface and Risk

Abstract

Current Internet of Things (IoT) systems comprise multiple software systems that are deployed to provide users with the required functionalities. System architects create system blueprints and draw specifications for the software artefacts that are needed; subsequently, either custom-made software is developed according to these specifications and/or ready-made COTS/open source software may be identified and customized to realize the overall system goals. All deployed software however may entail vulnerabilities, either due to insecure coding practices or owing to misconfigurations and unexpected interactions. Moreover, software artefacts may implement a much broader set of functionalities than may be strictly necessary for the system at hand, in order to serve a wider range of needs, and failure to appropriately configure the deployed software to include only the required modules results in the further increase of the system attack surface and the associated risk. In this chapter, we present a software vulnerability management framework which facilitates (a) the configuration of software to include only the necessary features, (b) the execution of security-related tests and the compilation of platform-wide software vulnerability lists, and (c) the prioritization of vulnerability addressing, considering the impact of each vulnerability, the associated technical debt for its remediation, and the available security budget. The proposed framework can be used as an aid in IoT platform implementation by software architects, developers, and security experts.

3.1 Introduction

The Internet of Things (IoT) concept involves devices with Internet connectivity, which can exchange data, information, and services, obtain data from their environment through sensors, and initiate changes to it through actuators. The building

blocks for IoT systems exhibit considerable diversity, ranging from specialized industrial or enterprise products, such as production line robots [160, 161], smart grid devices including smart meters [162], connected and autonomous cars [163, 164], or consumer products including wrist bands and smart watches, smart air conditioners, and smart TVs. IoT systems are expected to proliferate over the next years: Statista projects that the number of connected IoT devices will double from 2023 to 2030 [165], while IHS Markit predicts that the number of connected IoT devices will exhibit an annual increase of 12%, escalating to 125 billion devices in 2030 [166].

A critical factor for the operation of existing IoT systems and the proliferation of new ones is security. Multiple studies e.g., refs. [167, 168, 169, 170] have identified a number of key security areas, spanning across physical aspects, the network layer, the edge layer and the application layer. These include physical device protection, architectural concerns, authentication, encryption, trust, secure routing protocols, privacy concerns, and so forth. A major factor affecting the security of IoT systems, at virtually any layer, is the software. All deployed software (either custom-made to serve the requirements of the particular system, or ready-made "Commercial off-the-Shelf (COTS)"/open-source software that has potentially been customized) may entail vulnerabilities, which may be exploited by attackers. Characteristically, the study reported in [171] reports that after security tests were conducted on thirteen routers and network attached storage (NAS) devices for small office/home office (SOHO) environments, a number of vulnerabilities was discovered in each of them, totalling 125 common vulnerabilities and exposures (CVEs). Ref. [171] also reports that manufacturers were notified about the vulnerabilities, however the number of responses collected was very limited, and very few of the vulnerabilities were actually addressed.

The increased number of vulnerabilities in IoT devices, coupled with the large number of these devices and their often unrestricted accessibility through the internet, has led to some world-scale security incidents affecting millions of devices, including Hajime [172], BASHLITE [173], and Log4Shell [174].

Software-rooted vulnerabilities occur either due to insecure coding practices due to misconfigurations and unexpected interactions (e.g., race conditions). Moreover, deployed software artefacts may be poorly customized, and offer a broader set of functionalities than may be strictly necessary for the system at hand: for instance, if the OpenLiberty server [175] is deployed at some system, the Java Message Service 2.0 [176] feature may be left at an enabled state, although it is not needed in the particular installation. In such a case, the attack surface of the system increases (all APIs/endpoints of the Java Message Service 2.0 can be used by attackers), and any vulnerabilities present in the code realising the Java Message Service can be targeted for exploitation.

Addressing software-rooted vulnerabilities in an IoT system is thus a complex, multi-faceted issue, involving (a) the minimization of the vulnerabilities present in the code, through the inclusion of only the necessary features, (b) the identification of the remaining vulnerabilities, and (c) the fixing of these vulnerabilities. However, fixing the vulnerabilities incurs a cost—in both time and human resources—and the time available or the budget allocated to this task may be limited, not permitting the tackling of all issues. In such cases, the effort of the developer team should

be directed to fixing the errors that would minimize the overall *residual risk*, i.e., the risk owing to the vulnerabilities that will not be fixed, due to security budget constrains.

To the best of our knowledge, no commercial system or research proposal offers prioritization of software components' vulnerability addressing the associated technical debt for its remediation and the available security budget (considering the impact of each vulnerability); developers are not adequately supported to prioritize fixes and deploy platforms with minimized residual risk.In this chapter, we present a software vulnerability management framework which supports all the stages of a pipeline for the management of IoT platform software vulnerabilities, i.e., (a) the configuration of software to include only the necessary features (b) the execution of security-related tests and the compilation of platform-wide software vulnerability lists, (c) the estimation of the impact and the associated fixing cost for each vulnerability, and (d) the prioritization of vulnerability addressing the associated technical debt for its remediation and the available security budget, considering the impact of each vulnerability. The proposed framework can be used as an aid in IoT platform implementation, by software architects, developers, and security experts.

The work presented in this chapter advances the state-of-the-art by (a) proposing a statistics-based method for the estimation of impact of detected vulnerabilities, (b) proposing an integer programming-based algorithm for prioritizing security fixes with the goal of minimizing the residual risk level, and (c) harnessing the power of a test management framework [131] and static code analysis [177], and combining them with the estimation of impact of detected vulnerabilities and the integer programming-based prioritization algorithm to synthesize a comprehensive framework for the security analysis of platform software which formulates proposals on the prioritization of addressing security issues, taking into account the software components to be included and their features, the impact of each vulnerability within the source code, the associated technical debt for its remediation, and the overall available security budget.

In the remainder of this section we initially present the landscape of software-rooted security issues in IoT systems, as reported in papers where the authors exploited (a) security testing tools, (b) lists of known vulnerabilities, and (c) catalogues of common weaknesses found in IoT software (Section 3.1.1). Note that these works aim to identify the issues that may be present in the software without providing tools for assessing their impact on specific configurations or platform deployments and without providing any means for prioritizing fixing issues. Subsequently, we present the state-of-the-art in the security assessment of IoT software code (Section 3.1.2). In this subsection, we focus on static code analysis techniques [178], which are employed by the proposed framework. We also briefly cover other techniques, including dynamic testing and fuzzing, to provide a more complete picture of the software security assessment methods.

In Section 3.2 we present the proposed method for the management of vulnerabilities in IoT platform software, while in Section 3.3 we demonstrate the application of the framework in IoT platform software. Finally, in Section 3.4 the practical and theoretical implications of the proposed framework are discussed, and future work is outlined.

3.1.1 Software-Related Security Issues in IoT Software

According to OWASP [179], a large percentage of the vulnerabilities present in applications can be linked to insecure coding practices followed by software developers. Even though extensive research has been published on the analysis of IoT vulnerabilities, it is focused mainly on black-box methods such as penetration testing and fuzzing [sachidanada][samtani][geneiatakis][overstreet]. There is limited research available on the white-box testing of open-source software for IoT devices, which involves examining the source code to identify vulnerabilities. Source code reviews could be significantly beneficial for vulnerability research in the IoT field, as they often detect different vulnerabilities than those revealed through black-box techniques. Additionally, the detailed documentation of vulnerabilities in open-source software constitutes useful information for preventing similar coding mistakes from occurring in the future [180].

In Schiller et al. [181], the authors discuss the landscape of IoT security by first providing some background information on IoT in general, on the concept of security, IoT networking, and the available IoT architectures. Furthermore, they discuss challenges in achieving IoT security and propose a threat taxonomy. The authors composed the taxonomy by reviewing the available research and by integrating the threats and attack methods identified. The classification system used organizes the threats according to a three-level architecture model of the IoT which includes the sensing, network, and application layers. In the threats against IoT classified under the application layer, we can find several that can be traced back to insecure coding practices, some on a higher level and some on a lower level: data modification, elevation of privilege, DoS, password change, password guessing, buffer overflow, memory corruption, code execution, SQL injection, XSS, and CSRF.

In Calatayud et al. [182], the authors utilize the Raspberry Pi hardware platform as a base to test operating systems used in high-end IoT devices against a multitude of buffer overflow attack forms. The attacks were carried out using the RIPE tool [wilander]modified accordingly for the ARM architecture. The operating systems were tested with and without buffer overflow protections in place. The outcomes offer valuable information about common buffer overflow vulnerabilities found in IoT operating systems along with a persistent pattern of the preventive measures that are used to protect against this type of attack. The target operating systems are RPi OS, Xubuntu, Alpine Linux, Arch Linux ARM, and Chromium OS. RPi OS v10 was found vulnerable to 12 attacks with the vulnerable functions being memcpy and homebrew. Xubuntu v18.04 and Alpine Linux v3.14 were found to be vulnerable to 50 attacks with the vulnerable functions being memcpy, strcpy, sprint, and fscan. Arch Linux ARM v5.10 was vulnerable to 10 attacks with vulnerable functions being the same ones as with RPi OS. Lastly, in Chromium OS v5.4, 59 attacks were successful with the set of vulnerable functions being the same as with Xubuntu and Alpine Linux.

In Al-Boghdady et al. [177], the authors investigate the security posture and the vulnerabilities present in the source code of four popular IoT operating systems through static analysis. They utilize three static analysis tools—Cppcheck, Flawfinder, and RATS—to analyze sixteen different versions of the four C/C++ IoT OSs with the research goals being the identification of vulnerabilities from the common weakness

enumeration (CWE) scheme and to find out if the security errors and their density (errors per 1K source lines of code) increase or decrease over time. Furthermore, the last research question posed by the authors asks what the relationship between the vulnerabilities of the IoT OSs and their evolutionary properties is. The tool CodeScene is utilized to that end. The results showed that while the total number of security errors increases with each version, the error density decreases over time for all examined operating systems, with few exceptions. The most prevalent vulnerabilities in the OSs examined for Cppcheck were CWE-561 (dead code), CWE-398 (7PK—code quality), and CWE-563 (assignment to variable without use), while CWE-119 (improper restriction of operations within the bounds of a memory buffer), CWE-120 (buffer copy without checking zize of input ("classic buffer overflow')), and CWE-126 (buffer over-read) for Flawfinder. For RATS, it was CWE-119, CWE-120, and CWE-134 (use of externally-controlled format string).

Mathas et al. [180] evaluate the vulnerabilities in IoT software used mainly in smart grid applications through static analysis of the source code. The authors have analyzed open-source software which could be used at any level of the software stack, including operating system level, application level, and library level, in order to obtain a comprehensive understanding of the relevant vulnerability landscape. The assessed software includes jSML, lib60870, libiec61850, JavaSMQ, Pymodbus, Modbus4j, Minnow server, Boa Webserver, thttpd, MicroWebSrv 2, and Busybox. The static application security testing (SAST) is performed by utilizing the SonarCloud[13] and Codacy[14], platforms. The results of the two platforms are manually reviewed to discern between true and false positives. The final results are categorized based on a customized vulnerability categorization scheme which was created by combining the OWASP Top 10 list and the MITRE common weakness enumeration (CWE) scheme. Based on the results received from the static analysis, the custom categorization scheme includes improper certificate validation, buffer overflow, weak cryptography, sensitive data exposure, race condition, and broken access Control. Furthermore, the frequency and the potential impact of the identified vulnerabilities is considered. The article provides a detailed examination of true and false positives which can assist both researchers and practitioners to better focus on the areas requiring review. The vulnerabilities results report 2 for the improper certificate validation category, 6 for buffer overflow, 23 for weak cryptography, 59 for sensitive data exposure, 8 for race condition, and 3 for broken access control. Additionally, the results show numerous false positives reported by the SAST tools utilized.

3.1.2 Assessing the Security of IoT Software

Static code analysis [178] is a part of the security development lifecycle [183] and is performed with the code review of static code, which can also be realized running static code analysis tools to detect possible security vulnerabilities by analyzing the code of the tested software. Static analysis can be used with both source and compiled code. While the development flaws detected through static analysis may include non-security-related issues as well, in this work, we focus only on security

[13] `https://sonarcloud.io`
[14] `https://app.codacy.com`

issues. Static analysis with a focus on security issues is usually referred to as static application security testing (SAST). SAST identifies problematic patterns by checking the code statically rather than inspecting it during runtime. Depending on the implementation, SAST can be simple or complex, and can detect patterns in the code, produce control graphs, or analyze data flow logic to identify user input that reaches sensitive code segments [184].

SAST has become an easy and efficient practice that has gained widespread acceptance in recent years. It is available in various forms, including IDE plugins, standalone applications, online services, and solutions integrated into continuous integration/continuous delivery (CI/CD) pipelines [185]. SAST tools are available in both open-source and commercial formats, offering different functionalities that cater to specific needs. Various techniques are employed by static analyzer tools, such as data flow analysis, control flow graphs (CFGs), taint analysis, and lexical analysis [186]. However, even with recent advancements in SAST, tools still have high false-positive rates, necessitating human intervention for results evaluation [184]. One basic limitation of SAST solutions is that they suffer from a high rate of false positives in their results [180]. To that end, we begin this section by presenting one of the latest works towards improving SAST for IoT by utilizing machine learning. SAST is an invaluable way of assessing the security of IoT but not the only one needed as it is meant to be complementary (but necessary) to the more traditional black-box techniques used. Black-box methods used for vulnerability detection in IoT systems include, among others, fuzzing, taint-analysis, symbolic execution, homology analysis, and penetration testing [187, 188, 189, 190, 191]. In the remainder of this section, we discuss some of the latest scientific work conducted on these methods.

Kotenko et al. [192] propose an intelligent framework concept for the static analysis of IoT systems utilizing machine learning techniques. They systematize the fundamental components of static analysis and machine learning areas to form two models: the SA (static analysis) model and the ML (machine learning) application model for SA. SA is broken down into stages and ML into tasks. The models are represented as matrices with rows corresponding to the tasks and columns to the stages.

The SA stages considered are data collection, data preparation, data processing, and result formation (the columns of the SA model matrix). For the assignment of each activity to the stages of SA, they utilize the following formalization: any given data is represented in terms of its content (C), which refers to the information it contains, and its form (F), which pertains to its appearance. Thus, data stored within the IoT system and modified during the SA process can be expressed as a tuple $<F|C>$. Based on this, a formalized description of each stage is defined.

The ML tasks were chosen based on both the general theory and a large number of scientific papers and their reviews. The ML tasks considered are classification, anomaly detection, regression, clustering, and generalization (the rows of the SA model matrix). The tasks are described through formalized definitions as well. The resulting matrix contains in its cells a formalized record of stage actions based on the solution of one of the ML tasks. The authors note that non-ML statements will precede and succeed the ML ones, since apart from the intelligent component, each step comprises strictly defined rules (e.g., unpacking archives with files, ranking

documents by their size, etc.). The matrix's validity was confirmed through expert analysis.

For the second part, an analysis of research papers relevant to analyzing IoT systems is conducted. Each study is categorized and assigned based on its attributes to one or more SA stages and one or more ML tasks. The second matrix has the same types of rows and columns, but the cells are the research papers that were categorized under the corresponding stage-task couple.

The two models make it possible to create methodological solutions that are theoretically and practically sound in order to provide information security in the IoT systems domain. This, of course, necessitates the development of a suitable framework that can ensure the execution of all phases utilizing the wide range of ML methods available for big data and heterogeneous data. The novelty of this work resides in the fact that it takes into consideration the phases of data collection and preparation that precede the code analysis. Thus, this work defers from previous ones in that it covers the entirety of the SA process. Additionally, this review is the first to consider the full range of ML solutions, both theoretically and practically, for each stage of SA. Finally, not only is SA divided into stages, but it is also suggested to represent the actions of these stages in a formalized manner, which involves transforming the form and content of the data being studied in the IoT system.

He et al. [191] propose a homology detection method based on a clonal selection algorithm for detecting vulnerabilities in IoT firmwares. The proposed problem indicates that methods utilizing machine learning for detecting vulnerabilities in IoT firmwares need sample data of firmware vulnerabilities. This sample data is very scarce for some types of vulnerabilities. The authors characterize the results achieved by machine learning algorithms on this matter as ''not ideal" and attribute this to their need for a large set of sample data. To that end, this work proposes a firmware vulnerability homology detection method based on the clonal selection algorithm combined with the simulated annealing algorithm. Unlike preexisting machine learning methods, this method relies solely on the affinity between the objective function and the detector, eliminating the need for extensive sample data sets. In the clonal selection algorithm of biology, the immune system identifies antigens and creates a variety of plasma cells to produce antibodies that are customized based on the specific characteristics of each antigen. Antibodies that have a high affinity to their respective antigens are retained, while those with low affinities are discarded. When applied to the method in question, the optimization problem is the antigen, the feasible solution to the optimization problem is the antibody, and the quality of the feasible solution is represented by affinity. Essentially, the antigen is a vulnerability function, and affinity represents the similarity between the sample function and the original one. The simulated annealing algorithm is a general optimization algorithm. It begins from a high initial temperature and randomly explores the solution space to find the global optimal solution of the objective function. The main idea is to use the probability of a local optimal solution for obtaining the global optimal solution. This work combines the efficient local search capability of simulated annealing with the rapid convergence of the clonal selection algorithm. The proposed method is tested against a neural network algorithm that attempts to tackle the same issue. The evaluating indicators are recall rate and accuracy. The recall rate is the ratio

of the number of actual vulnerabilities in the algorithm's vulnerability set and the total number of vulnerabilities that should be detected. The accuracy indicator is the ratio of the number of actual vulnerabilities in the algorithm's vulnerability set and the total number of vulnerabilities predicted by the algorithm. The proposed method was found to be faster than its counterpart, as well as having a significantly higher recall rate, improving it by about 13%. There is a slight improvement in accuracy as well.

In Akhilesh et al. [193], the authors describe an automated penetration testing framework for smart home IoT devices. The goal and novelty of the work lies in the full automation of the framework and the ease of use by both technical and non-technical users so that anyone can assess the security of the devices deployed in her/his smart home network. The authors begin by reviewing and comparing the relevant research discussing differences and limitations in order to choose the most suitable work to adopt. The method selected [luis costa]is based on the penetration testing execution standard (PTES). The framework is designed to detect the five most common vulnerabilities for smart home IoT devices. The five most common vulnerabilities were chosen by the authors based on the OWASP IoT Top 10, the work they adopted [luis costa], and the OWASP Top 10. The resulting list includes insecure web interface, remote access vulnerability (improper authentication), insecure network services, lack of transport encryption, and insecure Firmware/Software. The framework can be divided into five consecutive parts: reconnaissance, check for remote access vulnerability, check for insecure web interface, automated traffic capture, and vulnerability detection through traffic analysis. The framework is written as a Python program which executes a combination of tools in a specific order at the user's discretion. Each vulnerability is assigned a corresponding method and one or more tools used to detect it. The tools utilized by the framework are Net Discover, Nmap, OWASP ZAP (zap-cli), Medusa, WhatWeb, Wireshark (t-shark and pyshark), Binwalk, and Firmwalker. The framework was executed in a home network with the following devices connected to it: Tp-Link SmartPlug, Tp-Link Smart bulb, Tp-Link Smart Camera, Google Home Mini, and LIFX Smart Bulb. For the Tp-Link SmartPlug, a probable network services vulnerability was reported. Lack of transport encryption and insecure firmware vulnerabilities were reported for the Tp-Link Smart Bulb and Smart Camera, while no vulnerabilities were reported for Google Home Mini and LIFX Smart Bulb. The execution time varied from 1 to 8 s for each device scan. Additionally, the authors calculated the CVSS scores of the detected vulnerabilities. The base scores were summarized to form a total score for each device. The Tp-Link Smart Bulb and Smart Camera were found to be the most vulnerable. Google Home Mini had the same score as LIFX Smart Bulb (zero), but further analysis conducted by the authors showed that Google Home Mini employs more secure mechanisms than its counterpart and is therefore the most secure.

Zheng et al. [194] propose a novel approach to greybox fuzzing for Linux-based IoT devices. Greybox fuzzing is a very effective vulnerability discovery technique but when applied to IoT devices it faces various limitations. The basic limitation occurs from the IoT application's high reliance on specific system environments and hardware. Some techniques use full-system emulation to bypass that limitation, but this technique has a high overhead. To that end, some works, such as Firm-AFL,

propose to combine full-system emulation with user-mode emulation in order to provide full compatibility in user emulation. Firm-AFL executes the application in user-emulation mode until a system call is needed to continue the execution. When that occurs, the emulation shifts to full-system emulation to execute the system call. This approach proved to be less efficient than full-system emulation, especially in the cases of applications that make frequent system calls. To that end, the authors propose EQUAFL, a greybox fuzzing framework with enhanced user-mode emulation. EQUAFL first executes the application in a full-system emulation and observes the key points, such as the setting of launch variables, the generation of configuration files, and network setup, etc. Next, EQUAFL sets up the execution environment for the application by replaying the observed behaviors. This called an observe–replay strategy by the authors. The framework's performance is evaluated on compatibility, efficiency, and vulnerability discovery. Two different datasets were used as benchmarks. The first dataset comprises two standard benchmarks and the other dataset comprises 70 embedded firmware images from D-Link, TRENDnet, and NETGEAR. The first dataset is small and does not contain bugs, so it is used for the evaluation of efficiency and compatibility. The second dataset is used for vulnerability discovery as well. The baselines used are simple user-mode emulation, full-system emulation, and Firm-AFL. Additionally, experiments are conducted 5 times each to mitigate the randomness of the fuzzers. The compatibility results show the successful execution of the first dataset and 66 out of 70 applications in the second (real world) dataset. EQUAFL's results are much better than simple user emulation and comparable to full-system emulation in terms of compatibility. In terms of efficiency EQUAFL was found to be 26 times faster than full-system emulation and 14 times faster than Firm-AFL in real-world applications. The overhead of EQUAFL on the benchmark data was marginal compared to user-mode emulation. It discovered 10 vulnerabilities, for 6 of which CVEs were assigned after reporting them to the corresponding vendors. Finally, EQUAFL was shown to detect vulnerabilities much faster than its counterparts.

3.2 Materials and Methods

In this section we briefly present the proposed framework that is described in the previous chapters. Firstly, in Section 3.2.1 we describe the overall framework architecture, while in the remaining subsections we provide details on the operation of each architectural component.

3.2.1 Architecture of the Proposed Framework

The overall architecture of the proposed framework is illustrated in Figure 3.1. The processing pipeline begins by gathering the software components that will be deployed on the IoT platform. The source of each such software component is stored in the corresponding code repository. This step is instrumented by the additional testsuite framework [131] and detailed in Section 3.2.2.

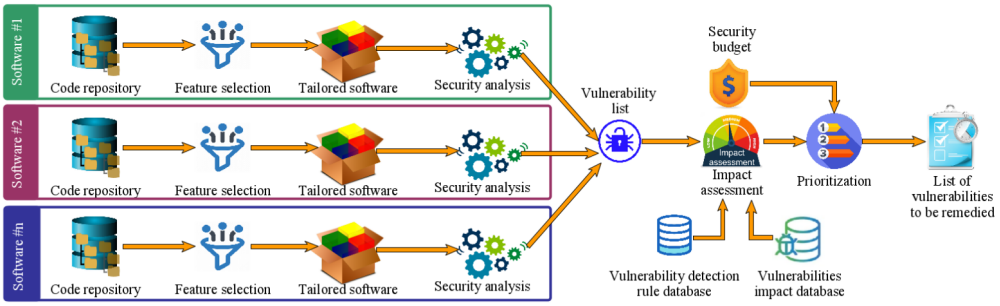

Figure 3.1: Proposed framework architecture.

Subsequently, feature selection is applied to each software component, producing the respective tailored software. Recall from the introduction that feature selection is a step that assists in limiting the attack surface and the associated risk, while it can also lead to configurations with smaller memory footprint and resource requirements. The outcome of the feature-selection step is a set of software components that are tailored to the needs of the specific IoT platform deployment. In order to perform feature-based tailoring of software components, the proposed framework utilizes the relevant capabilities of the additional testsuite framework [131]; the additional testsuite framework is summarized in Section 3.2.2, while Section 3.2.3 provides details on the feature selection capabilities of the additional testsuite framework.

Afterwards, each software component is analyzed to identify security issues, forming a list of software-related vulnerabilities related to the individual software component. In this context, the additional testsuite framework submits the relevant projects to the SonarQube static code-analysis platform, and for each project, it gathers the results through the SonarQube API[15]. Project-specific lists are then merged to formulate a comprehensive vulnerability list for the whole of the platform. Security analysis is presented in Section 3.1.2.

Each vulnerability in the comprehensive platform-wide list is assessed to estimate its impact on the platform. To this end, a statistical approach is used. Vulnerability impact estimation is discussed in Section 3.2.5.

Finally, the vulnerabilities are prioritized, taking into account the impact of each vulnerability and the cost to fix it, as well as the overall security budget, producing the final list of vulnerabilities to be remedied. This is accomplished through an integer programming-based optimization scheme, which is detailed in Section 3.2.6.

This list can guide developers to the process of maintaining the software so as to minimize the overall *residual risk*, i.e., the risk owing to the vulnerabilities that will not be fixed, due to security budget constrains.

3.2.2 The Additional Testsuite Framework: An Overview

The additional testsuite framework (ATF) [131] is novel approach for the management of code testsuites, providing relevant structures and instrumentation. ATF

[15] `https://sonarqube.inria.fr/sonarqube/web_api/api/`

supports a multitude of features, including management of tests for multi-version applications, test-driven development, dynamic/selective program builds, feature-based builds, testing in different environments, and source code analysis. ATF utilizes annotations to associate tests with specific application characteristics, and dynamically matches these characteristics against build specifications and/or deployment environment attributes to (a) retrieve from the source code repositories the relevant sources to be used for the specific build, (b) create the executable image of the tailored software component according to the build specifications, and (c) deploy the executable image to the designated deployment environment.

The benefits stemming from the introduction and use of ATF include the following:

1. Tests are written once, and can be flexibly associated with any number of software programs and versions, limiting the effort and complexity needed for the maintenance of test cases.

2. It supports dynamic/selective program builds that include only the portions of the software that match some designated functionality.

3. For software applications that are developed or organized according to the featured-based development paradigm [195], builds can be tailored to create executables that only support a subset of the available features.

4. It can underpin the localization of bugs introduced during software evolution, including regression bugs, through the comparison of code in versions producing erroneous results against the code in versions yielding correct results.

5. It can facilitate documentation compilation, since functionality-oriented and feature-based tests can be included in documentation on the functionality/feature they pertain to, serving as examples of the specific functionality/feature as well as providing examples of usage.

For more information regarding the capabilities and functionality of the ATF, the interested reader is referred to [131]. In the next subsection, we describe the feature management functionality of ATF, which are utilized in the context of the proposed software vulnerability management framework.

3.2.3 Feature Management Using the ATF

A software program can be defined as the unit of code parts that implement a set of features that are intended as the provided functionalities of the software [195]. Each feature of a software system is an optional or incremental unit of functionality, and is associated with relevant code that realizes this functionality [196, 197]. While software programs constantly evolve to accommodate an increasing number of features, specific deployments of these programs in the context of IoT platforms may necessitate and utilize only a limited number of the available features. Configuring the software program so as to only include the code that realizes the actually needed features may contribute to decreasing the attack surface and minimizing the memory footprint.

ATF provides support for the testing of programs that are dynamically tailored to specific needs through the usage of the `FEATURE_LIST` tailoring specification and the `@ATFeature` code annotation. In more detail, when execution tailoring commences:

1. ATF consults the environment variable `FEATURE_LIST`, which includes the path to a feature-tailoring configuration file listing the features that should be enabled in the specific software build; for each feature, the relevant version that should be enabled is also specified, as illustrated in Listing 3.1. Then, ATF arranges so that the tailored software bundle includes the relevant code realizing the specific features, retrieving the respective code from relevant repositories.

2. ATF scans the code for instances of the `@ATFeature` annotation; this annotation is associated to methods and specifies the features that need to be enabled for the method to be included in the final executable, effectively thus providing an advanced conditional compilation mechanism. More specifically, the `@ATFeature` annotation lists the program features that the specific method is dependent on, and during the tailoring procedure the ATF matches these features against the feature tailoring configuration specified via the `FEATURE_LIST` environment variable, and arranges so that the method implementation code is included in the tailored version of the software if all the specified features are enabled via `FEATURE_LIST` and the version of each enabled feature also matches the designated version range. Listing 3.2 presents an example of the usage of the `@ATFeature` annotation.

Listing 3.1: Example `FEATURE_LIST` file contents.

```
jaxrs,2.3.1\\
jaxb,2.4.0\\
jsonp,1.2\\
cdi,2.1\\
localConnector,1.1\\
servlet,4.1.34
```

Listing 3.2: Example usage of the `@ATFeature`.

```
@ATFeature(feature =\{"jaxrs, jaxb, jsonp, cdi, localConnector,
servlet"\},\\
minVersion =\{"2.1, 2.2, 1.1 ,2.0, 1.0, 4.0"\},\\
maxVersion =\{"null, null, null, null, null, null"\})\\
\textbf{public void} doJaxRs () \textbf{throws} Exception \{// Feature-
    dependent code\}
```

3.2.4 Static Code Analysis for Vulnerability Detection

In the context of the proposed software vulnerability management framework, the facilities of the additional testuite framework are used to gather the tailored software components and submit them to static code analyzers. In our current configuration, the SonarQube static code analyzer is employed; in particular, the SonarQube API[16] is used to submit tailored software components for analysis and retrieve the analysis results, which are filtered to contain only vulnerabilities, by setting the `types` REST API parameter to the value `VULNERABILITY`.

The SonarQube analyzer returns for each security issue identified numerous information items which include:

- A textual description of the issue;
- A designation of the estimated severity of the issue, which may be *INFO*, *MINOR*, *MAJOR*, *CRITICAL*, or *BLOCKER*;
- The component (directly identifying the source file) and the range of the code lines where the security issue was found;
- the SonarQube security rule that triggered the vulnerability flagging.
- An estimate of the *technical debt* associated with the vulnerability, i.e., the time needed to modify the code in order to eliminate the security issue.

3.2.5 Vulnerability Impact Estimation

Once the vulnerability management framework has determined the list of vulnerabilities present in the software, the next step is to estimate the impact that each of these vulnerabilities will have on the IoT platform. Recall (from Section 3.2.4)

[16] `https://sonarqube.inria.fr/sonarqube/web_api/api`

that the list of vulnerabilities contains, for each vulnerability, a reference to the SonarQube security rule that triggered the vulnerability flagging. This information is exploited by the vulnerability management framework to compute an estimate of the vulnerability impact, according to the following process:

1. The rule is looked up in the SonarQube rules database[17] and its full record is retrieved. This record includes:

   - *Common weakness enumeration (CWE) identifiers.* CWE identifiers are codes assigned to typical security-related code anti-patterns, i.e., patterns of code that are known to lead to vulnerabilities. For instance the *java/RSPEC-6437* SonarQube security rule[18] is linked to the *CWE-798—use of hard-coded credentials* [19] and the *CWE-259—of hard-coded password weaknesses*[20]. These identifiers are saved and used in the vulnerability impact estimation, as described below.
   - *A detailed description of the vulnerability*, including an explanation of the mechanics of the code anti-pattern, a substantiation of why the anti-pattern leads to vulnerabilities, and recommendations on how the code can be transformed to eliminate the vulnerability. This information is saved, to be presented to software security experts and to assist them in their vulnerability remediation tasks.

2. Subsequently, the vulnerability management framework applies a statistical approach to compute an estimate of the security issue. More specifically, the vulnerability management framework utilizes the information present in the common vulnerability enumeration (CVE) database[21], to identify known vulnerabilities that are owing to the exact same code anti-patterns to which the current security issue is associated to. This database will be denoted as *VulDB*. For each vulnerability $vul \in VulDB$, the following fields are retrieved:

   - $id(vul)$, which corresponds to the id of the vulnerability
   - $weak(vul)$, which denotes the set of common weaknesses to which the vulnerability is associated. For instance, for the vulnerability with an ID equal to *CVE-2009-0003* it holds that $weak(CVE\text{-}2009\text{-}0003) = CWE\text{-}119$, i.e., vulnerability *CVE-2009-0003* is associated with the CWE having an ID equal to *CWE-119*, corresponding to the anti-pattern of improper restriction of operations within the bounds of a memory buffer[22], commonly referred to as *buffer overflow*.
   - $impact(vul)$, which corresponds to the impact of the vulnerability, i.e., a measure of the adverse effects that the exploitation of the vulnerability by attackers may have on the platform. The value of $impact(vul)$ is assigned by human experts, after careful review of the application code.

[17] `https://rules.sonarsource.com/`
[18] `https://rules.sonarsource.com/java/RSPEC-6437`
[19] `https://cwe.mitre.org/data/definitions/798.html`
[20] `https://cwe.mitre.org/data/definitions/259.html`
[21] `https://cve.mitre.org/data/downloads/`
[22] `https://cwe.mitre.org/data/definitions/119.html`

If the current security issue S_i is associated to weaknesses $W(S_i) = CWE_1, CWE_2, \ldots CWE_n$, then the impact estimate of S_i, which will be denoted as $IE(S_i)$, is computed, as shown in Equation (3.1):

$$IE(S_i) = \max_{cwe \in W(S_i)} \frac{1}{|Vuls(cwe)|} * \sum_{vul \in Vuls(cwe)} impact(vul) \tag{3.1}$$

where $Vuls(cwe) = v \in VulDB : cwe \in weak(v)$, i.e., the set of all vulnerabilities in *VulDB* that are rooted to the particular CWE. Effectively, for each weakness that is associated with the current security issue, the average impact values of all known and human expert-assessed vulnerabilities rooted to the particular weakness are calculated, and finally the maximum of these values is used as the impact assessment for the security issue under the premise that attackers may pursue the weakness path that will result in the maximum possible damage to the system.

3.2.6 Prioritizing Security Fixes

Following the stages of (a) security issue identification and (b) extraction of attributes for each security issue (potential for remote exploitation; impact on confidentiality, integrity and availability; time required for fixing the vulnerability), the framework executes a security issue prioritization step. The goal of this step is to consider a security budget allocation (which is expressed in terms of working hours), and arrange for assigning portions of the budget to the correction of security issues, in order to minimize the residual risk, i.e., the risk owing to the impact of security issues that cannot be fixed, due to security budget constraints.

The security issue prioritization step is performed using linear programming [198]. More specifically, the framework formulates an integer programming optimization problem, where the optimization goal is the minimization of the residual security risk, whereas the available security budget is modeled as a constraint. The formulation of the integer programming optimization problem is presented in detail in the following paragraphs. In this formulation, the notations are given.

Table 3.1: Notations used in the integer programming problem formulation.

Notation	Description
S_i	The ith security issue
N	The number of security issues detected
SB	The available security budget, expressed in available working hours
RE_i	1 if S_i is remotely exploitable, otherwise 0
$I_{conf}(S_i)$	The impact of S_i on confidentiality
$I_{integ}(S_i)$	The impact of S_i on integrity
$I_{avail}(S_i)$	The impact of S_i on availability
$I(S_i)$	The overall impact of S_i on the IoT platform, considering all security dimensions (confidentiality, integrity, and availability)
FT_i	The time needed to fix S_i, expressed in working hours
x_i	An output variable of the problem; x_i is set to *1* if security budget is allocated to fixing S_i, otherwise x_i is set to *0*.

At this stage, we consider that no detailed information is available for the following:

- *The importance of each security dimension, either globally or per specific software deployment*. For instance, a web server may be used to make a public database available, and the integrity of the database records may be deemed more important than the availability of the service, while confidentiality may be considered of low importance (since the database is public). On the contrary, for a web server managing a health record database, all the security dimensions (confidentiality, integrity, and availability) may be deemed of high importance.

- *The importance of each software deployment*. In the previous example, the impact of any demotion of the value of the public database may be deemed to be lower than the impact of a corresponding demotion in the value of the medical record database.

Under the absence of the information listed above, the algorithm will operate under the assumption that (a) all security dimensions are of equal importance and (b) all software deployments are of equal importance. Extensions that will consider potential sources of additional information that will enable the algorithm to take into account variations in the importance of security dimensions and resources are considered part of our future work.

The formulation of the integer programming problem proceeds as follows:

- Firstly, the optimization target is formulated. Since the goal of the optimization is the minimization of the residual risk, which is mapped to the impact of the software vulnerabilities that remain unfixed, the objective function of the integer programming problem is

$$minimize \sum_{i=1}^{N} RE_i * (1 - x_i) * I(S_i) \tag{3.2}$$

where $I(S_i)$ is the security impact of vulnerability S_i. The impact can be directly drawn from the vulnerability impact assessment step, as detailed in Section 3.2.5; if the vulnerability impact assessment step produces separate assessments of the vulnerability on the different security dimensions (confidentiality, integrity, and availability), the overall impact of the vulnerability can be estimated as the sum of its individual impact on each security dimension, i.e.,:

$$I(S_i) = I_{conf}(S_i) + I_{integ}(S_i) + I_{avail}(S_i) \tag{3.3}$$

Note that in Equation (3.2) the impact of each vulnerability $impact(S_i)$ is multiplied (a) by the quantity RE_i, positioning it so that only the impact of remotely exploitable vulnerabilities is considered, and (b) by the quantity $(1 - x_i)$, positioning it so that only the impact of security issues that remain unfixed is taken into account.

- Subsequently, the cost of implementing fixes to the security issues is calculated using the formula

$$Cost = \sum_{i=1}^{N} x_i * FT_i \tag{3.4}$$

In Equation (3.4) the cost FT_i of fixing a security issue S_i is multiplied by variable x_i positioning it so that only the cost of fixes that are selected to be applied is considered.

- Finally, the security budget constraint is applied, which is formulated as follows:

$$Cost \leq SB \tag{3.5}$$

The solution of the integer programming optimization problem is a set of value assignments to variables x_i

$$\begin{gathered} Solution = \{x_1 = v_1, x_2 = v_2, \ldots, x_N = v_N\} \\ \text{where } v_i = \begin{cases} 1 & \text{if security issue } S_i \text{ is selected to be fixed,} \\ 0 & otherwise \end{cases} \end{gathered} \tag{3.6}$$

3.3 Results

To validate our approach, we conducted experiments to (a) provide a proof of concept for the proposed software vulnerability management framework and (b) gain insight on the quality of the vulnerability prioritization recommendations produced by the framework.

In these experiments, we used the following software components and platform setups:

1. Software components implementing the five most widely used technologies in IoT networks [199], both individually and as elements of an IoT platform;

2. An indicative small office/home office (SOHO) configuration.

Since, to the best of our knowledge, no commercial system or research proposal offers prioritization of addressing the vulnerability software components, considering the impact of each vulnerability, the associated technical debt for its remediation, and the available security budget, the following baselines were used to assess the effectiveness of the prioritization mechanism:

1. *BCMM*, i.e., an approach according to which vulnerabilities are processed according to the characterization assigned by the SonarQube analysis [200], and more specifically **b**locker vulnerabilities are handled first, followed by vulnerabilities characterized as **c**ritical, **m**ajor, and **m**inor, in that order. Considering that for the vulnerabilities reported by Sonar, only the technical debt (fix time) is available, three variants of BCCM are considered, namely (a) $BCCM_{SFT}$, where within each categorization, vulnerabilities with the shortest fix time are handled first, (b) $BCCM_{LFT}$, where within each categorization vulnerabilities with the largest technical debt are handled first, and (c) $BCCM_{Rand}$, where vulnerabilities within each categorization are considered in random order.

2. *IMM*: According to the descriptions given for SonarQube vulnerability characterizations [200], *blocker* issues are bugs with a high probability to impact the behavior of the application in production, and should be fixed immediately and *critical* issues are bugs with a low probability to impact the behavior of the application in production or issues that represent security flaw vulnerabilities and must also be fixed immediately. Since both classes are designated to require immediate handling, in the *IMM* approach they are merged to a single class, **i**mmediate, while issues in the **m**ajor and **m**inor classes are retained in their original characterization. Similarly to the *BCMM* approach, three variants are considered, namely IMM_{SFT}, IMM_{LFT}, and IMM_{Rand}.

3.3.1 Experiments for the Commonly Used IoT Technologies

In this subsection we present our experiments concerning a configuration which comprises the five most widely used technologies in the IoT. As reported by [199], the five most widely used technologies in IoT networks are:

1. The *advanced message-queuing protocol* (AMQP), an open standard protocol used for message exchange, including publish/subscribe and point-to-point, as well as queues [201],

2. *Bluetooth and Bluetooth low-energy* (BLE), a short-range communication protocol and its low-energy variant [202],

3. *Cellular communications*, i.e., implementation of communication through cellular telephony netowrks (2G, 3G, 4G/LTE, and 5G),

4. The *constrained application protocol* (CoAP) [203], a specialized internet protocol for devices with constrained resources (e.g., wireless sensors), which enables both (a) pairs or groups of devices running CoAP and (b) devices running CoAP and the internet.

5. The *data distribution service* for real-time systems (DDS) [204], a networking middleware for realtime systems specified by the object management group (OMG), realizing data-centric publish-subscribe mechanisms which can be easily integrated in the application layer.

Following these data, an IoT system configuration was formulated, running instances of software implementing the above listed technologies as follows:

- AMQP was implemented using RabbitMQ v. 3.4.0[23],
- Bluetooth/Bluetooth LE was implemented using the Android 13 drivers[24]
- Cellular communications were implemented using Open5GS v. 2.1.3[25]
- CoAP was implemented using the CoAP library in Arm Mbed OS 5.14.0 [26]
- DDS was implemented using OpenDDS v. 3.16.1[27]

The software listed above was analyzed and was found to entail 47 vulnerabilities, accounting for a total technical debt of 967 min, with an overall risk equal to 273.34. In the prioritization experiments we considered the security budget values of 50, 125, 250, and 500 min. Table 3.2 lists indicative results from applying the security issue prioritization method described in Section 3.2.6 to the identified security issues. As shown in Table 3.3, the recommendation in all cases consumes (almost) all the available budget, and manages to effectively direct the available budget to the mitigation of the issues having the highest impact, since the percentage of the total impact mitigated in the recommendation list is consistently higher than the ratio of the available budget to the total technical debt.

Table 3.2: Results of applying the security issue prioritization to the commonly used IoT technologies' configuration.

Security Budget	**# Issues Mitigated**	**Consumed Budget**	**Impact Mitigated**	**% of Total Budget Available**	**% Issues Mitigated**	**% Impact Mitigated**
50	8	47	58.95	5.17%	17.02%	21.57%
125	15	122	98.25	12.93%	12.62%	35.94%
250	23	247	144.13	25.85%	48.94%	52.73%
500	33	497	210.52	51.71%	70.21%	77.02%

[23] https://github.com/rabbitmq/rabbitmq-server/tree/rabbitmq_v3_4_0

[24] https://android.googlesource.com/kernel/msm/+/refs/tags/android-13.0.0_r0.1/drivers/bluetooth/

[25] https://github.com/open5gs/open5gs/releases/tag/v2.1.3

[26] https://github.com/ARMmbed/mbed-os/releases/tag/mbed-os-5.14.0

[27] https://github.com/OpenDDS/OpenDDS/releases/tag/DDS-3.16.1

Table 3.3: Results of applying the security issue prioritization to the SOHO configuration.

Security Budget	**# Issues Mitigated**	**Consumed Budget**	**Impact Mitigated**	**% of Total Budget Available**	**% Issues Mitigated**	**% Impact Mitigated**
250	13	245	107.46	8.63%	13%	14.88%
500	22	500	185.899	17.61%	22%	25.74%
1000	38	980	308.05	34.51%	38%	42.65%
1500	55	1490	437.78	52.46%	55%	60.61%

The results in Table 3.2 demonstrate that especially when the security budget is very limited, the scarce resources can be efficiently allocated to the mitigation of security issues with a very high impact, which is testified by the fact that for the case of having a security budget equal to 50, the ratio of the mitigated impact percentage is 21.57%, approximately 4 times higher than the percentage of the available budget. When the available security budget increases, this performance margin narrows, declining to the value of approximately 1.5 times higher when the security budget is equal to 500 (or 51.71% of the total available budget).

Figure 3.2 depicts the effectiveness of the proposed approach against the baseline algorithms concerning the configuration including the commonly used IoT technologies. We can observe that under all security budgets, the proposed approach achieves the highest percentage of the mitigated impact, with a margin ranging from 0.51% to 23.14% against the runner-up (which is the IMM_{SFT} algorithm in all cases) when comparing absolute magnitudes (i.e., $impactMitigated(proposed) - impactMitigated(baseline)$); when considering relative improvements (i.e., $\frac{impactMitigated(proposed) - impactMitigated(baseline)}{impactMitigated(baseline)}$), the effectiveness margin of the proposed algorithm ranges from 0.87% to 12.35%. When the security budget is very small (50, i.e., approximately 5% of the total technical debt), the performance edge of the proposed algorithm is small (relative improvement equal to 0.87%), due to the fact that many vulnerabilities with small fix times with "blocker" and "critical" characterizations have high impacts; hence the IMM_{SFT} algorithm is circumstantially led to "close to optimal" decisions, due to the combined distribution of the impact, criticality level, and technical debt of the software vulnerability dataset.

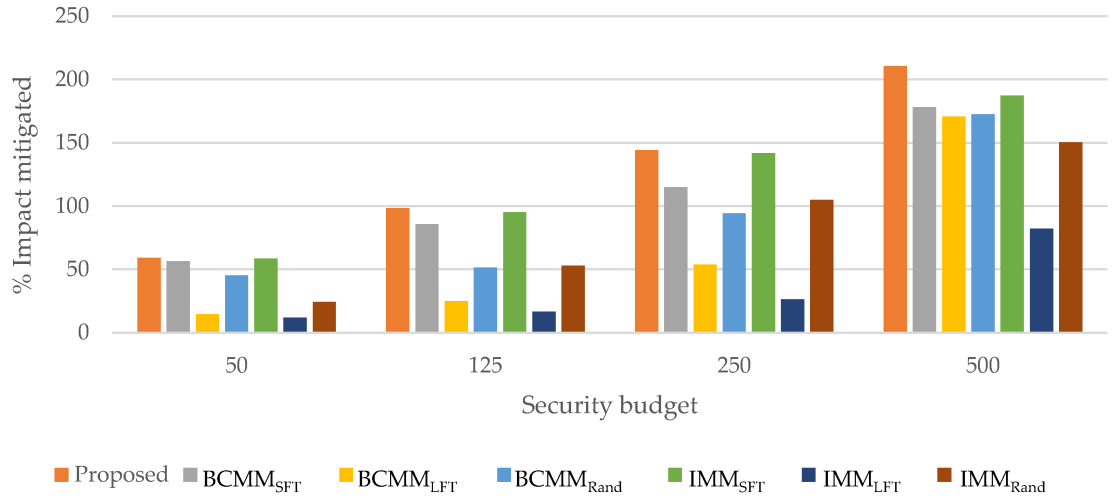

Figure 3.2: Effectiveness of security issue fix prioritization algorithms for the commonly used IoT technologies' configuration.

This distribution is depicted in Figure 3.3; in this figure, we can partition vulnerabilities into four quartiles, with Q1 including vulnerabilities with a low cost to fix and their remediation results to high gains in the residual risk of the overall configuration. The presence of numerous "blocker" and "critical" vulnerabilities in this quartile is the reason that leads to the "close to optimal" performance of the IMM_{SFT} for the constrained security budget.

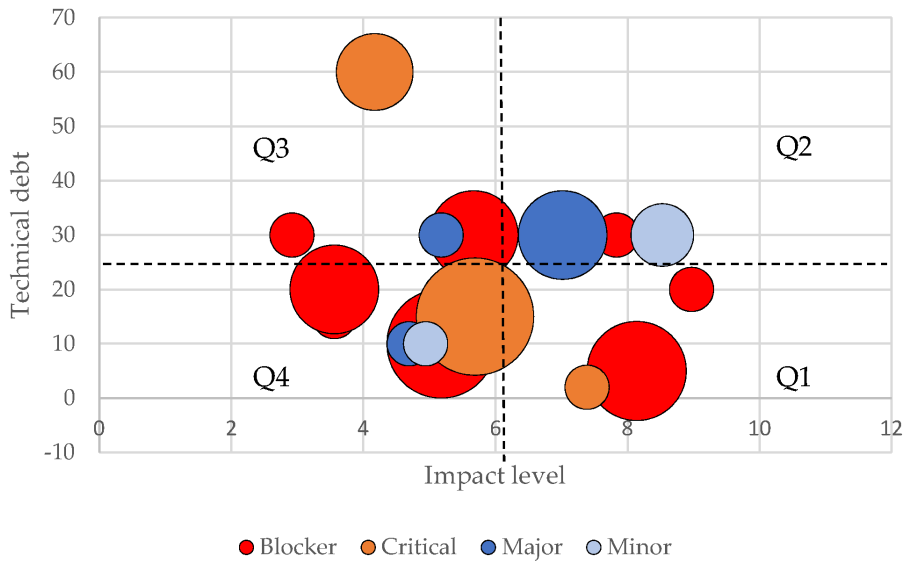

Figure 3.3: Distribution of the impact, criticality level, and technical debt of the software vulnerabilities in the "commonly used IoT technologies" configuration.

The margin between the proposed approach and other algorithms is larger, ranging from 2.72% to 32.40% in absolute magnitudes, while the corresponding relative improvement ranges from 4.84% to 496.18%. This performance margin is attributed to the capability of the proposed algorithm to direct the security budget

to issues whose fixing will lead to the largest reductions in the residual risk. The "largest fix times first" approach again produces the worst results, because each fix applied consumes a large amount of security budget, leading to its depletion without necessarily achieving a respectively high reduction of the residual risk.

3.3.2 Experiments for the SOHO Configuration

In this subsection we present our experiments concerning a small office/home office (SOHO) configuration, whose architecture follows the typical SOHO architectural style, i.e., the platform a is configured as a "flat" network, where network connectivity is realized by a single device acting both as (a) a layer 2 switch accommodating both wired and wireless protocols for internal nodes (which are few), and (b) as a router providing internet connectivity [205, 206]. The specific topology used in the experiment comprises:

1. A router running PFsense (`https://github.com/pfsense/pfsense/tree/a81a848e7565cf4b5e1679fe6d08c39d13ab7a6f` (accessed on February 25, 2025)).),
2. A NAS appliance running the Minnow Server (`https://github.com/RealTimeLogic/MinnowServer`, (accessed on February 25, 2025))
3. A smart air-conditioning appliance running the Pymodbus software (`https://github.com/pymodbus-dev/pymodbus`, (accessed on February 25, 2025))
4. A mobile phone and a PC which include Modbus4j software (`https://github.com/MangoAutomation/modbus4j`, (accessed on February 25, 2025)) in order to control the air-conditioning appliance.

The SOHO topology corresponding to this configuration is illustrated in Figure 3.4.

The source code for the software components listed above was collected and processed through the software vulnerability management framework pipeline illustrated in Figure 3.1 and detailed in Sections 3.2.1–3.2.6. In the security fix prioritization step, multiple values were used for the security budget to gain insight on the effect of this parameter in the recommendations formulated, regarding the list of software issues to be mitigated.

The analysis of the software identified 100 security issues with the code, with a total impact equal to 722.24 and an estimated technical debt equal to 2840 min, i.e., a security budget of 2840 developer working minutes is required to mitigate all security issues. Table 3.3 lists indicative results from applying the security issue prioritization method described in Section 3.2.6 to the identified security issues. As shown in Table 3.3, the recommendation in all cases consumes (almost) all the available budget, and manages to effectively direct the available budget to the mitigation of the issues having the highest impact, since the percentage of the total impact mitigated in the recommendation list is consistently higher than the ratio of the available budget to the total technical debt.

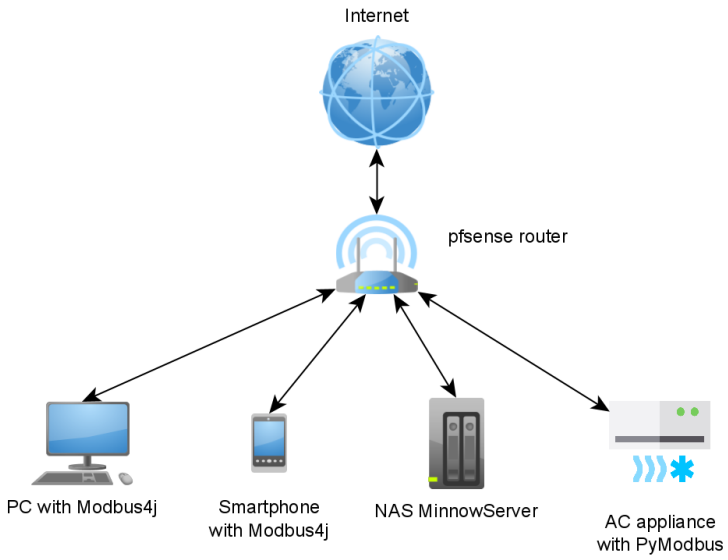

Figure 3.4: Proposed framework architecture (`https://www.flaticon.com/`, (accessed on February 25, 2025)).

In Table 3.3, we can also notice that especially when the security budget is limited, the constrained resources can be efficiently allocated to the mitigation of security issues with a very high impact, which is demonstrated by the fact that for the case of having a security budget equal to 250, the ratio of the mitigated impact percentage is 1.72 times higher than the percentage of the total available budget. When the available security budget increases, this performance margin narrows, declining to the value of 1.15 times when the security budget is equal to 1500 (or 52.46% of the total available budget).

Figure 3.5 illustrates the effectiveness of the proposed approach against the baseline algorithms. We can observe that under all security budgets, the proposed approach achieves the highest percentage of mitigated impact, with a margin ranging from 1.42% to 6.78% against the runner-up (which is the IMM_{SFT} algorithm in all cases) when comparing absolute magnitudes (i.e., $impactMitigated(proposed) - impactMitigated(baseline)$); when considering relative improvements (i.e., $\frac{impactMitigated(proposed) - impactMitigated(baseline)}{impactMitigated(baseline)}$), the effectiveness margin of the proposed algorithm ranges from 10.63% to 12.59%. The margin between the proposed approach and other algorithms is larger, ranging from 6.15% to 12.73% in absolute magnitudes, while the relative improvement ranges from 14.06% to 88.45%. This performance margin is again attributed to the capability of the proposed algorithm to direct the security budget to issues whose fixing will lead to the largest reductions in the residual risk. The "largest fix times first" approach again produces the worst results, because each fix applied consumes a large amount of security budget, leading to its depletion, without necessarily achieving a, respectively, high reduction of the residual risk.

Detailed information on the security issues identified, the integer programming problem formulations used for the prioritization of security issue mitigations and the

solutions of these problems is available on GitHub[28].

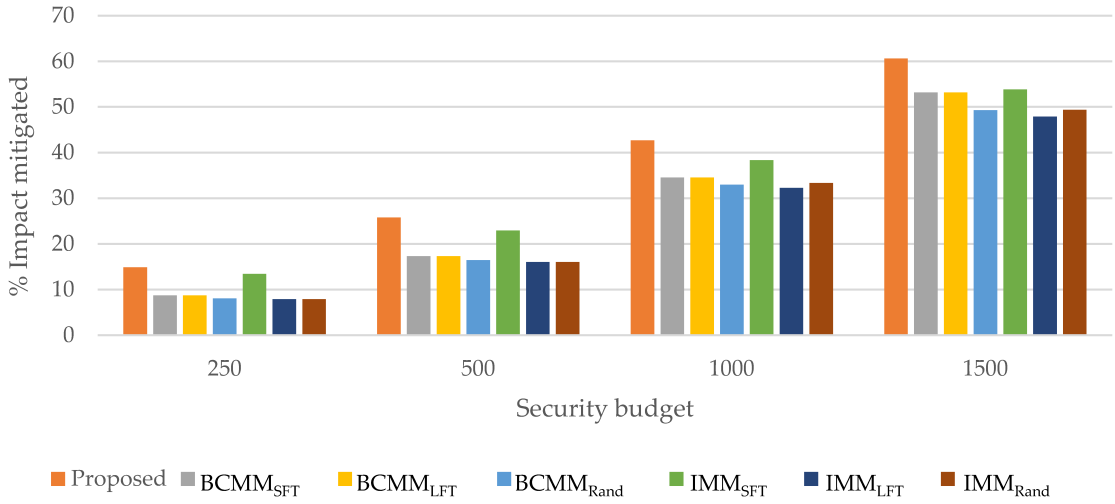

Figure 3.5: Effectiveness of security issue fix prioritization algorithms for the SOHO configuration.

3.4 Discussion

The framework presented in this chapter and the methods proposed to realize the constituent components of the related workflow can be used by practitioners and researchers alike.

As far as the practical implications of this work are concerned, the proposed framework can be used as an aid in IoT platform implementation by software architects, developers, and security experts, supporting the task of minimizing the overall residual risk. The proposed framework may be also utilized to support different code development and maintenance tasks; for instance, issues of type *BUG* can be extracted from the results of the analysis performed by SAST tools, and the prioritization step could be applied to the bug list in order to guide developers in addressing bugs, or—more generally—improving code quality. Under the design-by-contract approach [207, 208], the proposed framework may automatically perform security assessments of alternative implementations of the same contract, and automatically bundle into the executable the implementation providing higher security levels, or issue relevant recommendations to the developers.

In the research domain, starting from the proposed framework, a number of aspects may be further analyzed and elaborated on. The first topic that can be explored is the exploitation of attack graphs [209, 210] to fully consider all possible attack paths to the IoT infrastructure and their repercussions, since remote attacks with low impacts may act as stepping stones for additional attacks that may expose the infrastructure to more serious adverse effects, e.g., by combining a remotely exploitable attack offering to the attacker low privileges with an attack that can be executed locally only and offers privilege escalation [211].

[28] `https://github.com/costasvassilakis/vulnerability-management-framework`

Static security code analysis may be complemented with dynamic security code analysis [212] to uncover a complex flaws or vulnerabilities that cannot be identified by SAST tools due to their perplexed nature.

The estimation of the impact of security issues may be further refined by considering the similarity of the code entailing each particular security issue with code fragments that are rooted to the same weakness and for which the impact has been assessed by human experts. Approaches that assess the functional similarity of code [213] may be used to that effect.

Concerning the prioritization of security issue fixing, a number of extensions are envisioned. The current algorithm assumes the equal importance of all resources in the IoT platform, however this may not hold in all environments; for instance, in the example presented in Section 3.3, the air-conditioning appliance can be deemed of lower importance than the PC or the NAS appliance, hence software issues with the software running in the air-conditioning appliance may be assigned a lower fixing priority. The business importance of each appliance could be provided externally by human experts and subsequently be considered by the algorithm. Similarly, the prioritization algorithm could be extended to accommodate diverse importance and could be applied to different security dimension of each appliance or resource: for instance, a public information database may have low requirements for confidentiality and high requirements for integrity and availability, while a health record database would have high requirements for all security dimensions.

Finally, this work may be adapted and used in numerous areas of the software engineering domain, including performance/stress testing and identification of hotspots, testing in different deployment environments, etc.

3.5 Conclusions

In this chapter, we have presented a software vulnerability management framework which supports all the stages of a pipeline for the management of IoT platform software vulnerabilities. More specifically, the framework supports (a) the configuration of software to include only the necessary features, (b) the execution of security-related tests and the compilation of platform-wide software vulnerability lists, (c) the estimation of the impact and the associated fixing cost for each vulnerability, and (d) the prioritization of vulnerability addressing (considering the impact of each vulnerability) the associated technical debt for its remediation and the available security budget.

The work presented in this chapter advances the state-of-the-art by (i) proposing a statistics-based method for the estimation of impact of detected vulnerabilities, (ii) proposing an integer programming-based algorithm for prioritizing security fixes with the goal of minimizing the residual risk level, and (iii) proposing a comprehensive framework for the security analysis of platform software which formulates proposals on the prioritization of security issue addressing, taking into account all the aspects (a)–(d) listed in the previous paragraph.

Our future work will focus on the incorporation of dynamic security code analysis and attack graphs into the workflow, as well as the refinement of vulnerability impact estimation.

Chapter 4

Minimizing Software-Rooted Risk through Library Implementation Selection

Abstract

In contemporary Internet of Things (IoT) systems, complex software artefacts are deployed to realize the required functionalities. The business logic of these software artefacts is uniquely composed through code that is customly developed according to the requirements, all software artefacts utilize libraries that implement generic functionalities, which are needed in the context of the realized operations. Libraries, however, often entail vulnerabilities, which may be exploited by threat agents to attack the system. In many cases, the functionality required by an application is realized by a number of alternative libraries, with each library having its own list of vulnerabilities, while differentiations in other non-functional properties (e.g. execution efficiency, memory footprint etc.) may also be present. In this chapter, we present an approach for automating the task of minimizing the risk level of IoT systems that is owing to the vulnerabilities of libraries required by software artefacts. The proposed approach exploits knowledge on which libraries provide equivalent implementations of the same functionalities, and automatically assesses the risk level of candidate library combinations and finally selects the library configuration exhibiting the minimum risk level to bundle into the executable software artefact. Additionally, the risk level of candidate implementations is constantly monitored for new vulnerability identifications or fixes in the implementations, triggering new risk assessments and producing new executables as appropriate. The proposed approach can be used in IoT platform deployment to minimize the software-rooted risk level.

4.1 Introduction

In modern Internet of Things (IoT) systems, individual devices offer and consume functionalities, with each functionality being realized by software installed and operating on a device. This software is customly developed and/or tailored for the particular platform deployment, realizing business processes and requirements

collected and documented in the analysis phase of the software lifecycle. All software artefacts utilize libraries that implement generic functionalities, which are needed in the context of the realized operations.

Libraries, however, often entail vulnerabilities, which may be exploited by threat agents to attack the system. In many cases, the functionality required by an application is realized by a number of alternative libraries, with each library having its own list of vulnerabilities, while differentiations in other non-functional properties (e.g. execution efficiency, memory footprint etc.) may also be present among different library implementations or versions: for instance, the functionality of logging for Java programs may be provided using –among others–(a) the Log4j2 library provided by the Apache group [214], (b) the tinylog library [215] or (c) the logback library [216], and a developer may chose any of these libraries to realize logging for a software artefact. The Log4j2 library suffers from the well-known *log4shell* vulnerability, which is present in versions 2.0-beta9 through 2.15.0 (excluding security releases 2.12.2, 2.12.3, and 2.3.1) [217]. On the other hand, the logback library suffers from an arbitrary code execution vulnerability, which is present in its versions up to and including 1.2.7. Considering the above, we can observe that the choice of the library that implements some functionality required for a software artefact, and the selection of the exact version of the library that will be bundled into the executable, ultimately affects the security properties of the executable and –more generally–the security properties of the platform in the context of which the executable is run.

In order to minimize the risk owing to the libraries bundled in a software artefact, a developer should

a) review the functionalities in the relevant project that are supported by libraries,

b) identify the candidate libraries for the realization of each functionality, and the versions of each candidate library that fulfill the required functional and non-functional requirements; for instance, in order to use MongoDB v4 as a storage backend for logging using Log4J2, Log4J2 version 2.14.0 or newer is required [218],

c) assess the risk associated with each usable version of candidate libraries, considering the vulnerabilities present in the library and the impact of each vulnerability,

d) select the libraries and the specific library versions that minimize the overall risk, and

e) perform amendments to the application code to cater for differences in the API between the different libraries.

The procedure listed above entails considerable additional work that should be performed by the developers, while it is also tedious and error-prone. Moreover as vulnerabilities are continuously discovered and new libraries and library versions are released, the pool of library/library version choices is expanded and the risk associated with each library version is modified, necessitating thus a continuous review, risk assessment and library version choice process, which further increases the workload assumed by developers.

In this chapter, we present an approach for automating the task of minimizing the risk level of IoT systems that is owing to the vulnerabilities of libraries required by software artefacts. The proposed approach exploits knowledge on which libraries provide equivalent implementations of the same functionalities, automatically assesses the risk level of candidate library combinations and finally selects the library configuration exhibiting the minimum risk level to bundle into the executable software artefact. Additionally, the risk level of candidate implementations is constantly monitored for new vulnerability identifications or fixes in the implementations, triggering new risk assessments and producing new executables as appropriate.

The rest of the chapter is organized as follows: section 4.2 presents related work, while section 4.3 presents the proposed approach. Finally, section 4.4 concludes the chapter and outlines future work.

4.2 Related work

In this section we overview related work from the domain of *static and dynamic analyzers*, which are used for uncovering bugs within the code. These methods can be also used with libraries, either in the context of white box testing (assessment of code, especially when it is available e.g. for open-source libraries), or in the context of black box testing (only the externally observable behavior of libraries is assessed). We also provide an overview of the *Additional Testing Framework*, a test management and execution framework which is utilized for the instrumentation of the proposed approach.

4.2.1 Static and dynamic analyzers

The method of Static Code Analysis [178] is an essential part of the Security Development Lifecycle [183] and can be performed manually or with the use of Static Code Analysis tools. It involves reviewing the code of the tested software using tools that can detect potential security vulnerabilities. Static analysis can be conducted on both source and compiled code. Static analysis checks code patterns statically to identify problematic areas, rather than inspecting them at runtime. While non-security issues may also be detected through this process, our focus is solely on security issues. The term used for static analysis focusing on security issues is Static Application Security Testing (SAST). The complexity of SAST varies depending on the implementation, but it may involve tasks such as producing control graphs and analyzing data flow logic to identify sensitive code segments that may be reached by user input [184].

SAST has gained widespread acceptance in recent years, with various forms of tools being available, such as IDE plugins, standalone applications, online services, and solutions integrated into CI/CD pipelines [185]. Some of the techniques employed by static analysis tools include –but are not limited to– Lexical Analysis, Control Flow Graphs (CFGs), Taint Analysis, and Data Flow Analysis [186]. Despite recent advancements, SAST tools still suffer from high false-positive rates, requiring human intervention for results evaluation [180]. Therefore, SAST is necessary but also complementary to more traditional black-box techniques used in vulnerability

detection in IoT systems, such as fuzzing, taint-analysis and symbolic execution [187, 188, 189].

The stages of SA that are taken into account include Data collection, Data preparation, Data processing, and Result Formation; these stages map to the columns of the SA model matrix. To assign each activity to the stages of SA, a formalization technique is used based on the representation of any given data in terms of its content (C) and form (F). Content (C) refers to the information it contains and form (F) refers to its appearance. Consequently, data stored within the IoT system and changed during the SA process can be expressed as <F|C> tuple. Using this approach, a formal description is given for each stage.

In the domain of dynamic analyzers, Akhilesh et al. [193] propose an automated penetration testing framework for smart home IoT devices. The authors' aim is to create a framework that is fully automated and easy to use for both technical and non-technical users, allowing anyone to assess the security of the devices in their smart home network. To achieve this, the authors reviewed and compared relevant research to select the most appropriate work to adopt, taking into account differences and limitations. Zheng et al. [194] introduce a new method for greybox fuzzing of linux-based IoT devices. While greybox fuzzing is an effective technique for discovering vulnerabilities, it encounters several limitations when applied to IoT devices.

4.2.2 The Additional Testsuite Framework: An Overview

The Additional Testsuite Framework (ATF) [131], prsented in the first and second chapters, is novel approach for the management of code testsuites, providing efficient management of tests, rule-based association of tests with specific builds and relevant structures for testing multi-versioned and multi-featured software programs supporting –among other capabilities–test-driven development, dynamic/selective program builds, feature-based builds. ATF uses annotations to collect tests with specific characteristics (e.g. for specific versions or with specific features), which will be executed for a particular build. Utilizing the annotation metadata of the tests included in the final testsuite, the modules and libraries that are necessary for the tests to succeed can be gathered, in order to create the executable image to deploy to the designated environment.

Some advantages of ATF are the following:

1. The tests are written once and can be distributed to test any number of software programs and versions.

2. The tests can be stored in a single place / repository or in general a close set of repositories.

3. It facilitates the comparison of different software programs and versions, based on the supported features of distributed tests to each testsuite.

4. Since any test can be executed with any version of the software program tested, it can be used for fixing any bugs by comparing the code of the successfully tested version with the erroneous one.

5. It supports the concept of features, enabling feature-based testing and dynamic building of executables based on given feature lists.

6. It can support test-driven development, through proactive coding of tests for features that will be implemented in future program versions; these tests will remain latent until the new features are integrated in the software program.

7. Feature-based tests can be included in documentation on the feature they examine, serving as examples of the specific feature as well as providing examples of the feature's usage in combination with other features.

For more information regarding the capabilities and functionality of the ATF, the interested reader is referred to [131]. The AT Framework assumes that all available functionalities of each server / service are available at the tests used / distributed from the global test repository for this specific server / service. Each of the tests of the server / service is dependent on a subset of the server / service modules and on the client code and libraries that use the server / service functionalities. In the following, we will assume that all tests required for the particular deployment of the software are properly entered in the ATF repository.

4.3 Proposed Approach

The architecture of the proposed approach is illustrated in Figure 4.1. According to the Additional Testsuite Framework architecture [131], both the application code and the tests are stored in a single repository. This repository, following the software development practices, also stores the software configuration, in the form of Makefiles [7], the Project Object Model (POM) [8], or other equivalent representations, with the software configuration notably including the libraries that need to be bundled within the executable.

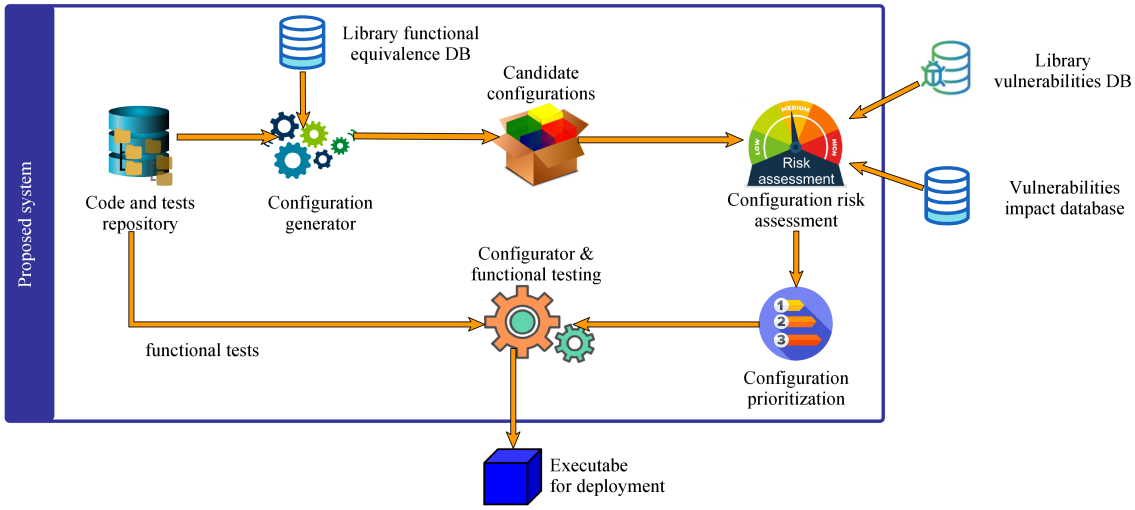

Figure 4.1: Proposed approach

The processing pipeline of the proposed approach for building the executable proceeds as follows:

1. firstly, the *configuration generator* module extracts from the software configuration the libraries that are needed to build the software artifact and, for each library L, it consults the *library functional equivalence database* (LFEDB) to identify other libraries that provide the same functionality with L, creating a set of candidate libraries for L, considering also the different library versions that can be utilised. Then it proceeds with creating candidate configurations, with each configuration corresponding to a different library/library version combination.

2. afterwards, each configuration is assessed to determine the risk level associated with it, by (a) extracting the vulnerabilities that are present in the libraries it uses and (b) estimating the impact of these vulnerabilities to the security of the software artifact.

3. Subsequently, the configurations are sorted in increasing order of their risk assessment, i.e. configurations with low risk are prioritized.

4. Finally, *configurator & functional testing* module traverses the list of prioritized configurations, and for each configuration the relevant executable is built and the functional tests for the target deployment are extracted from the Additional Testsuite Framework repository and executed. If all functional tests succeed for some configuration, then this configuration is chosen and the corresponding executable is qualified for deployment on the IoT platform; otherwise, the *configuration functional testing* module proceeds with the testing of the remaining configurations in the priority list.

In the following subsections, we present the steps of the processing pipeline in detail. We note here, that the design decision to perform the configuration risk assessment and prioritization before the functional tests was based on the rationale that configuration risk assessment is computationally cheaper than conducting functional tests, and therefore it is beneficial to arrange the pipeline stages in an order that minimizes the number of functional testings that need to be executed.

4.3.1 The Configuration Generator Module

The Configuration Generator module utilizes the information in the program configuration to identify the libraries that are used to build the software artifact. Libraries are specified in a configuration format-dependent fashion, e.g. Apache Maven configurations typically utilize the `<dependency>` element, whereas in *makefiles*, libraries are typically specified using the `LIBS` variable. For each library, if specific versions are designated (e.g. in Apache Maven a specific version –e.g. 1.0.1.0–or a range of versions –e.g. *[1.0,2.0)*–) can be specified using the `<version>` element in dependencies), this version specification is retained, otherwise the version specification for the particular library is set to *null*, and all versions of the library are considered as candidates. The information extracted regarding the required libraries is stored in a set $L = \{(L_1, vs_1), (L_1, vs_2), \ldots, (L_n, vs_n)\}$, where L_i represents a required library and vs_i represents the corresponding version specification.

Subsequently, for each library L_i, the Configuration Generator module queries the library repository (e.g. `repo.maven.apache.org`) to identify the specific versions of library L_i that match the respective version specification; if the version specification is equal to *null*, all versions of L_i are retrieved. Through this step, the set of required libraries L is transformed to $LV = \{(L_1, vset_1), (L_1, vset_2), \ldots, (L_n, vset_n)\}$, where $vset_i$ is an explicit listing of the qualifying versions of library L_i.

Afterwards, the Configuration Generator module utilizes the *Library functional equivalence database* (LFEDB), a database storing information about (a) which libraries realize the same functionalities, (b) which versions of functionally equivalent libraries are compatible and (c) transformations that may be needed so that some library L_i could be replaced by a functionally equivalent library L_i^j. To exemplify this aspect, consider some application that utilises JSON representations, which are realised using the Jackson library [219]. The LFEDB lists that the Gson library [220] provides equivalent functionality (i.e. JSON representations). If it is determined that the Gson library can offer better security levels, in order to replace the Jackson library with Gson, the following transformations should be made to the application:

1. in the program configuration, artifact `com.fasterxml.jackson.core.jackson-databind` should be replaced by artifact `com.google.code.gson.gson`,

2. imports `com.fasterxml.jackson.core.*` should be transformed to `import com.google.gson.*`,

3. code stubs of the form

   ```
   ObjectMapper om = new ObjectMapper();
   om.writeValue(new File("outputFile.json"), theInstance);
   ```

   where `theInstance` is the object for which the JSON representation will be created, should be transformed to

   ```
   Gson gson = new Gson();
   gson.toJson(theInstance, new FileWriter(new File(outputFile.json)));
   ```

   and similarly for other JSON operations (creating Java objects from JSON etc).

Therefore, the *Library functional equivalence database* contains appropriate transformation specifications, which transform a program using the Jackson library in a form that uses the Gson library instead.

The LFEDB is populated and maintained as part of the proposed approach.

When the aforementioned library equivalence information is retrieved from the LFEDB for a library L_i, the configuration Generator module compiles a list of *artifact candidates* for library L_i, where each artifact candidate is a triple (L_i^j, v_i^j, T_i^j), where L_i^j is the candidate replacement library (e.g. Gson), v_i^j is a version of the candidate replacement library that may be utilised, and T_i^j is a set of transforms that need to be applied on the program code so that it will use version v_i^j of library L_i^j instead of the originally used version of library L_i. Note that all usable versions of the library originally specified in the program configuration are included in the artifact candidates, since they will be also considered for inclusion in the final executable, along with their functionally equivalent counterparts. For the versions of L_i initially specified in the program configuration, the relevant transformation specification will be *null*, and this is also expected to be the case for any other versions of the library with the same

major version number, since equal version numbers typically indicate conformance to the same API; however, the LFEDB may designate non-null transformations, even between versions of the same library, in which case these transformations are retrieved and used. Formally, for each library $Li \in L$, having p replacement candidates $\{L_i^1, \ldots, L_i^p\}$, the corresponding artifact candidate set is formulated as follows:

$$\begin{aligned} AC(L_i) = \{&(L_i, v_{i,1}, null), \ldots, (L_i, v_{i,k_i}, null), \\ &(L_i^1, v_{i^1,1}, T_{i^1,1}), \ldots (L_i^1, v_{i,k_i^1}, T_{i^1,k_i^1}), \\ &\ldots \\ &(L_i^p, v_{i^p,1}, T_{i^p,1}), \ldots (L_i^p, v_{i^p,k_i^p}, T_{i^p,k_i^p})\} \end{aligned}$$

where k_i^j denotes the number of usable versions of the candidate replacement library L_i^j.

The final task of the configuration generator module is to create a distinct configuration for each selection of artifact candidates; formally, the set of generated configurations is

$$Configs = AC(L_1) \times AC(L_2) \times \ldots \times AC(L_p)$$

4.3.2 The Configuration Risk Assessment Module

When the different configurations have been formulated, the Configuration risk assessment module proceeds in computing an estimation of the risk level of each configuration that is owing to the libraries that the configuration involves. For each library version $L_{i,v}$ present in the configuration, the Configuration risk assessment module applies the following steps:

1. it searches the NVD database[29] to locate vulnerabilities that are associated with the particular library version. The search is made using the Common Platform Enumeration (CPE)[30] identifier or the library, which is hosted in the LFEDB and retrieved from there. For each vulnerability V_i^j of library $L_{i,v}$ retrieved from the NVD, the CVSS (Common Vulnerability Scoring System) score of the vulnerability is extracted; this score will be denoted as $CVSS(V_{i,v}^j)$. Then, the risk level of a library version $L_{i,v}$ containing vulnerabilities $V_{i,v}^1, V_{i,v}^2, \ldots, V_{i,v}^k$ is computed as the sum of the CVSS scores of the vulnerabilities, i.e.

$$libraryRiskLevel(L_{i,v}) = \sum_{j=1}^{k} CVSS(V_{i,v}^j)$$

2. Having computed the risk level of each individual library version of a configuration C containing library versions $L_{1,v_1}, L_{2,v_2}, \ldots, L_{m,v_m}$, the risk level of the configuration C is then computed as the sum of the risk level of the libraries it utilises, i.e.

$$configRiskLevel(C) = \sum_{j=1}^{m} libraryRiskLevel(L_{j,v_j})$$

[29] `https://nvd.nist.gov/`

[30] `https://nvd.nist.gov/products/cpe`

The procedure described above considers vulnerabilities that have been identified, analyzed, assessed and recorded into the NVD database. However, numerous new vulnerabilities are identified each day and relevant exploits are crafted; under this view, timely response to these threats is critical for the security of systems.

To facilitate eager response to threats, the proposed framework may be used in conjunction with systems that crawl a variety of online sources, to identify vulnerabilities and exploits, such as the inTIME system presented in [221]. The inTIME system scans clear/deep/dark web sites, retrieving potential vulnerabilities and exploits. In the following, we will only consider the use of unstructured vulnerability information retrieved by inTIME, since any structured information source may be handled similarly the case of NVD, which is described above.

The inTIME system implements an internal *Common Platform Enumeration (CPE) Suggestion Engine*, which is able to correlate a vulnerability with the corresponding platform (which includes libraries). Therefore, the inTIME system may offer information on new vulnerabilities that are present in libraries and have not been recorded in official databases with CVSS score assessments, and for each such vulnerability a textual description is offered. For each such vulnerability V, the approach proposed in this work may estimate the risk level associated with the vulnerability as follows:

1. the text associated with the vulnerability is matched against the Common Weakness Enumeration (CWE)[31] identifiers and descriptions. This step will produce a number of matches against CWE records, such as “Access of Uninitialized Pointer" (CWE-824) or “SQL Injection" (CWE-89). We will denote as $CWEs = \{CWE_1, CWE_2, \ldots, CWE_n\}$ the list of all CWEs that match to the textual description of the vulnerability, while $conf_1, conf_2, \ldots, conf_n$ are the respective match confidence metrics.

2. Afterwards, the proposed approach employs a statistical approach to estimate the risk associated to the vulnerability. In more detail, the system queries the Common Vulnerability Enumeration (CVE) database[32] to extract vulnerabilities that are rooted to the same CWEs. For each such vulnerability vul, the $imp(vul)$ field, denoting the impact level of the vulnerability, which is used as a measure of the risk associated with the presence if this vulnerability, is extracted. The set of vulnerabilities retrieved that is linked to a particular CWE CWE_i will be denoted as VUL_i.

3. Finally, the risk level of the vulnerability V is estimated using equation 4.1:

$$RL(V) = \max_{cwe \in CWEs} \frac{conf_i}{|VUL_i|} * \sum_{vul \in VUL_i} imp(vul) \tag{4.1}$$

   Effectively, each CWE associated with the vulnerability V for which the estimate is computed is assessed separately and the average impact of expert-assessed vulnerabilities rooted to the same CWE is computed; this impact is weighted by the confidence to association of V to the CWE, and finally the

[31] `https://cwe.mitre.org/`

[32] `https://cve.mitre.org/data/downloads/`

maximum of all CWE risk estimates is used as the risk estimate for V, under the rationale that risk agents will exploit the attack path that will incur the maximum damage to the system.

4.3.3 The Configuration Prioritization Module

The Configuration prioritization module receives a list of possible software configurations, each tagged with the risk estimate computed by the configuration risk assessment module, and orders the list in ascending order the estimated risk level. Then, the sorted configuration list is forwarded to the *Configurator and functional testing module*, which will create the final executable for deployment in the target platform.

4.3.4 The Configurator and Functional Testing Module

The Configurator and functional testing module receives as input the list of configurations created by the software configuration generator module, sorted in ascending order of their estimated risk. Recall from subsection 4.3.1 that each configuration is effectively a list of triples (L_i^j, v_i^j, T_i^j), where L_i^j is a library that will be bundled into the executable, v_i^j is the specific version of (L_i^j that will be used, and T_i^j is a set of transforms that need to be applied on the program code so that it will use library L_i^j instead of the originally specified library L_i. Having received this input, the Configurator and functional testing module initially extracts from the code and tests repository (a) the source code SC of the software to be built, and (b) the set of functional tests FT that need to be performed to verify that the software operates successfully on the target platform. Subsequently, it traverses its input configuration list in ascending order of risk estimation, and processes each candidate configuration CC as follows:

1. it applies all transformations T_i^j included in the candidate configuration CC on the software source code SC, producing the configuration-specific transformed source code SC_{CC}.

2. it arranges for executing all functional tests FT on the transformed source code SC_{CC}. This is accomplished using the `test` target of the relevant software project management tool, e.g. `mvn test` for Apache Maven or `make test` for the “make" tool.

3. if all functional tests are successful, then the final executable is built based on SC_{CC}, using the `build` target of the software project management tool; the `install` target may be also used to perform automatic deployment.

4. if some of the functional tests for some configuration fail, then the configuration is dropped and the next one is extracted for processing.

We note here that the step of performing functional tests is required to ensure the correct behaviour of the application in the target environment for the following reasons:

1. some replacement libraries may not be fully compatible with the originally specified ones; for instance the fast serialization library [222] states that compatibility with the built-in Java serialization library is “100%, but it may be 99%", therefore the correctness of the operation of replacement libraries needs to be verified.

2. compatibility issues may also arise between replacement library versions and external elements of the deployment platform, e.g. databases, where some library version in a configuration may be incompatible with the particular version of a database system in the target platform.

3. especially in the development phases of the LFEDB, transformations may be imperfect, and functional tests may provide feedback to the curators of the LFEDB, to amend transformation errors.

4.3.5 Monitoring Changes to the Risk Level

The discovery of new vulnerabilities, as well as the release of new versions or new libraries implementing different functionalities may alter the risk assessment of candidate libraries and/or introduce new configurations that need to be assessed and tested. To this end, the pipeline may be run periodically or when new additions that reference the libraries needed by the software are made to the LFEDB. When a new executable with a different configuration is produced, this executable will need to be deployed to the target platform. In our future work, we will investigate the use of hotswapping mechanisms for the in-place update of the deployed and running executable.

4.4 Conclusions

In this chapter, we have presented an approach for automating the task of minimizing the risk level of IoT systems that is owing to the vulnerabilities of libraries required by software artefacts. The proposed method exploits knowledge on which libraries provide equivalent implementations of the same functionalities, and when some library is utilized, the system identifies candidate libraries realizing the same functionality, automatically assesses the risk level that corresponds to each library implementation and finally selects the library exhibiting the minimum risk level to bundle into the executable software artefact. Additionally, the risk level of candidate implementations is constantly monitored for new vulnerability identifications or fixes in the implementations, triggering new risk assessments and producing new executables as appropriate. The proposed methodology can be used in IoT platform deployment to minimize the software-rooted risk level.

In our future work we will consider methods that will facilitate the creation and update of the library functional equivalence database, including the introduction of a compatibility/abstraction layer which will be used on top of functionally-equivalent libraries, in the same fashion that Hibernate [223] provides a compatibility/abstraction on top of database drivers. Methods for the more accurate estimation of the

risk level of libraries will be explored, including the matching of the vulnerabilities whose impact needs to be assessed against other vulnerabilities of known impact, either at textual description level or at code level. Finally, the use of hotswapping mechanisms for the in-place update of the deployed and running executables will be investigated.

Chapter 5

Detection of intermittent faults in software programs through identification of suspicious shared variable access patterns

Abstract

Intermittent faults are a very common problem in the software world, while difficult to be debugged. Most of the existing approaches though assume that suitable instrumentation has been provided in the program, typically in the form of assertions that dictate which program states are considered to be erroneous. In this chapter we propose a method that can be used to detect probable sources of intermittent faults within a program. Our method proposes certain points in the code, whose data interdependencies combined with their execution interweaving indicate that they could be the cause of intermittent faults. It is the responsibility of the user to accept or reject these proposals. An advantage of this method is that it removes the need for having predefined assertion points in the code, being able to detect potential sources of intermittent faults in the whole bulk of the code, with no instrumentation requirements on the side of the programmer. The proposed approach exploits information from the dynamic behavior of the program. In comparison with parser-based approaches which analyze only the program structure, our approach is immutable to language term changes and in general is not depending on any user-provided assertions or configuration.

5.1 Introduction

An intermittent fault in computer software is a malfunction of a software program that occurs at intervals, usually irregular, while the software functions normally at other times. Avizienis et al. [224] defines intermittent faults as the union of (a) *elusive permanent faults*, i.e. faults that manifest themselves conditionally, with their activation conditions depending on complex combinations of internal state and external requests, that occur rarely and can be very difficult to reproduce and (b) *transient*

faults, which includes *physical faults* (i.e. faults associated with the hardware) as well as *interaction faults*, stemming from reciprocal actions with external systems. The root causes of intermittent faults can be traced to (a) particular hardware conditions, e.g. radiation-induced transient faults are caused by alpha particles found in chip packages and atmospheric neutrons [225], (b) limit conditions (e.g. out of memory or disk storage, lost interrupts, not initialized memory, unexpected data from external sources including interactions with other systems) and (c) concurrency errors, including race conditions and scheduling decisions [226][224].

Software-rooted intermittent faults are referred to as MandelBugs [227][228]. A Mandelbug is a bug residing some location in the code, however applying test cases on the code even under seemingly exact conditions does not always lead to a failure. The reason for this non-deterministic behavior is twofold: firstly, the execution of the buggy code leads to an erroneous internal condition (e.g. a wrong variable value) which does not necessarily manifest itself as a failure immediately, but rather it may necessitate a chain between errors (*error propagation*) until the system uses elements (e.g. variable values) involved in the erroneous internal conditions in a way that influences a perceivable system behavior. And secondly, other elements of the software system, including other applications, the operating system or the hardware, may affect the behavior of a fault in a specific application. For instance, if a multi-threaded application lacks adequate synchronization mechanisms, race conditions may occur, depending on the choices made by the operating system scheduler regarding the exact time points that threads are dispatched on the CPU for execution, or preempted. Gray [226] uses the term *Heisenbugs*, to refer to software bugs that either do not appear or change their behavior when attempts are made to discover them. Typically this is owing to the fact that when programs are debugged the execution environment and conditions change [229]: optimization features are turned off; debugger programs may initialize memory contents to zero or modify the memory layout during execution; stepwise execution alters timings; statements inserted to print out variable values differentiate register values and so forth. Grottke et al. [230] identifies aging-related bugs as an interesting a sub-type of Mandelbugs: aging-related bugs are faults capable of causing degraded performance or increased failure rate because they either accumulate internal error states and/or the activation and/or error propagation of the fault is influenced by the total time the system has been running. MandelBugs and Heisenbugs are contrasted to *Bohrbugs* [226], which refers to the class of bugs that always produce a failure on retrying the operation that involves the bug; in this respect, a Bohrbug is a solid and easily detectable bug, that can be isolated by standard debugging techniques.

In the analysis presented in [230] Mandelbugs correspond to the 36.5% of the total number of the faults discovered in the on-board software for 18 JPL/NASA space missions. Carrozza et al. [228] studied an industrial mission-critical software system, in which Mandelbugs accounted for the 14.56% of the total number of faults. Cotroneo et al. [231] examine four major open source projects, and report that the percentage of Mandelbugs ranges from 7.5% (for the AXIS project) to 50.2% (for the Linux project); they also assert that in their sample, the fault densities for Bohrbugs and Mandelbugs are similar for large software projects, while for smaller projects the fault density for Bohrbugs tends to be higher and that Mandelbugs take more time

to fix than Bohrbugs. Chillarege [232] concludes that Mandelbugs predominantly affect non-functional aspects, such as reliability, availability and serviceability, while they rarely affect software functionality.

A common cause of software-rooted intermittent faults in applications, is the erroneous order of accessing shared variables in multi-threaded applications, e.g. when a write-after-write (WAW) hazard occurs, a shared variable is written by a thread while it should have first been written by another one, and so forth. The more complex the software program, the greater the likelihood of an intermittent fault to occur and the harder to locate its root cause. Many research efforts have targeted the issue of intermittent faults, and in this context a number of concurrency anti-patterns, (i.e. concurrency control mechanisms that have been proven to be ineffective and error-prone) and possible solutions have been identified, e.g. [233, 234]).

Intermittent faults can be detected both using static and dynamic debugging techniques [235]. For the detection of logical faults, in particular, in the context of dynamic approaches, the programmer typically needs to add appropriate *assert* statements expressing program-specific invariants (i.e. conditions that must always hold), which is evaluated at runtime. When the invariant is found not to hold, then a fault is flagged and the developer may use a dynamic debugger to examine the program state, trying to trace back the root cause of the error.

The method proposed in this chapter, intends to help programmers discover locations in the code that could cause intermittent faults that are owing to improper order of accessing shared variables. On top of an existing debugging and verification tool, we add mechanisms that create traces of shared variable access sequences and rules that are able to identify such improper access patterns within these traces; these patterns may be manifestations of intermittent fault presence. Then, the system is able to suggest to the developer code locations that may be the root cause of these intermittent faults. In this work, we have chosen Java Path Finder (JPF [236]; a brief overview of JPF is given in section 5.2.3) as the base debugger tool, on top of which the proposed method is built; we exploit the capabilities of JPF to extract runtime information from the executing program. Our approach is immutable to any user configuration (e.g. parser configurations), as it exploits information from the dynamic behavior of the program, which is sourced through the mechanisms provided by JPF. More specifically, JPF functionalities are used to gather all the information about possible interleavings of the accesses of the shared variables from the different threads in a tree structure, and after the tree structure is shaped, it is searched for the presence of shared variable access patterns that indicate the presence of an intermittent fault. Code locations that are involved in the suspectable shared variable accesses are then identified, and these locations are proposed to the user (i.e. the developer) for check (e.g. code review to verify whether synchronization mechanisms are used appropriately). The developer is the one who makes the final decision on whether a suggestion made by the tool should be accepted or not. Contrary to other algorithms in the literature, the proposed approach needs no instrumentation (e.g. insertion of appropriate assertions in selected code locations) to work. In this way, the whole extent of the executed code is always checked, and no additional effort on the side of the developer is required. The proposed technique can be used in conjunction with other intermittent fault detection techniques, both at hardware and

software level (e.g. [237][238][239][240]); combined application can be achieved either by the simultaneous use of individual techniques (this is directly applicable for other techniques that are hardware-based, e.g. [237][239]; for software-based techniques, an integration step will be required), or through a more loosely coupled approach where the proposed algorithm is run in parallel with other techniques and their results are combined.

In addition, in this work, we examine the complexity of the proposed intermittent fault detection algorithm, by experimentally quantifying the effect that partial order reduction techniques [241] have on to the limitation of this number of paths.

The rest of the chapter is structured as follows: section 5.2 overviews related work, including static and dynamic verification tools and elaborating on JPF, which is used in our approach. Section 5.3 introduces the proposed algorithm, while section 5.4 discusses the complexity of the algorithm. Section 5.5 explores methods for speeding up the execution of the proposed algorithm by (a) exploiting parallelism and (b) pruning the possible execution paths tree, with the latter techniques being able to also tackle the state explosion issue, which is inherent in state space-based approaches. Section 5.6 presents an experimental evaluation of the algorithm, and finally section 5.7 concludes the chapter and outlines future work.

5.2 Related work

Since reliability is a key objective in software development, numerous techniques have been proposed and employed to aid developers to localize, identify and remove faults. Some techniques examine the source code statically to identify *code smells*, i.e. characteristics that may indicate a deeper problem. Towards this direction, code smell detectors have been employed [242][243]. Similarly, software fault prediction aims to identify fault-prone software modules by using some underlying properties of the software project before the actual testing process begins [244].

Considering the dynamic behavior of the software, using test cases for unit-level [33] or integration testing was one of the first tools to verify software correctness [33]. Considering the size and complexity of modern software, methods for automatically generating comprehensive test case suites have been developed [245][246][247]. Since test case-based fault detection may miss certain faults, even under high code coverage, approaches to identifying faults that evade test-case based detection processes have also been proposed [248]. Additionally, taking into account that execution of test cases consumes time and resources, their minimization and management have been explored [249][27]. With security aspects gaining increasing attention in the past few years, specialized methods for analyzing and detecting software vulnerabilities have been developed [250].

When faults do manifest in software, either in the context of testing or while execution in production environments, testers and developers need to pinpoint the actual fault location and root cause: to this end, a number of relevant algorithms and techniques have been proposed. Besides “traditional" fault localization techniques, which include logging, assertions, breakpoints and profiling, a number of advanced fault localization techniques have been proposed, which are classified as (a) slice-based, (b) program spectrum-based, (c) statistics-based, (d) program state-based,

(e) machine learning-based, (f) data mining-based and (g) model-based techniques. Wong et al. [87] provides a survey on fault localization techniques.

Intermittent faults however, due to their nature, may evade detection from typical fault discovery tools [226], therefore specialized methods have been developed to assist developers in identifying and removing intermittent faults. In the rest of section we overview related work for intermittent fault detection. We initially survey work in the domain of static debuggers, and subsequently we examine approaches using dynamic debuggers. Finally, we give a brief introduction to JPF, the dynamic debugging tool used for the instrumentation of the proposed intermittent fault detection approach.

5.2.1 Static debuggers

Static debuggers analyze the software code without running it. Because these debuggers do not rely on tests, they can be extremely thorough. Theoretically, static debuggers can test even code paths which are rarely executed in practice [251]. Because they are based on static analysis and satisfying predefined constraints, they could fail to detect some errors. Moreover, while static debuggers can be used in unsafe languages[33] to reveal potential bugs, they cannot guarantee that the data in memory is coherent according to any high-level criteria [252].

It is very common that a static analyzer tool is used to analyze the software code and then symbolic execution with SMT (Satisfiability Modulo Theory) [253] formulas of defined constraints is used for the verification of the code [254].

Symbiosis is an example of a static debugger [255]. Symbiosis necessitates the existence of a failing scheduling, which is then analyzed to determine the root cause of the fault.

5.2.2 Dynamic debuggers

Dynamic debuggers examine the software code while it is running. The code in instrumented and all the possible paths are executed in order to detect candidate errors; the Partial Order Reduction technique ([241]) can be used to reduce the number of paths tested, by avoiding to re-examine some path that has been already examined while exploring some other branch. However, depending on the actual values assigned to input variables of the code, it is possible that some paths are not executed and thus the tools may miss certain code defects. In addition, because dynamic debuggers use information available at run time, which is harder to extract statically from the source code, dynamic debuggers can detect errors that are harder to discover when using static analysis tools [251].

The CHESS tool [256] is an example of a dynamic debugger. CHESS creates multiple versions of the debugged program, each one suitably instrumented to control the scheduling of threads. The instrumentation step generates $O(2^n)$ versions for a function with n components, however [256] reports that the execution of $O(n)$ versions (context switch at one of the components each time) is usually enough to activate a concurrency fault; this however may lead to missing Heisenbugs with

[33] An unsafe language does not ensure that primitive program operations are applied to arguments of the proper form, e.g. does not ensure that array subscripts are within the allowable range.

complex activation patterns. Furthermore, [257] reports that CHESS necessitates additional scaffolding and test code, on top of the test code that would be normally needed for unit or integration testing.

The SCURF tool [257] also follows the instrumentation approach to create particular combinations of thread interleaving. Then, each of these versions is run against a number of test cases -coupled with test oracles- for checking system functionality that need to be available, and a spectrum-based fault localization [258] is utilized to correlate detected errors with concurrently executing code blocks.

CTrigger [259] focuses on atomicity violation bugs; the fault identification process begins by profiling the software and identifying potential unserializable interleavings, while subsequently infeasible interleavings are pruned and low-probability interleavings are ranked. Afterwards, the unserializable interleaving space is explored. CTrigger also requires testing inputs and oracles.

Java Path Finder (JPF) [236], is another dynamic debugger which is used by the proposed algorithm. In the following subsection, we provide a brief introduction to JPF, to present the core functionalities exploited by the proposed algorithm. JPF can locate a number of concurrency bugs, such as deadlocks and missed signals, as well as Java-related faults e.g. unhandled exceptions and improper heap usage; in order to identify faults related to application semantics (e.g. erroneous variable values), relevant assertions should be given within the application code.

Table 5.1 summarizes the existing tools and methods, their features and capabilities and compares them to those of the proposed algorithm.

Table 5.1: Comparison of existing fault identification tools and methods

Tool-method	Scope	Capabilities	Limitations
Test cases	Unit & integration testing	Mostly detects Bohrbugs.	MandelBugs and Heisenbugs typically evade detection; coverage alone cannot guarantee a comprehensive fault detection. Manual creation of test cases is laborious and tedious, however test case generators are available.
Static debuggers	Static checking of code properties; symbolic execution can be also performed	Identification of resource leaks, security issues and code smells. Can be used in conjunction with SMT models for increased detection capabilities. With failing schedules available, faults can be localized.	Cannot capture dynamic behavior and may miss some errors; cannot guarantee data coherence in memory according to any high-level criteria. Use with SMT models necessitates definition of constraints (*program invariants*).
Chess [256]	Concurrency faults	Detects faults owing to thread interleaving	May miss Heisenbugs with complex activation patterns; necessitates additional scaffolding and test code.
SCURF [257]	Concurrency faults	Detects faults owing to thread interleaving and inadequate atomicity guarantees.	Necessitates pre-crafted test cases and test oracles; errors not foreseen in these cases may be missed.
CTrigger [259]	Atomicity violation bugs	Locates faults owing to thread interleaving, catering for efficiency.	Necessitates pre-crafted test cases and test oracles; errors not foreseen in these cases may be missed.
JPF [236]	Concurrency faults, including deadlocks and atomicity violations; generic faults.	Powerful detection engine, extendable via the listener mechanism.	Needs programmer-provider assertions to detect errors related to high-level data coherence.
Proposed algorithm	Enhances JPF with detection of erroneous/-suspect shared variable access patterns.	Captures all errors detected by JPF and errors related to high-level data coherence, without the need to pre-define assertions, test cases or oracles.	May flag false positives.

5.2.3 JPF - A brief overview

Java Path Finder (JPF) is an open-source software verification system, initially developed by NASA, that performs model checking for Java programs. While test case-based software checks only some of the potential program executions and may

thus miss errors, model checking automatically combines the behavior of state machines with a specification, which corresponds to the properties that the system should satisfy [236, 260, 261]. In more detail, the model checker accepts as input the state machine (FSM) of the program and the specification, and exhaustively explores all executions in a systematic way, flagging executions where the specification is found not to hold [262, 263]. The JPF code is available at [264].

While the systematic generation of all potential execution paths covers the whole search space of program states, handling millions of combinations which are hard to be modeled by manually crafted test cases [262], and thus expose all errors, this approach entails excessive computation cost, which renders it infeasible [262]. Two techniques can be used here to reduce computation costs, namely backtracking and state matching.

- *Backtracking* is a technique that allows the restoration of previous execution states, to examine if there are unexplored choices left. For instance, if JPF reaches a program end state, it can walk backwards to find different possible scheduling sequences that have not been explored yet. While this theoretically can be achieved by re-executing the program from the beginning, and arranging that a different scheduling sequence is adopted in each execution, backtracking is a much more efficient mechanism if state storage is optimized.

- *State Matching* is another key mechanism to avoid unnecessary work. The *execution state* of a program mainly consists of the heap and thread-stack snapshots. While JPF executes, it checks for every new state, whether an identical one has already been explored (c.f. Fig. 5.1); in this case, there is no use to explore again from that state onwards. When state matching occurs, JPF backtracks to the nearest non-explored non-deterministic choice.

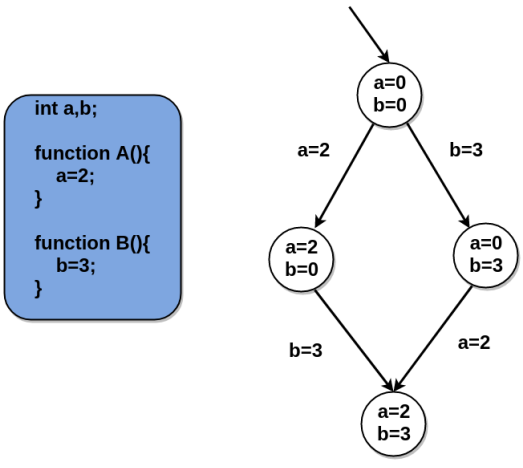

Figure 5.1: State matching: both execution paths lead to the same state (state 3).

Since concurrent actions can be executed in any arbitrary order, considering all possible interleavings of concurrent actions can lead to a very large state space. It

can be shown that the number of states increases exponentially with the number of threads [265]. JPF uses a technique called *Partial Order Reduction* (POR [266]), which basically identifies statements whose order of interleaving does not affect in any way the overall program execution, and groups them into a single state transition, reducing thus drastically the number of states that must be maintained. For instance, the instructions of threads *T1* and *T2* in Fig. 5.2 can be interleaved in any of the six ways listed in the same figure, however to verify program correctness it suffices to explore the two paths highlighted in Fig. 5.3 [267, 266].

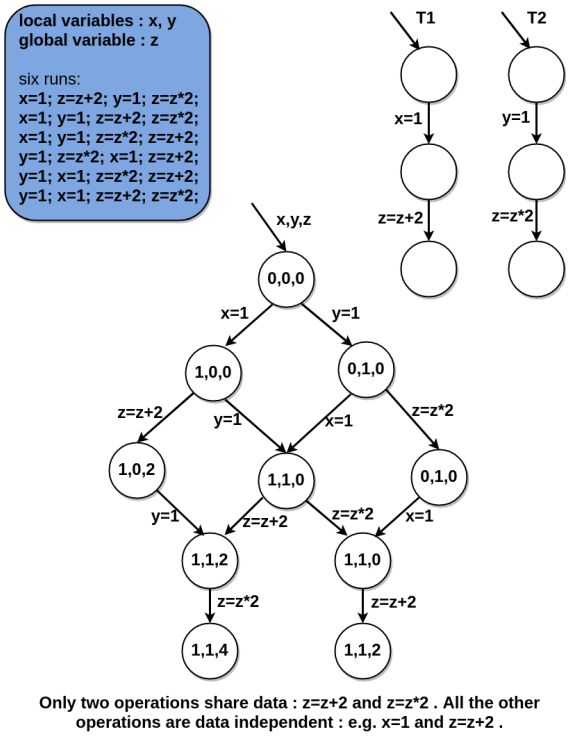

Figure 5.2: POR - Partial Order Reduction Example

JPF uses a customizable Virtual Machine that supports various features related to model checking, including state storage and state matching. Actually, JPF is a virtual machine (VM) running on top of the Java Virtual Machine (JVM) and controlling its operation. The core JPF model supports checks for generic properties, such as absence of unhandled exceptions, deadlocks, and race conditions.

Listeners are perhaps the most important extension mechanism of JPF. They provide a way to observe, interact with and extend JPF execution through code provided in the form of custom classes. Listeners are dynamically configured at runtime, and therefore they do not require any modification to the JPF core. Listeners are executed at the same authorization level as JPF, so no limitations are imposed to their functionality (c.f. Fig. 5.4). In our approach, we use one listener that observes shared variable access by threads, and logs these accesses for further analysis.

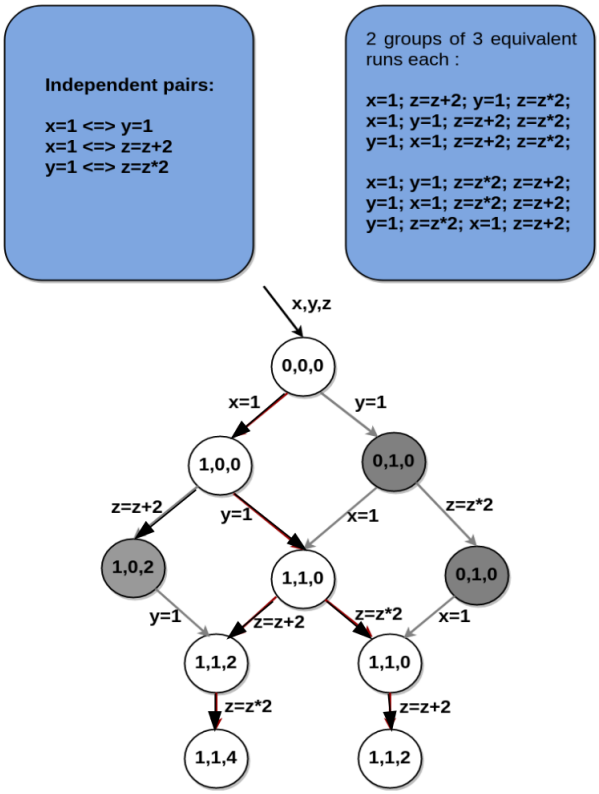

Figure 5.3: POR - Partial Order Reduction Example

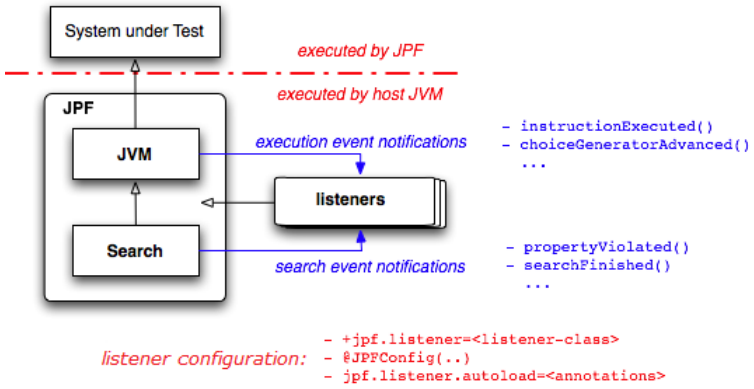

Figure 5.4: JPF listeners

5.3 The Proposed Intermittent Fault Detection Algorithm

The method proposed in this chapter comprises three parts: The first one encompasses the development of a rule base for the detection of shared variable access patterns that may indicate sources of intermittent faults. The second one is about the generation of complete execution traces for the target program, to record all possible shared variable access patterns. In this part, JPF, augmented

with additional logging listeners, is used to implement the generation of the program traces. The third one comprises the application of the rule base developed in part 1 on the traces generated during step 2, in order to detect possible sources of intermittent faults within the program. These points are proposed to the user for review and, if appropriate, application of the necessary corrections.

The first phase (rule base development) need not be performed for each program. Instead, a generic rule base can be developed once and be subsequently applied to all target programs. It is possible that derivatives of the generic rule base are created to match the requirements of specific program classes, e.g. programs with different isolation levels. The basic rules that we use in this chapter, are :

1. sequences of operations of the form $read_{T1}(X)$, $write_{T2}(X)$, $write_{T1}(X)$, corresponding to *write-after-write* hazards (the notation $read_T(X)$ denotes that thread T reads variable X; and similarly for $write_T(X)$)

2. sequences of operation of the form $read_{T1}(X)$, $write_{T2}(X)$, $read_{T1}(X)$, corresponding to the *read-after-write* hazard

When either of these patterns is detected in the program traces, then the corresponding code *may* be the cause of intermittent faults. In case of the write-after-write hazard rule, we can observe that this order is not equivalent to any serializable order: if T1 were scheduled before T2, then the value finally stored in X would be the one written by T2 instead of the value written by T1, which is finally stored by the schedule above (this is known as the lost update problem [268]); and if T2 were scheduled before T1, T1 would have read the value of X stored by T2 instead of the value previously stored for X (recall that T1 may have used the value read for X to compute a new value for X, so reading a different value leads to erroneous computation). In case of the *read-after-write* hazard rule, before we read our shared variable X for the second time on thread T1, it is modified on T2 with a write operation, which makes the value of X on thread T1 to have a new value (without having saved the product of the previous X read value or simply using different values of X in different parts of the computation in T1; the latter is also known as the non-repeatable read problem [269]), which may be a potential cause of some intermittent fault.

In the second phase, we create a tree structure, where we store the data of the states of the JPF run. Each state can be a combination of data from the different threads that are accessed concurrently. Typical data that we store during this state are: the variable name, the class name, whether the access is a read or write one, the thread id, the method name, synchronization info, the package name, the value of the variable, if there is a monitor enter or exit operation, lock info, the line of the code that is executed and the source line, etc. When the JPF run is done, the detailed trace about the accesses of the state variables is available for analysis. Since each state may have one or more previous states (recall that the state matching mechanism specifically searches for cases where multiple execution paths have led to the same state) and may lead to multiple subsequent states, the trace effectively forms a directed acyclic graph (DAG).

In the third phase (processing phase), we apply our rule base on this structure in order to detect the points in the code that match our rules. Multiple rule bases or

even ad-hoc rules could be applied during this phase. The JPF should be executed only once while the tree structure can be stored and reused for the application of all user rules.

As an example of applying the proposed algorithm, consider the case that the rule base contains the two rules listed above, and that the code illustrated in Listing 5.1 is checked for the presence of intermittent faults. Before the execution of line (3) of this code, the value of the ***filled*** variable should always be smaller than ***MAXNUM***, a condition that is checked by the condition at line (1) and the code associated with it. However, in the context of concurrent executions of the ***put*** method, it is possible that two distinct threads detect that the value of the ***filled*** variable is equal to ***MAXNUM-1***, and subsequently each one increases the value of the variable by 1, therefore violating the invariant $filled \leq MAXNUM$; this is owing to the premature lock release, occurring at line (2).

Listing 5.1: A simple code example, which produces faults intermittently

```
   private final static Lock l = new ReentrantLock();
   private static int filled = 0;
   private static ArrayList queue = new ArrayList();
   private static final int MAXNUM = 2;

   public void put(Object elem) {

       l.lock();
(1)    if (filled < MAXNUM) {
           //other code
(2)        l.unlock();
       } else {
           l.unlock();
           return;
       }
       l.lock();
       // assert (filled < MAXNUM);
(3)    filled++;
       queue.add(elem);
       l.unlock();
       return ;

   }

   public Object get() {
       Object elem = null;
       l.lock();

       if (filled > 0) {
(4)        filled--;
           elem = queue.remove(0);
       }
       l.unlock();
```

```
        return elem;
    }
```

Fig. 5.5 demonstrates the different access interleavings that may occur when the code of the *put* method is executed concurrently by two threads, *T1* and *T2*, assuming that the condition at line (1) evaluates to *true* for both threads. We can notice that six distinct interleavings are possible, out of which four (the 1st, 2nd, 4th and 5th branches of the tree) entail the appearance of the non-repeatable read problem. For instance, in the second branch of the tree, the following shared variable accesses will be performed: $read_{T1}$*(filled)*, $read_{T2}$*(filled)*, $read_{T1}$*(filled)*, $write_{T1}$*(filled)*, $read_{T2}$*(filled)*, $write_{T2}$*(filled)*, with the first two reads corresponding to the checking of condition at line (1), and subsequently each read/write pair corresponding to the variable increment at line (3). In this sequence, we can observe that a *read-after-write* hazard occurs, since a write operation on variable *filled* is performed by thread *T1* between the two read operations performed on the same variable by thread *T2*, therefore the read performed by *T2* is non-repeatable.

Fig. 5.6 shows the respective states of the *filled* variable, again assuming that the condition at line (1) evaluates to *true* for both threads. Notably, when the *filled* variable is less than *MAXNUM-1* all interleavings lead to a correct state, increasing the *filled* variable by two. However, if *filled == MAXNUM-1*, the execution of the branches entailing the *read-after-write* hazard leads to an incorrect state, where the *filled* variable is set to *MAXNUM+1*. The proposed algorithm can thus identify code that is bound to cause intermittent faults, without any knowledge about the correctness of the states.

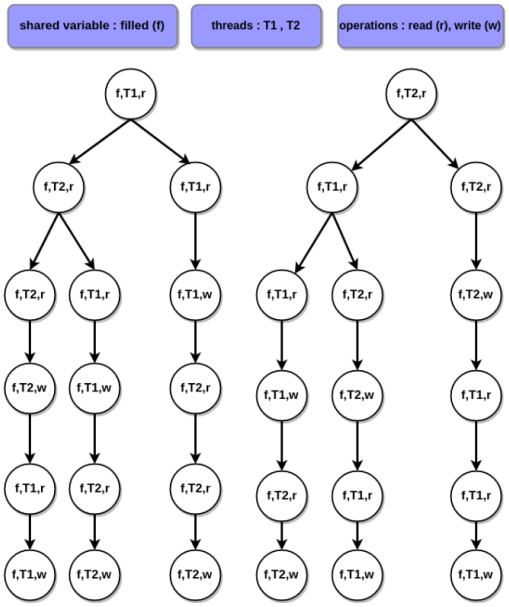

Figure 5.5: The possible access interleavings of the shared variable "filled" when two put methods of two different threads are executed concurrently.

Using JPF to generate our state tree structure, has the advantage that the user

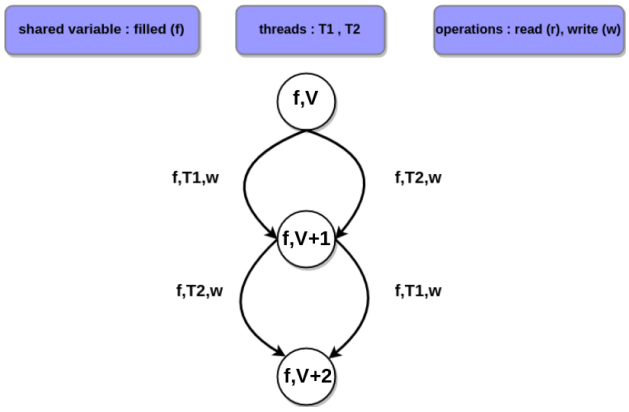

Figure 5.6: The states of the shared variable "filled" when two put methods of two different threads are executed concurrently.

does not need to give any a-priori information about the code (shared variables, atomic blocks, etc.), while JPF is not sensitive in possible code structure changes as a static analyzer could potentially be.

An implementation of the algorithm is available in open source at `https://github.com/pansot2/JPF`.

5.4 Complexity analysis

When a multithreaded program with k threads executes, at each time point the scheduler may pick any of the threads that are not in a suspended state to execute, thus having a maximum of k alternative choices. Since we focus on operations that access shared variables (because inappropriate shared variable access patterns are a major cause of intermittent faults), if we consider that each thread t_i performs n_i shared variable accesses, then the corresponding states of a multithreaded program can be arranged in a tree whose rank is equal to the number of threads k and its depth is equal to:

$$depth = \sum_{i=1}^{k} n_i \tag{5.1}$$

The root of the tree corresponds to the initial state of the program, while an edge denotes a transition from a state to a subsequent one, through the execution of an instruction that accesses a shared variable, with the instruction belonging to some thread t_i (c.f. Fig. 5.5). Sibling tree states correspond to different scheduling decisions. The total number of paths in the tree is equal to the number of ways to interleave k ordered sequences.

In order to compute the number of paths, we consider that the instructions in each thread t_i $(1 \leq i \leq k)$ are essentially ordered lists, and we want to interleave the elements of these lists while preserving the order of the elements in each ordered list.

According to equation (5.1), there will be $n_1 + n_2 + \cdots + n_k$ places that we must fill (one place for each level of the tree). We can proceed by first assigning the elements of the first list, corresponding to the instructions of the first thread, to

places. Therefore, we select n_1 out of the available $n_1 + n_2 + \cdots + n_k$ places, and we assign to the selected n_1 places the instructions of the first thread, preserving their order. The number of possible alternatives is $\binom{n_1+n_2+\cdots+n_k}{n_1}$.

Next, we choose n_2 of the remaining places that will accommodate members of the second list, which correspond to the instructions of the second thread. Out of the total number of $n_1 + n_2 + \cdots + n_k$ places in the list, n_1 are now occupied by elements of the first list, therefore the number of available places in the list is $n_2 + \cdots + n_k$. Consequently, the number of possible alternatives is $\binom{n_2+\cdots+n_k}{n_2}$. Working in the same way with the remaining ordered lists, when placing the elements of the k^{th} list there are n_k elements to be placed in n_k positions, therefore there exist $\binom{n_k}{n_k}$ alternatives.

Combining all the above, the mathematical formula that calculates the number of ways to interleave k ordered sequences is:

$$\prod_{i=1}^{k} \binom{\sum_{j=i}^{k} n_j}{n_i} \tag{5.2}$$

In each state of the execution tree, we store information about the current accesses of the shared variables, synchronization info, etc. for all active threads. If a thread progresses, by accessing a shared variable, recording changes in the synchronization info, etc., then a new tree node is created as a child of the previous state (c.f. Fig. 5.7).

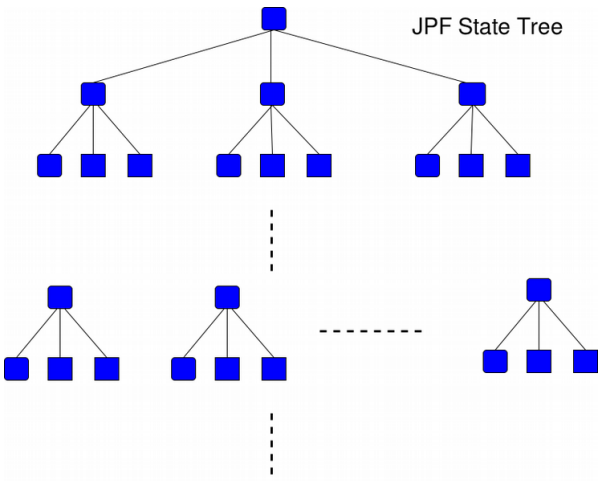

Figure 5.7: JPF State Tree - Rank of the tree: The maximum number of threads that can run in parallel. Depth of the tree: The sum of the accesses of all shared memory variables for all the threads.

In our work, execution path traversal is instrumented via the JPF model checker, which is a so-called explicit-state model checker, since it enumerates all visited states, and therefore suffers from the state-explosion problem inherent in analyzing large programs [270], while the number of paths to be examined also increases rapidly, with the number of threads and instructions per thread (c.f. equation 5.2. However, JPF employs a number of techniques including POR (Partial Order Reduction), state matching, and branch coverage [270] to reduce the number of states and the number of paths that will be examined. Using these techniques, JPF can scale up to analyzing programs up to 100,000 lines of code [271].

Insofar, there has not been any theoretical analysis of the effect that POR and state matching have on the complexity of the algorithms that explore the search space of possible execution paths. This is due to the fact that the final effect is highly dependent on the specific instruction placement for each program (which affects the number of cases that POR can be applied), existence and location of lock/unlock instructions (which may limits the actual choices available to the scheduler at each step), as well as volatility of external inputs, which is a determinant factor for the number of cases that states will be actually matched. Further theoretical analysis of this aspect is part of our future work.

The proposed algorithm dictates that shared variable accesses that are performed along an execution path are recorded, and access sequences are scanned for occurrences of the two concurrency hazards, i.e.:

1. access patterns of the form $read_{T1}(X)$, $write_{T2}(X)$, $write_{T1}(X)$, corresponding to *write-after-write* hazards

2. access patterns of the form $read_{T1}(X)$, $write_{T2}(X)$, $read_{T1}(X)$, corresponding to the *read-after-write* hazard

The complexity of recording shared variable access sequences within an execution path is $O(n)$, where n is the number of shared variable access sequences occurring within the execution path. Once the access sequence is recorded, the next task is to determine whether this sequence contains any of the access patterns (1) and (2) above. In the following, we discuss the matching procedure, considering initially the first form of access pattern for a specific thread, and subsequently we generalize for the second form of access patterns and all threads of a program.

When searching for access patterns of the form (1) above, it is not necessary that the instructions are found in strict sequence. The following types of instructions may intervene between the first instruction of the pattern ($read_{T1}(X)$) and the last one ($write_{T3}(X)$):

- accesses to other shared variables, either by the same or by other threads, e.g. $read_{T1}(Y)$ and $write_{T2}(Y)$;

- accesses to the same shared variable by threads other than T1, e.g. $read_{T3}(X)$ and $write_{T2}(X)$;

If such instructions occur, then still the hazard can be flagged, since they have no effect on the semantics of the pattern. Additionally, we can note the following:

- if a $read_{T1}(X)$ instruction occurs between the first and the second instruction of the pattern (i.e. we have an access sequence $read_{T1}(X)$, $\boldsymbol{read_{T1}(X)}$, $write_{T2}(X)$, $write_{T1}(X)$), then the hazard still exists, and it actually maps to the instructions 2-4 of the extended access pattern (i.e. the first $read_{T1}(X)$ is not a part of the hazard; the hazard occurs later on).

- similarly, if a $write_{T1}(X)$ instruction occurs between the second and the third instruction of the pattern (i.e. we have an access sequence $read_{T1}(X)$, $write_{T2}(X)$,

write$_{T1}$***(X)***, *write*$_{T1}$*(X)*), then the hazard still exists, and it actually maps to the instructions 1-3 of the extended access patterns (i.e. the last *write*$_{T1}$*(X)* is not a part of the hazard; the hazard has already occurred upon the execution of the third instruction of the extended access pattern).

- if a *read*$_{T1}$*(X)* occurs between the second and the third instruction of the pattern (i.e. we have an access sequence *read*$_{T1}$*(X), write*$_{T2}$*(X),* ***read***$_{T1}$***(X)***, *write*$_{T1}$*(X)*), then the hazard *does not occur*, since the computation of the value written by the fourth instruction has been performed based on a "fresh" copy of variable X (i.e. a copy obtained after thread *T2* has written a new value [268]).

- finally, if a *write*$_{T1}$*(X)* instruction occurs between the first and the second instruction of the pattern (i.e. we have an access sequence *read*$_{T1}$*(X),* ***write***$_{T1}$***(X)***, *write*$_{T2}$*(X), write*$_{T1}$*(X)*) then the hazard *may occur*, since the value stored by the fourth instruction *may* be dependent on the value read by thread *T1* during the execution of the first instruction of the extended access sequence.

Considering all the above, the target access pattern may be formulated as a regular expression of the form:
read$_{T1}$(X) (all except read$_{T1}$(X))* write$_{T2}$(X) (all except read$_{T1}$(X))* write$_{T1}$(X)
where the notation *all except read*$_{T1}$*(X)* means any either a read or write on any shared variable, by any thread except for a read access by thread $T1$; note here that since both the number of shared variables and the number of threads are finite, the notation *all except read*$_{T1}$*(X)* corresponds to a finite set of elements, whose cardinality is *(#threads * #shared variables - 1)*. Furthermore, the star operator denotes "zero or more occurrences of the preceding element".

In order to match a regular expression against an element sequence, a deterministic finite state automaton can be used [272]; Fig. 5.8 depicts the deterministic finite state automaton which matches the regular expression described above. The finite state automaton performs the match in linear time, performing one state transition for each input symbol. Therefore, matching a single instance of a rule, pertaining to a specific thread and a specific shared variable, can be performed in linear time.

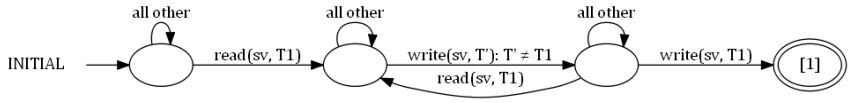

Figure 5.8: Deterministic finite state automaton for matching the access pattern

In the detection phase multiple instances of rules must be matched against the shared variable access traces of each execution path; effectively, each of the two rules corresponding to the *write-after-write* and the *read-after-write* hazard must be specialized for each thread and each shared variable. Therefore a maximum of (2 * #threads * #shared variables) rules must be matched; the number may be lower if some thread T does not read or write some shared variable V, in which case the respective rule instances specialized for thread T and shared variable V need not be considered.

Matching of all rule instances can be performed by a single reading of the shared variable access trace, by combining the deterministic finite state automata into a single deterministic finite state automaton capable of recognizing all suspect shared variable access patterns. The procedure for building the automaton is described in [272] and summarized in the following. Let $M_i = (K_i, \Sigma, s_i, F_i, \delta_i)$ be the automaton that realizes the match of rule instance R_i, where K_i is the set of states of the automaton, Σ is the alphabet (read and write operations on shared variables by threads), s_i is the start state of the automaton, F_i is the set of final states and δ_i the transition function for M_i. A new non-deterministic automaton $M_{nd} = (K_{nd}, \Sigma, s_{nd}, F_{nd}, \delta_{nd})$ is constructed where:

$$K_{nd} = S_{nd} \bigcup (\bigcup_i K_i) \tag{5.3}$$

$$s_{nd} = S_{nd} \tag{5.4}$$

$$F_{nd} = \bigcup_i F_i \tag{5.5}$$

$$\delta_{nd} = (\bigcup_i \delta_i) \bigcup (\bigcup_i \{S_{nd} \xrightarrow{\epsilon}\}) \tag{5.6}$$

Effectively, a new start state S_{nd} is introduced which is non-deterministically linked to all start states of the individual automata under an ϵ-transition (i.e. a transition that occurs with no input), and all final states of the individual automata are considered as final states in the merged automaton. The non-deterministic automaton is depicted in Fig. 5.9. Finally, the non-deterministic automaton is converted to a deterministic one, using the algorithm described in [272].

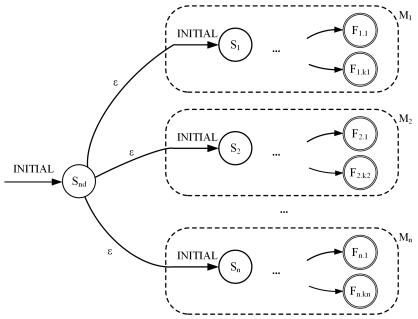

Figure 5.9: The non-deterministic automaton.

Since an execution path typically includes other instructions besides accesses to shared variables (e.g. accesses to local variables, computations, etc.), the overall complexity of shared variable access recording and matching is inferior to that of the execution of the path. Additionally, note here that in the context of the execution performed by JPF as part of the explicit state-model checking, some operations such as state matching are expensive ones, needing to examine a number of data elements (e.g. values of state variables), as contrasted to the shared variable access recording and rule matching operations introduced by the algorithm, where each shared variable access is recorded or processed in the merged deterministic finite state automaton in a $O(1)$ operation. Therefore, the overall complexity of the suspicious shared variable access pattern detection procedure is dominated by the complexity of

the execution of the different execution paths, which is instrumented by JPF which -as noted above- can satisfactorily handle programs of the magnitude of 100,000 lines of code.

5.5 Optimizing Intermittent Fault Identification

While the presented algorithm leverages the intermittent error detection potential, it introduces additional overheads. In this section, we examine methods for limiting overheads and increasing the efficiency of intermittent fault detection.

5.5.1 Separate analysis of independent thread partitions

Our method targets the identification of access patterns on shared variables which may lead to errors; to test whether such patterns may appear, all possible execution paths are examined. However, when two threads do not access any variable in common, it is not necessary to test all possible interleavings of these threads' execution, since obviously no "suspect" variable access patterns may be identified among these threads.

Generalizing, we can partition the threads in the program in subsets TS_1, TS_2, ..., TS_n where:

- $TS_i \cap TS_j = \emptyset$, $\forall$ i, j: i $\neq$ j
- $\bigcup_i TS_i = T$, where T is the set of all program threads
- $SVA(TS_i) \cap SVA(TS_j) = \emptyset$, $\forall$ i,j: i $\neq$ j, where $SVA(TS_j)$ is the set of all shared variables accessed by any thread in TS_j

In order to exploit this aspect towards the optimization of the intermittent fault identification process, we override the default choice generation and backtracking behavior of JPF, to allow the user to specify the threads whose execution will be monitored in a particular execution. At implementation level, this is realized by overriding the `stateAdvanced` method of the listener, which controls what happens when a new state is generated. In more detail, the user defines in the configuration file the threads that contain related shared variables, which form a thread subset, and the rest of the threads being partitioned into trivial, single-thread subsets which are ignored in order not to produce any alternative choices; only a single choice is considered for these threads. Effectively we have a model involving some "interacting threads" and some "independent threads". This is accomplished using the following configuration parameter:
vm.watched.threads = the threads that should trigger alternative choice generations in JPF

The code in the listener that handles this functionality is illustrated in the following Algorithm (Listing 5.2):

Listing 5.2: Process followed by the listener used for choice generation only for a subset of the threads of the software program

```
1. Get information about the watched threads defined with the
   configuration parameter vm.watched.threads
2. Get information about the threads that the watched threads depend on;
   this is defined via the configuration parameter
   vm.watched.threads.seqdeps
3. In the stateAdvanced overriden method, ignore the states that are not
   caused by executing instructions of the watched threads or the
   threads they depend on. This is accomplished by invoking the
   search.getVM().ignoreState() method.
```

As it can be noticed in the process above, the *ignoreState()* method that JPF provides is used in order to ignore the states related to a thread change that are not included at the *vm.watched.threads* list in the configuration. There is no need to make alternative choices for the scheduling of threads that are not included in the *vm.watched.threads* variable and have no watched thread depending on them, as these, in general, do not influence the subset of threads for which the intermittent fault analysis is conducted.

In order to comprehensively analyze the program for existence of intermittent faults, the intermittent fault detection procedure should be run for each thread subset TS_i. If the number of states examined when analyzing thread subset TS_i is equal to $numStates(TS_i)$, then the analysis of all subsets entails the examination of $\sum_i numStates(TS_i)$ states. Contrary, if a combined analysis of all threads is employed (i.e. the thread "independence property" is not exploited), then the analysis will entail the examination of $\prod_i numStates(TS_i)$ states, under the assumption that no dependent threads exist for each thread subset. It is clear that the thread partitioning scheme introduces significant performance gains.

The formulation of independent thread partitions that are separately examined for suspect shared variable access patterns contributes to the reduction of the state space that need to be examined, alleviating thus the issue of state explosion. The gains regarding the aspect of state space size reduction are quantified through the experiments presented in subsection 5.6.3.

At this stage of development, we have delegated the responsibility of partitioning threads to thread subsets to the user; in our future work, we will consider automatic or semi-automated ways to determine independent thread subsets.

5.5.2 Pruning state subtrees of specific nodes

In this section we explore the potential to optimize the intermittent fault analysis time, by reducing the nodes of the JPF tree for different ranks and depths. This method can be used for software programs employing a Boss-workers model [273] (or the dispatcher-worker model, as listed in [274]), where the different tasks are distributed to workers which use code/libraries that are independent (in terms of shared variables) from the rest of the program. A typical case of this example is the web server process service loop, where requests are accepted by the main thread and then their execution is delegated to worker threads, with worker threads being

totally independent and not accessing any shared variables. In this case, we could avoid the expansion of alternatives for worker thread states, since -by virtue of their independence- are not bound to be the source of intermittent faults (cf. Fig. 5.10).

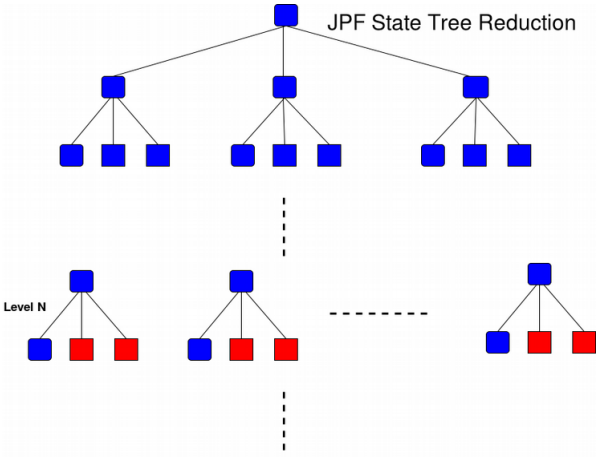

Figure 5.10: JPF State Tree Reduction : Pruning subtrees of nodes at Level N by allowing execution of the first child of each node only.

Pruning state subtrees can be configured by specifying the depth of the state tree at which pruning will occur and the order of the child nodes at this level to be allowed: at the present state of development, pruning is regulated via the properties listed in table 5.2. At runtime, when the listener detects that a subtree should be pruned it executes the statement `ti.breakTransition(true);`, which breaks the current transition and forces an end state.

Property name	Description
vm.parallel.allow.depth	the level of the nodes where the reduction will be applied (single value or a range)
vm.parallel.allow.child	the order of the child node(s) at the specified level that will be allowed to continue (e.g. the value “2" designates that the 2nd child will be allowed to continue, while execution of other children will be inhibited)

Table 5.2: Parameters regulating the pruning of state subtrees

Pruning the state subtrees of specific nodes contributes to the alleviation of the state explosion problem, since the state space that is explored within the program execution is reduced. The gains regarding the aspect of state space size reduction are quantified through the experiments presented in subsection 5.6.3.

5.5.3 Exploiting processing power in share-nothing architectures

The proposed method is orchestrated on top of JPF, and can thus benefit from JPF's potential to run efficiently on shared memory architectures, exploiting multiple execution cores for accelerating the state space search procedure [275]. To further scale parallelism potential and take advantage of share-nothing architectures, the user could designate specific JPF paths whose exploration would be assigned to a different machine. More specifically:

- the listener examines all transitions
- when a path that is designated to be transferred to another machine is reached, the state space is serialized and transferred over the network to the destination machine
- at the local machine further exploration is inhibited by means of the `ti.breakTransition(true,` statement, which breaks the current transition and forces an end state.
- On the remote machine, the state is deserialized, and execution resumes from the point that it was suspended; in this case, the listener does not issue the `ti.breakTransition(true);` statement, allowing the exploration of the path.

The state serialization, transfer and execution resumption mechanisms are currently under implementation.

5.6 Experimental evaluation

In this section, we present the experiments conducted to:

1. validate the proposed algorithm in terms of its fault detection potential,
2. experimentally assess the complexity of the algorithm and quantify the overhead introduced over the "plain JPF" software validation and
3. assess and quantify the gains reaped from applying the optimization methods presented in section 5.5.

5.6.1 Algorithm validation

Small-scale validation

Initially, when we ran the non-extended version of JPF against the code illustrated in listing 5.1 using three parallel threads. The validation succeeds without identifying any potential error sources, exhibiting thus a false negative.

Then we run the proposed algorithm to generate data access traces, applying the rules *read(s,T1)-write(s,T2)-write(s,T1)* and *read(s,T1)-write(s,T2)-read(s,T1)*,

which can identify data access patterns that are potentially erroneous. After the processing of the tree, which has been generated via our listener, the following instruction interleavings are identified as possible intermittent fault causes:

1. $t_1(1) - t_2(3) - t_1(3)$
2. $t_1(1) - t_2(4) - t_1(3)$
3. $t_1(1) - t_2(4) - t_1(4)$

where $t_i(j)$ denotes the execution of instruction j (cf. code example in Section 3) by thread i.

The user is invited to review the relevant code and accept or reject those proposals. In the above case, the first case flagged by the algorithm is a source of intermittent faults, since it may lead to the violation of the $filled < MAXNUM$ program invariant, as we would like in every case before executing line (3) to have a shared variable which is smaller than $MAXNUM$ ($filled < MAXNUM$). The two last sequences do not generate an intermittent fault in our case, however it is worth noting that when the instructions `l.unlock();` (line (2) in listing 5.1) as well as the corresponding *l.lock*() instruction immediately preceding line (3) in the same listing, effectively thus removing the atomicity violation which is the root cause of the error flagged in the first case, the second error flagging are removed and only the third one is reported.

Our approach is able to identify previously missed intermittent faults. On the other hand, it introduces some false positives. An annotation-based approach could be used to inhibit the reporting of specific patterns that have been validated not to cause intermittent faults.

Validation in real-world scale

To validate our approach in a real-world scale, we conducted experiments using the multithreaded Java webseever available in [276], which extensively uses multithreading (using a thread pool of configurable size) and shared variables. We initially ran the non-extended version of JPF against the simulation code given in GitHub [277], and no errors were flagged. The code was also checked by the proposed algorithm, and no errors were flagged either.

Subsequently, we followed a fault injection approach [231], to inject faults within the code and test whether these faults are detected by (a) the non-extended versions of the JPF and (b) the proposed algorithm (i.e. JPF extended with our listener and the potentially erroneous access pattern detection). The results of the tests are summarized in table 5.3.

Effectively, the proposed algorithm was able to detect all faults detected by JPF (which underpins the proposed algorithm), plus errors related to erroneous access patterns, which were missed by JPF. Therefore, the proposed algorithm offers more comprehensive error detection, at the expense of flagging a limited number of false positives and a performance overhead, which has been quantified to be up to 10.7%, as discussed in subsection 5.6.2. Recall here that the potential of the proposed algorithm to detect erroneous shared variable access patterns is advantageous over

Fault type	JPF	Proposed algorithm
Deadlock	Yes	Yes
Unhandled exception	Yes	Yes
Race conditions	Yes	Yes
Application-specific assertions	Yes	Yes
Erroneous shared variable access patterns as in listing 5.1	No	Yes; for the injected erroneous access pattern faults, one related false positive was also raised

Table 5.3: Detection of injected faults by JPF and the proposed algorithm

the detection of errors based on application-specific assertions, in that (a) the former does not necessitate any instrumentation by the programmer (i.e. insertion of *assert* statements), while the latter does and (b) assertion-based error detection is limited to detecting errors at the locations that assertions have been inserted and related to the conditions within the assertions, whereas erroneous shared variable access pattern detection can identify errors at any location and under any condition.

An instance of the code of [276] with injected faults is available at [278] [34]

5.6.2 Complexity Assessment Experiments

In this subsection, we report on the experiments conducted to gain insight regarding the size of the state space and the execution time needed under different thread mixtures, and present the obtained metrics. To promote example clarity, we initially present a complexity analysis on the code depicted in listing 5.3, while subsequently we present our complexity analysis findings on our real world-scale example of the multithreaded Java websever [276]. All the experiments reported in this subsection, as well as in the following one, have been performed on a PowerEdge M910 blade server, with 256 GBytes of physical memory and four 8-core E7-4830 Intel Xeon processors. The Java environment had been configured to use up to 40 GBytes of memory.

The example in listing 5.3 entails instructions belonging to three threads, namely *T1*, *T2* and *T3*. Threads *T1* and *T2* access two shared variables *A* and *B*, therefore the interleaving of their instructions can be the root cause of intermittent fault occurrence. On the contrary, thread T3 accesses only local variables, and consequently no intermittent faults can occur due to the instructions of this thread.

Listing 5.3: Thread code

```
shared int A, B;

T1:
  B = 2;
```

[34] At different phases of the test, different faults were injected; [278] contains a specific set of faults.

```
  A = B + 1;

T2:
  B = 0;
  B = B + 2;
  A = B + 1;

T3:
  local int c, d;
  d = 0;
  d = d + 2;
  c = d + 1;
```

In terms of shared variable read and write operations, threads *T1* and *T2* can be written as:

Listing 5.4: Thread read and write operations

```
T1: w(B), r(B), w(A)
T2: w(B), r(B), w(B), r(B), w(A)
T3: -
```

Some possible execution schedules of threads *T1* and *T2* are presented in the following list. For conciseness, we have only included those execution schedules which end with the last instruction of thread *T1*; inclusion of cases that end with the last instruction of *T2* is done in an identical fashion. The instructions of *T1* (i.e. instructions of thread *T1* for which at least one instruction of *T2* intervenes between them and the last instruction of *T1*)are denoted using boldface, to promote readability.

For the creation of the execution schedules presented in the following list, we assume a simple processor addressing mode, where each instruction can fetch access only one memory location (corresponding to a variable), fetching its contents to a register or storing register contents to it. This is in-line with the Java model, where instructions operate on operands individually stored on the operand stack, and results are then copied to variables[35]. In this sense, read and write operations executed in the context of the same instruction (e.g. *r(B)T1*, *w(A)T1* corresponding to the instruction *A = B + 1;* are separable, in the sense that thread switching can occur between these two accesses. However, in some processors it is possible to execute multiple variable accesses in a single instruction: for instance in the Pentium it is possible to map the program instruction *B = B + 2;* to a single machine-language instruction *ADD WORD PTR [ESI],0x2* (assuming that the ESI register points to the memory location of variable *B*) [279]. Notably, this can happen when Java bytecode is compiled into optimized machine instructions e.g. through the Java HotSpot VM[36]. In these cases, execution schedules in which the involved read and write operations appear separated cannot occur and therefore should not be considered; henceforth, the following list contains only execution schedules where

[35] https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html

[36] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html

the operations *r(B)T2* and *w(B)T2* realizing the *B = B + 2;* instruction of thread *T2* are adjacent.

1. w(B)T2, r(B)T2, w(B)T2, r(B)T2, w(A)T2, w(B)T1, r(B)T1, w(A)T1
2. **w(B)T1**, w(B)T2, r(B)T2, w(B)T2, r(B)T2, w(A)T2, r(B)T1, w(A)T1
3. w(B)T2, **w(B)T1**, r(B)T2, w(B)T2, r(B)T2, w(A)T2, r(B)T1, w(A)T1
4. w(B)T2, r(B)T2, w(B)T2, **w(B)T1**, r(B)T2, w(A)T2, r(B)T1, w(A)T1
5. w(B)T2, r(B)T2, w(B)T2, r(B)T2, **w(B)T1**, w(A)T2, r(B)T1, w(A)T1
6. **w(B)T1**, **r(B)T1**, w(B)T2, r(B)T2, w(B)T2, r(B)T2, w(A)T2, w(A)T1
7. **w(B)T1**, w(B)T2, **r(B)T1**, r(B)T2, w(B)T2, r(B)T2, w(A)T2, w(A)T1
8. **w(B)T1**, w(B)T2, r(B)T2, w(B)T2, **r(B)T1**, r(B)T2, w(A)T2, w(A)T1
9. **w(B)T1**, w(B)T2, r(B)T2, w(B)T2, r(B)T2, **r(B)T1**, w(A)T2, w(A)T1
10. w(B)T2, **w(B)T1**, **r(B)T1**, r(B)T2, w(B)T2, r(B)T2, w(A)T2, w(A)T1
11. w(B)T2, **w(B)T1**, r(B)T2, w(B)T2, **r(B)T1**, r(B)T2, w(A)T2, w(A)T1
12. w(B)T2, **w(B)T1**, r(B)T2, w(B)T2, r(B)T2, **r(B)T1**, w(A)T2, w(A)T1
13. w(B)T2, r(B)T2, w(B)T2, **w(B)T1**, **r(B)T1**, r(B)T2, w(A)T2, w(A)T1
14. w(B)T2, r(B)T2, w(B)T2, **w(B)T1**, r(B)T2, **r(B)T1**, w(A)T2, w(A)T1
15. w(B)T2, r(B)T2, w(B)T2, r(B)T2, **w(B)T1**, **r(B)T1**, w(A)T2, w(A)T1

Table 5.4 depicts the experimental results obtained from running the code in listing 5.3 with a varying number of instances of threads *T1*, *T2* and *T3*. The experimental results are contrasted with the theoretical maximum of possible paths, which is equal to the number of possible distinct execution schedules (c.f. Section 5.4).

Complexity Comparison Table								
	2 Threads (1xT1, 1xT2)	3 Threads (1xT1, 1xT2, 1xT3)	4 Threads (1xT1, 1xT2, 2xT3)	4 Threads (2xT1, 2xT2)	5 Threads (1xT1, 1xT2, 3xT3)	5 Threads (2xT1, 2xT2, 1xT3)	6 Threads (1xT1, 1xT2, 4xT3)	4 Threads (2xT1, 2xT2, 2xT3)
Theoretical number of possible paths	56	56	56	40360320	56	40360320	56	40360320
Experimentally determined number of paths (JPF)	13	18	63	49766	63	90627	63	154081
Practical number of final states (JPF)	2	2	2	3	2	3	2	3
Experimentally determined number of final states (JPF)	4	4	4	11	4	11	4	11
JPF end states	29	33	37	331	41	342	45	353

Table 5.4: Complexity comparison for varying number of threads

At this point, we note the following :

1. For the number of possible paths, we assume all possible execution schedules regarding instructions accessing shared variables, as calculated by equation 5.2.

2. The metric *Practical number of final states* corresponds to the number of the different program results, in terms of shared variable values.

3. The difference between the *Practical number of final states* and the *JPF end states* exists because of the additional information regarding shared data used in JPF (e.g thread shared information).

Regarding the complexity analysis experiments conducted using our real world-scale example of the multithreaded Java websever [276], table 5.5 depicts the execution statistics of the multithreaded Java websever [276] regarding the number of states, and table 5.6 depicts the runtimes measured, while varying the parameter of the execution tree depth search limit (c.f. subsection 5.5.2). In all experiments, all injected faults were flagged, while in the setting where the JPF search depth was set to 350 it was observed that the limit was not reached within the fault detection process execution, indicating that further increments to that parameter would not affect the time and resources needed. We can observe that the overhead introduced by the proposed algorithm over the non-extended version of JPF is up to 10.7%, which is deemed acceptable, considering the increased fault detection potential of the proposed algorithm.
where:

- *new states* is the number of unique states visited during the run;

new states	35,185,856
visited states	24,779,490
backtracked states	59,965,346
end states	0
instructions	499,329,9768
max memory	30,7 GB

Table 5.5: Execution statistics for the fault detection process of the multithreaded Java websever

JPF execution time			
	JPF search depth=120	JPF search depth=240	JPF search depth=350
JPF (not extended)	01:27:19	04:01:59	04:05:09
Proposed algotithm	01:34:25	04:26:15	04:31:26

Table 5.6: Fault detection process execution time for Java web server Simulation

- *visited states* is the number of states that are examined and have been revisited during the same execution;
- *backtracked states* refers to the states from which the search backtracked, so as to examine different paths;
- *end states* refers to the concluding states of the program execution, from which there are no forward transitions to try.

5.6.3 Optimization Experiments

In this subsection we report on the experiments conducted to assess the gains introduced by the optimization methods presented in section 5.5, and present our findings. As noted above, all the experiments reported in this subsection have been performed on a PowerEdge M910 blade server, with 256 GBytes of physical memory and four 8-core E7-4830 Intel Xeon processors. The Java environment had been configured to use up to 40 GBytes of memory.

Table 5.7 illustrates the performance gains obtained by isolating threads that are not bound to be involved in the occurrence of intermittent faults, regarding the code depicted in listing 5.3. In more detail, the first data column in Table 5.7 corresponds to the measurements obtained from the execution of a program whose main thread creates one instance of threads *T1* and *T2*, as well as four instances of *T3* (1xT1, 1xT2, 4xT3); in this execution, JPF monitors all threads. The second data column in Table 5.7 corresponds to the execution of the same program, with JPF being however instructed to monitor only the main thread and the instances of threads *T1* and *T2*, since no instance of thread *T3* is bound to be involved in the generation of

intermittent faults. As described in Section 5.5.1, this is realized through the setting *vm.watched.threads=main,1,2*.

watched threads	All Threads	main,T1,T2
elapsed time	00:00:10	00:00:03
new states	5147	2192
visited states	14365	1264
backtracked states	19512	3456
end states	45	-
instructions	418886	88756
max memory	303MB	169MB

Table 5.7: Examining all possible threads vs. limiting the set of threads examined by JVM.

Table 5.8 depicts how performance benefits can be obtained from applying the Children Node Reduction technique described in Section 5.5.2, regarding the code depicted in listing 5.3. The figures in this table refer to the execution of a java program with 6 threads (1xT1, 1xT2, 4xT3), varying the order of the child that is allowed to continue. Since in our example the first and second children correspond to executions of instructions by threads *T1* and *T2*, which include accesses to shared variables, these choices entail more states to be examined. Given that only these two choices may actually lead to intermittent faults, it suffices to examine only these two cases to fully uncover all intermittent fault root causes.

Children Node Reduction				
	1st child node for depths between 10-30	2nd child node for depths between 10-30	3rd child node for depths between 10-30	No cut off
Time	2 sec	1 sec	1 sec	10 sec
New states	1637	523	302	5147
Visited states	1118	344	253	14365
Backtracked states	2755	867	555	19512

Table 5.8: Children Node Reduction effect applied at different thread orders

Table 5.9 focuses on the scalability of the proposed algorithm under the optimization techniques presented in section 5.5, depicting the time needed to execute the proposed algorithm to detect faults injected to the open-source multithreaded Java web sever [276] when the thread partitioning and the state subtree pruning of specific nodes techniques (cf. subsections 5.5.1 and 5.5.2, respectively) are applied. The configuration used in this experiment is:

Number of threads in the web server request executor thread pool				
	5	10	50	100
Elapsed time	00:32:41	00:47:04	00:47:41	00:47:54

Table 5.9: JPF execution time for the Java web server simulation

```
vm.parallel.allowed.depth=40-350
vm.watched.threads=main,Thread-1,Thread-2,Thread-3,Thread-4
vm.parallel.allowed.child=[1] search.depth_limit = 350
```

This configuration effectively scans the full state tree up to the depth of 40, and beyond that point limits the detection to the first child only, since the Java web server [276] employs the worker thread model discussed in section 5.5.2 [274], and moreover worker threads are totally independent and are thus not bound to generate any intermittent errors.

We can observe that the time needed to run the fault detection algorithm increases very slowly with the overall number of threads, while additionally significant time savings against the non-optimized version (c.f. table 5.6) are introduced; these savings are quantified to 82%.

5.7 Conclusions and Future work

In this chapter we presented a methodology for intermittent fault detection that is based on the identification of suspicious shared variable access patterns in the code execution traces. Execution traces are generated using the JPF tool, which has been enhanced by a customized listener, while the suspicious access patterns that are searched for correspond to well-known parallel programming hazards. Out method has been shown to be capable of detecting intermittent faults that evade detection when other methods are used, while on the other hand introducing some false positives. In this sense, the programmer is asked to review the potential intermittent fault root causes and accept or reject them. In order to leverage the efficiency of the proposed method, we have introduced optimization methods which can exploit structural properties of the code, such as thread independence and thread subtree isolation, as well as parallel hardware capabilities. Our experiments on optimizations exploiting structural properties of the code have demonstrated that significant performance gains can be reaped.

In the context of our future work we plan to examine the following dimensions:

1. fully implement and evaluate the optimization method for exploiting parallel hardware capabilities.

2. take into account dependencies between global and local variables, which are established via assignment statements.

3. Identify and evaluate additional dependency rules.

4. study the computational complexity of the proposed algorithm, computing a theoretical upper bound for the number of possible execution paths that need to be explored.

Chapter 6

A generalized, rule-based method for the detection of intermittent faults in software programs

Abstract

Intermittent faults are a very common problem in the software world, and are very hard to locate and correct, due to the fact that they manifest their presence only under certain circumstances. Most of the existing approaches for intermittent fault identification assume that suitable instrumentation has been provided in the program, typically in the form of assertions that dictate which program states are considered to be erroneous, by examining variable values. This approach is, however, inefficient, since only errors for which appropriate instrumentation has been provided will be uncovered. In this chapter we propose a method that can be used to detect probable sources of intermittent faults within a program. Our method proposes certain points in the code, whose data interdependencies combined with their execution interweaving indicate that they could be the root cause of intermittent faults. The approach presented in this chapter extends our previous work, by examining shared variable access sequences and taking into account not only direct dependencies between shared variables, but additionally indirect ones, i.e. cases where values of local variables are computed using values of some shared variable(s), and subsequently the local variable values are used to set the value of other shared variables. The detection of suspicious access pattern, which may indicate the presence of intermittent faults, is formalized through the introduction of generalized rules; these rules are combined with model-based checking to ensure that all program execution paths are covered. The list of suspicious locations within the code is finally presented to the developer, who will decide after a thorough examination of the code, to accept or reject each of the proposals.

6.1 Introduction

In computer software, an intermittent fault is an erroneous behaviour of a software program that manifests itself at intervals, typically non-uniform, while at other times

the software operates in accordance to its specifications. Avizienis et al. [224] list a definition of intermittent faults which includes (a) *elusive permanent faults*, i.e. faults that occur under certain conditions, when some complex conditions involving the program's internal state and external interactions are satisfied, with these conditions being satisfied very infrequently and being hard to reproduce and (b) *transient faults*, which refers to *physical malfunctions* (i.e. malfunctions of the underlying hardware) and additionally as *interaction faults*, owing to interactions with external systems. Root cause analyses conducted for intermittent faults [226, 224] has determined that intermittent faults are associated with (a) concurrency errors, (b) limit conditions, including memory/disk depletion, non-expected interactions with external systems and/or data sources, lost interrupt signals, memory that has not been properly initialized etc., and (c) malfunctions of the underlying hardware.

Two special types of intermittent faults are *Mandelbugs* and *Heisenbugs*. A Mandelbug is a software-rooted intermittent fault, exists in the code and does not cause always a failure in spite of seemingly identical execution context, but makes the behavior of the software appear chaotic or even non-deterministic. The appearance of Mandelbugs can be triggered by inadequate concurrency control mechanisms in multi-threaded or parallel applications [230]. *Heisenbugs* are a different type of software-rooted intermittent faults, which either do not manifest themselves or exhibit different behavior when attempts are made to discover them [226], a behavior typically owing to changes in the software environment and execution conditions. On the contrary, *Bohrbugs* [226] are a category of bugs that always manifest themselves when the relevant code is executed, being thus significantly easier to detect and isolate using standard debugging tools and techniques.

While intuitively Mandelbugs can be deemed to be infrequent, in relation to the total number of faults in a software system, Grottke et al. [230] present an analysis of the software faults that were identified in the on-board software used for JPL/NASA space missions, and determined that Mandelbugs account for the 36.5% of the total number of software faults. Chillarege [232] correlated the types of bugs with the software quality dimensions, and concluded that Mandelbugs mainly have repercussions on the non-functional aspects of software (e.g. availability, mean time to failure and reliability), and typically do not affect the functionality of the software. These studies demonstrate that the effective tackling of intermittent faults, including Mandelbugs, is of importance.

Some usual causes of software-rooted intermittent faults in applications can be traced to race conditions and the erroneous order/inappropriate interleaving of shared variables access operations in multi-threaded applications. For example, a read-after-write (RAW) hazard [280] occurs when a variable value must be first written by some thread T_x and then read by some other thread T_y (so that T_y reads the value stored by T_x), however, due to imperfect synchronization mechanisms T_y reads the shared variable *before* T_x writes the new value. The probability that some software involves concurrency-related software faults, and the difficulty to identify the fault root cause has been determined to increase with the size, complexity and concurrency degree of the multi-threaded the software program [233, 234]. Avoiding ineffective concurrency control techniques (termed as *concurrency anti-patterns*), and adopting proper design patterns to achieve the required synchronization [233, 234]

may contribute to the reduction of the number of concurrency Mandelbugs.

The debugging techniques used to detect intermittent faults can be both static and dynamic [235]. For the detection of logical faults, using dynamic techniques, in particular, the programmer typically needs to add appropriate *assert* statements expressing program-specific conditions that must always hold. These conditions are evaluated at runtime, and when some condition is found not to hold, a fault is flagged and the developer may use a dynamic debugger to examine the program state, trying to trace the root cause of the error.

The approach proposed in this chapter aims to assist developers identify locations in the code that could lead to improper shared variables access order and therefore to intermittent faults; the root cause of improper shared variables access order is typically attributed to ineffective or erroneous concurrency control provisions, or the absence of such provisions. The work reported in this chapter is an extension of our previous work [88] which, on top of an existing debugging and verification tool, adds mechanisms that create traces of shared variable access sequences and rules that are able to identify such improper access patterns within these traces. Then, the system is able to suggest to the developer code locations that may be the root cause of these intermittent faults. It uses Java Path Finder (JPF [236]; a brief overview of JPF is given in Section 5.2.3) as the base debugger tool. In the chapter, we propose a generalized rule that extends the previous work, by identifying, also, intermittent faults caused by the interconnection / relation among multiple shared variables either explicitly or implicitly through local variables. As it uses the same general method, exploiting the capabilities of JPF, our approach is always immutable to any user configuration (e.g. parser configurations). More specifically, JPF functionalities are used to gather all the information about possible interleavings of the accesses of the shared variables from the different threads; this information is organized in a tree structure, and stored in a database, for subsequent offline processing and analysis. In particular, the tree structure is retrieved and examined order to identify the code locations that cause potential intermittent faults, and these locations are proposed to the user (i.e. the developer) for checking (e.g. code review to verify whether synchronization mechanisms are used appropriately). The examination of the tree structure is guided by a set of rules, which model patterns of erroneous order of accesses to shared variables. After the developer has examined the suggestions, s/he makes the final decision on whether each suggestion should be accepted or not. Contrary to other algorithms in the literature, the proposed approach needs no instrumentation (e.g. insertion of appropriate assertions in selected code locations) to work. In this chapter, we introduce new rules for the detection of shared variable access patterns that may indicate the presence of intermittent faults. The new rules can be combined with ones introduced in previous works, to cover a wider range of intermittent faults root causes, or quantify the probability that each point / location detected is actually a true positive, as opposed to a false positive (e.g. when more rules indicate a location as a potential intermittent fault cause, this location can be considered to have a higher probability to actually be an intermittent fault root cause).

The rest of the chapter is structured as follows: Section 6.2 overviews related work, including static and dynamic verification tools and elaborating on JPF, which is used

in our approach. Section 6.3 introduces the proposed algorithm, while Section 6.4 presents an experimental evaluation of the algorithm, considering its functionality, execution time and memory footprint. Finally, Section 6.5 concludes the chapter.

6.2 Related work

When a software program demonstrates faulty behavior, either in the testing phase or after its deployment in a production environment, software engineers undertake the task to identify the fault's exact location and its root cause. In the context of this task, software engineers utilize a multitude of techniques aiming to identify, localize and remove faults. Table 6.1 lists some major fault identification and localization techniques.

Table 6.1: Techniques for detecting software code faults

Techniques for detecting software faults	Bibliographic reference
Techniques that examine the source code statically to identify *code smells*, i.e. characteristics that may indicate a deeper problem	[242, 243]
Software fault prediction techniques which aim to identify fault-prone software modules by using some underlying properties of the software project before the actual testing process begins	[244, 281]
Techniques which use test cases for unit-level or integration testing	[33, 282, 283]
Methods for automatically generating comprehensive test case suites	[245, 246, 247, 284, 285]
Approaches for identifying faults that evade test-case based detection processes, since test case-based fault detection may miss certain faults, even under high code coverage	[248]
Techniques that minimize and manage test cases, the execution of which can consume time and resources	[249, 27]
Specialized methods for analyzing and detecting software vulnerabilities	[250, 286, 287]
Besides “traditional" fault localization techniques, which include logging, assertions, breakpoints and profiling, a number of advanced fault localization techniques have been proposed, which are classified as (a) slice-based, (b) program spectrum-based, (c) statistics-based, (d) program state-based, (e) machine learning-based, (f) data mining-based and (g) model-based techniques	[87, 288, 289, 290, 291, 292]

Recently, developments in AI have been exploited to aid fault localization, contributing additional approaches in many of the techniques presented above. For instance, Li et al. [290] locate suspicious code by treating fault localization using AI-based image pattern recognition algorithms, while Lou et al. [291] utilizes graph-based representation learning to the same effect. AI can also contribute to the explainability of fault localization [293, 294]. Nevertheless, standard fault discovery techniques and methods may fail to detect intermittent faults, due to the specific nature of this fault class [226], and consequently suitable methods need to be employed to aid software engineers to detecting and remove intermittent faults.

6.3 The Proposed Intermittent Fault Detection Algorithm

The intermittent fault detection method proposed in this chapter follows and extends the approach introduced in [88], which focuses on the identification of suspicious shared variable access sequences, which may signify the presence of intermittent faults. While existing works examine the access patterns of each shared variable in isolation, the algorithm proposed in this chapter considers additionally the value dependencies between multiple shared variables, which may be either *direct* (i.e. a shared variable $shVar_1$ is assigned the value of an expression in which shared variable $shVar_2$ participates), or *indirect*. In the latter case, one or more local variables $locVar_1$, $locVar_2$, ..., $locVar_n$ are assigned values that depend on one or more shared variables $shVar_1$, $shVar_2$, ..., $shVar_k$, and finally some shared variable $shVar_{target}$ is assigned a value that depends on the value(s) of the local variable(s). A mixture of direct and indirect dependencies between shared variables is also possible.

The enhanced intermittent fault detection algorithm presented in this chapter comprises tree parts: The first one encompasses the development of a rule base for the detection of shared variable access patterns that may indicate sources of intermittent faults. The second one is about the generation of complete execution traces for the target program, to record all possible shared variable access patterns. In this part, JPF, augmented with additional logging listeners, is used to implement the generation of the program traces. The third one comprises the application of the rule base developed in part 1 on the traces generated during step 2, in order to detect possible sources of intermittent faults within the program. These points are proposed to the user for review and, if appropriate, application of the necessary corrections.

In the following, we present the proposed algorithm in detail. For self-containment purposes, we provide a complete description of the algorithm, including the portions of the original algorithm presented in [88]; the extensions introduced in this chapter are clearly designated.

Central to the operation of the enhanced intermittent fault detection algorithm is the concept of *shared variable value dependency*. In the following, we initially present a formal definition of shared variable value dependency, and subsequently we proceed with the presentation of the rules for the detection of suspicious access patterns, the methods used to create execution traces and the detection of suspicious access patterns within the generated traces.

6.3.1 Shared variable value dependency

Let $T = \{t_1, t_2, \ldots, t_{NT}\}$ be the set of threads of a multi-threaded program P. Each thread t_i contains an ordered sequence of instructions $(i_{t_i,1}, i_{t_i,2}, \ldots, i_{t_i,NI_{t_i}})$, where NI_{t_i} is the number of instructions in thread t_i. A *code fragment* $CF(t_i, x, y)$ of thread t_i, where $1 \leq x \leq y \leq NI_{t_i}$ is the ordered sequence of instructions of t_i, $(i_{t_i,x}, i_{t_i,x+1}, \ldots, i_{t_i,y})$. Let $S = \{s_1, s_2, \ldots, s_{NS}\}$ be the set of shared variables appearing in P, and let $L = \{l_1, l_2, \ldots, l_{NL}\}$ be the set of local variables appearing

in P. Each local variable exists within a single thread in T and is accessed only by the specific thread, while a shared variable is common to all threads within T and can be accessed by any number of threads. Note that while a specific local variable name lvn may be referenced in different threads t_i and t_j, the name lvn in t_i corresponds to a different local variable than the name lvn in t_j, and therefore to different elements of L.

The value of a shared variable s_i depends on the value of shared variable s_j, denoted as $s_i \underset{CF}{\longleftarrow} s_j$ in the context of a code fragment CF iff one the following conditions hold:

1. an instruction $i_x \in CF$ is an assignment of the form $var = expression$, where var is the shared variable s_i and $expression$ involves the shared variable s_j. This is a case of a *direct dependency* of s_i on the value of s_j.

2. a sequence of instructions $\{i_{x_1}, i_{x_2}, \ldots i_{x_p}\} \subset CF$ exists, where $x_1 < x_2 < \ldots < x_p$, where all the following conditions hold:

   i each i_{x_i} is an assignment of the form $var_{x_i} = expression_{x_i}$

   ii $expression_{x_1}$ involves the shared variable s_j

   iii var_{x_p} is the shared variable s_i

   iv a set of local variables $\{l_{x_1}, l_{x_2}, \ldots, lx_{p-1}\} \subset L$ exists, such that $\forall i, 1 \leq i \leq p-1 : (var_{x_i}$ is the local variable $l_{x_i}) \wedge (expression_{x_{i+1}}$ involves local variable $l_{x_i})$.

   In this case, we have an indirect dependency of s_i on the value of s_j.

6.3.2 Detecting intermittent faults involving indirect shared variable dependencies

The extended hazard-based intermittent fault identification algorithm proposed in this chapter extends the method presented in [88], by augmenting the rules for locating *write-after-write* and *read-after-write* hazards owing to direct dependencies between shared variables (which may correspond to root causes of intermittent faults), with additional rules that are able to detect hazards owing to indirect dependencies. The initial ruleset introduced in [88] comprises rules 1 and 2 listed below, while rule 3 corresponds to the addition proposed in this chapter, to accommodate indirect variable dependencies. For clarity purposes, in this subsection we present an initial version of the enhanced ruleset that considers indirect accesses that span across two code fragments and involve exactly two shared variables, while in Section 6.3.3 we present a generalized version of the newly introduced rule, which is not subject to these limitations. More specifically, the enhanced rule set used for suspicious shared variable access pattern identification is as follows:

Rule 1) Sequences of operations of the form $read_{T1}(X)$, $write_{T2}(X)$, $write_{T1}(X)$, corresponding to *direct shared variable dependency write-after-write hazards* (the notation $read_T(X)$ denotes that thread T reads variable X; and similarly for $write_T(X)$).

Rule 2) Sequences of operation of the form $read_{T1}(X)$, $write_{T2}(X)$, $read_{T1}(X)$, corresponding to the *direct shared variable dependency read-after-write hazards*.

Rule 3) Let $CF_1 = \{i_{1.1}, i_{1.2}, \ldots, i_{1.k}\}$ be a code fragment, such that $i_{1.1} = read(X), i_{1.k} = write(Y) \wedge Y \underset{CF_1}{\longleftarrow} X$. Let $CF_2 = \{i_{2.1}, i_{2.2}, \ldots, i_{2.j}\}$ be a second code fragment, such that $i_{2.1} = read(Y), i_{2.j} = write(X) \wedge X \underset{CF_2}{\longleftarrow} Y$. Then, any execution schedule ES where threads T_1, T_2 execute CF_1, CF_2, respectively, and the instructions of CF_1 and CF_2 interleave in the execution schedule, i.e. $\overline{(i_{1.k} \prec i_{2.1}) \vee (i_{2.j} \prec i_{1.1})} \iff \overline{(i_{2.1} \prec i_{1.k})} \wedge \overline{(i_{1.1} \prec i_{2.j})}$ (the notation $i_x \prec i_j$ denotes that instruction i_x precedes instruction i_j in the execution schedule) involves a concurrency hazard, where the result of the execution schedule is not equivalent to any serial execution of threads T_1 and T_2.

As noted above, rules 1 and 2 are introduced and utilized in [88]. Rule 3 is introduced in this work and extends the operation of the intermittent fault detection framework to consider hazards owing to indirect shared variable dependencies. To illustrate the functionality of rule 3, let us consider the code listed in Listing 6.1.

Listing 6.1: An example with shared variable dependencies, which produces faults intermittently

```
private static Object filled1 = 0;
private static Object filled2 = 0;

public void method1() {
   int a = (int)filled1; (M1.1)
   a = a + 1; (M1.2)
   filled2 = a; (M1.3)

}

public void method2() {
   b = (int)filled2; (M2.1)
   b = b + 2; (M2.2)
   filled1 = b; (M2.3)
}

run in parallel:
method1();
method2();
```

In Listing 6.1 we can observe that the code in methods `method1` and `method2` may be executed concurrently. These methods access shared variables `filled1` and `filled2` without the use of synchronization primitives such as semaphores. Considering the premise that any concurrent execution arrangement is deemed correct if it is equivalent (i.e. it leads to the same state) to any serial execution of the relevant code fragments [295], the acceptable states after the conclusion of the concurrent execution of `method1` and `method2` are as follows:

1. the state corresponding to effect of the sequential execution of `method1` followed by the execution of `method2`, in which case variable `filled1` would be set to 3, and variable `filled2` would be set to 1.

2. the state corresponding to effect of the sequential execution of `method2` followed by the execution of `method1`, in which case variable `filled1` would be set to 2, and variable `filled2` would be set to 3.

However, when instructions interleave due to concurrency, a number of non-sequential schedules may be formulated and –under the absence of synchronization mechanisms–the concurrent execution of these schedules may lead to states that are not equivalent to any serial execution. Consider for example the schedule $ES = (M1.1)(M1.2)(M2.1)(M2.2)(M2.3)(M1.3)$, which may occur either due to physical concurrency or due to thread switching/preemption. In more detail, `method1` begins executing on thread $T1$ and after the execution of instruction $(M1.1)(M1.2)$ concludes, $T1$ is preempted, followed by a switch to thread $T2$ executing method `method2`. Then thread $T2$ executes all instructions of `method2` i.e. $(M2.1)(M2.2)(M2.3)$, and finally a second thread switching occurs, allowing thread $T1$ to resume and execute instruction $(M1.3)$. If this execution schedule is followed, variable `filled1` would be set to 2 while variable `filled2` would be set to 1, a result state that is not equivalent to any of the two possible serial execution arrangements, and is therefore considered incorrect.

In terms of shared variable accesses, the execution schedule ES produces the access pattern $AP(ES) = read_{T1}(filled1), read_{T2}(filled2), write_{T2}(filled1), write_{T1}(filled2)$. Clearly, neither rule 1 nor rule 2 are able to detect the concurrency hazard, because both rules detect hazards where the execution schedule involves at least three accesses of a single shared variable, and within $AP(ES)$ each shared variable is accessed exactly twice.

On the contrary, rule 3 is able to detect the concurrency hazard, since all antecedents of the rule are satisfied:

1. method `method1` is a code fragment $CF_1 = (M1.1), (M1.2), (M1.3)$ such that $(M1.1) = read(filled1) \wedge (M1.3) = write(filled2) \wedge filled2 \underset{CF_1}{\longleftarrow} filled1$ (the last element of the conjunction clearly holds, since in this code fragment the value assigned to $filled2$ depends on the value of $filled1$).

2. method `method2` is a code fragment $CF_2 = (M2.1), (M2.2), (M2.3)$ such that $(M2.1) = read(filled2) \wedge (M2.3) = write(filled1) \wedge filled1 \underset{CF_2}{\longleftarrow} filled2$.

3. within execution schedule ES, threads $T1$ and $T2$ executing code fragments CF_1 and CF_2 respectively interleave.

Therefore, the application of rule 3 leads to the detection of a concurrency hazard which would evade detection when utilizing the method proposed in [88].

6.3.3 Generalized detection of intermittent faults involving indirect shared variable dependencies

Let us now consider the code in Listing 6.2. This code has a similar structure to the code in Listing 6.2 and exhibits an analogous concurrency hazard. However, three shared variables (*filled*, *filled2* and *filled3*) and three code fragments (*method1*, *method2* and *method3*) are involved, instead of two; this renders the hazard not detectable by rule 3 presented in Section 6.3.2, because rule 3 is only able to detect issues involving exactly two code fragments, and respectively two shared variables.

Listing 6.2: An example with shared variable dependencies among three threads, which produces faults intermittently

```
private static int filled = 0;
private static int filled2 = 0;
private static int filled3 = 0;

public void method1() {
   int a = filled; (M1.1)
   a = a * 2; (M1.2)
   filled2 = a; (M1.3)
}

public void method2() {
   int b = filled2; (M2.1)
   b = b + 7; (M2.2)
   filled3=b; (M2.3)
}

public void method3() {
   int c = filled3; (M3.1)
   c = c - 7; (M3.2)
   filled = c; (M3.3)
}

run in parallel:
method1();
method2();
method3();
```

This example demonstrates the need to introduce a generalized form of rule 3, which is able to detect hazards that are due to interactions of multiple code fragments. The generalization of rule 3 is expressed in rule 3' is listed below. Please note that rule 3' replaces rule 3:

Rule 3') Let $SVs = \{SV_1, SV_2, \ldots, SV_N\}$ be a set of shared variables and $CFs = \{CF_1, CF_2, \ldots, CF_N\}$ be a set of code fragments such that:

(a) $CF_a = \{i_{a.1}, i_{a.2}, \ldots, i_{a.k_a}\}, \forall a : 1 \leq a \leq N$

(b) $i_{a.1} = read(SV_a) \wedge i_{a.k_a} = write(SV_{a+1}) \wedge SV_{a+1} \xleftarrow[CF_a]{} SV_a, \forall a : 1 \leq a \leq N-1$

(c) $i_{N.1} = read(SV_N) \wedge i_{N.k_N} = write(SV_1) \wedge SV_1 \xleftarrow[CF_N]{} SV_N$

Then, any execution schedule ES where threads $T_1, T_2, \ldots, T_N$ execute CF_1, $CF_2, \ldots CF_N$, respectively, and the instructions of $CF_1, CF_2, \ldots, CF_N$ interleave in the execution schedule i.e.
$\overline{(i_{2.1} \prec i_{1.k_1})} \wedge \overline{(i_{1.1} \prec i_{2.k_2})} \wedge$

$\overline{(i_{3.1} \prec i_{2.k_2})} \wedge \overline{(i_{2.1} \prec i_{3.k_3})} \wedge$

...

$\overline{(i_{1.1} \prec i_{N.k_N})} \wedge \overline{(i_{N.1} \prec i_{1.k_1})}$

involves a concurrency hazard, where the result of the execution schedule is not equivalent to any serial execution of threads $T_1, T_2, \ldots,$
T_N.

Please note that rule 3' does not subsume neither rule 1 or rule 2, since rule 3' is designed to match only hazards involving direct or indirect dependencies between distinct variables, while rules 1 and 2 match hazards that involve only a single variable. Hence, to provide full intermittent fault detection capabilities, all three rules 1, 2 and 3' must be utilized.

6.4 Experimental Evaluation

In this section we present the experiments conducted to assess the feasibility and effectiveness of the proposed generalized detection algorithm. The base code used for the experiments is the code for the Java Web Server [276] application, which was also used in the previous work [88]. The JPF listeners used to collect execution data in order to match these data against rules 1, 2 and 3' can be accessed at [296] (version which stores the data sensed from the listeners in the database in order to be processed at a latter stage) and [297] (a version which calculates the relations among fields and variables before the storage in the database). For the performance experiments, the former version of the listeners was used.

In particular the experiments aimed to evaluate

i) whether the proposed algorithm is able to detect potential sources of intermittent faults comprehensively and accurately, i.e. (a) whether potential sources of intermittent faults are identified and (b) whether all the code fragments flagged by the proposed algorithm actually correspond to potential sources of intermittent faults (as opposed to *false positives*).

ii) the amount of time needed by the algorithm to check a piece of software.

iii) the scalability of the algorithm considering (a) the degree of program parallelism (multithreading) and (b) the size and complexity of the code of the parallely executing threads.

The experiments were run using the Java web server [276] as a multithreaded application using two configurations. In the first configuration (which will be referred to as *simple thread code*), the worker code executing for each request was replaced by the code depicted in Listing 6.3. This code implements basic queue management functionality, with a faulty implementation of synchronization which is owing to improperly controlled *direct* dependencies/accesses of shared variables. More specifically, within the `put` method it is initially verified that the queue has adequate space to accommodate the newly inserted item (instruction `M2`); provided that this check is successful, the queue item count is increased instruction `M5`) and the item is accommodated in the queue (instruction `M6`). Note that while a mutex lock is obtained for the `put` operation (instruction `M1`), this lock is released immediately after the availability of queue positions is verified (instruction M3), and re-acquired before the queue is modified (instruction `M4`); therefore it is possible that while a thread that has determined that a single empty space exists in the queue executes code between instructions `M3` and `M4`, another thread executes the `put` method, and manages to acquire the lock through instruction `M2`, and also concludes (incorrectly) that one queue location is available. Proper synchronization would be achieved if instructions `M3` and `M4` were removed, allowing for the whole process to be carried out in an atomic fashion.

Additionally, the code in Listing 6.3 invokes the methods `method1`, `method2` and `method3` illustrated in Listing 6.2; this code has been analyzed in Section 6.3.2 and effectively contains code causing intermittent faults owing to improperly controlled *indirect* dependencies/accesses of shared variables. The complete replication package is available on Zenodo [298].

In the second configuration (which will be referred to as *complex thread code*) the original worker code was maintained, but the code depicted in Listing 6.3 was injected into the worker code. Since the Java web server [276] worker code includes shared variables and significantly more complex control logic than the code in Listing 6.3, this configuration entails a considerably larger state space than needs to be examined, while is additionally akin to real-world programs, in terms of code size and complexity.

Table 6.2: Complexity metrics for the two experimental configurations

Metric	Configuration	
	Simple thread code	Complex thread code
Number of classes used in worker code execution	1	43
Worker code lines	36	2239
Number of branch instructions in code[37]	4	482

[37]This metric corresponds to McCabe's complexity [299], reflecting the number of execution paths that need to be examined during testing.

In both experiments, we varied the following parameters:

i) the number of worker threads of the Java web server [276], to determine the effect of the degree of parallelism on the time needed by the algorithm to check the possible execution paths and identify the potential sources of intermittent faults and

ii) whether the traces of the program were stored in main memory or within a database, in order to gain insight on the trade-offs between scalability and performance.

As a baseline for the comparison of our work, we use the intermittent fault detection algorithm reported in [88], which also employs a rule-based approach examining access patterns and does not necessitate any additional work on behalf of programmers, such as injection of assertions or log file creation and examination.

Listing 6.3: Simple code for queue management, with intermittent fault occurences

```
    private final static Lock mutex = new ReentrantLock();
    private static int numFilled = 0;
    private static ArrayList itemQueue = new ArrayList();
    private static final int MAX_ITEMS = 2;

    public void put(Object item) {

       mutex.lock(); (M1)
       if (numFilled < MAX_ITEMS) { (M2)
          mutex.unlock(); (M3)
          //other code
       } else {
          mutex.unlock();
          return;
       }
       mutex.lock(); (M4)
       // assert (numFilled < MAX_ITEMS);
       numFilled++; (M5)
       itemQueue.add(item); (M6)
       mutex.unlock(); (M7}
    switch (threadId() % 3) {
       case 0: method1(); break;
       case 1: method2(); break;
       case 2: method3(); break;
    }
    return ;
}

public Object get() {
    Object item = null;
    mutex.lock();
```

```
    if (numFilled > 0) {
       numFilled--; (M8)
       item = itemQueue.remove(0);
    }
    mutex.unlock();
    switch (threadId() % 3) {
       case 0: method1(); break;
       case 1: method2(); break;
       case 2: method3(); break;
    }
    return item;
}
```

All the experiments reported in this section, have been executed on a PowerEdge M910 blade server, with 256 GBytes of physical memory and four 8-core E7-4830 Intel Xeon processors. Since JPF, however, runs as a single-threaded application, only one execution core was utilized for intermittent fault analysis. JPF's Java environment had been configured to use up to 64 GBytes of memory.

6.4.1 Intermittent fault detection

Table 6.3 illustrates the results regarding the intermittent fault detection capabilities of the proposed algorithm and the baseline algorithm. The proposed algorithm was able to detect both (a) intermittent faults owing to single variable concurrency hazards (i.e. the synchronization issue present in the code of Listing 6.3) and (b) intermittent faults owing to direct or indirect shared variable dependence (i.e. the synchronization issue present in the code of Listing 6.2). The enhanced detection capabilities of the proposed algorithm are owing to the presence of rule 3', which is capable of capturing suspicious access patterns involving dependencies between shared variables.

In both the proposed and the baseline algorithm, a single false positive was flagged; the false positive is owing to the shared variable access sequence $read(T_1)$, $write(T_2)$, $write(T_1)$ identified in the program execution trace, which matches rule 1. However, after the programmer reviewed the code, it was concluded that the lock and unlock instructions present in the code effectively provide sufficient atomicity guarantees for the specific access sequence, and therefore this report is flagged as false positive. This false positive instance is analyzed in detail in [88].

The number of variable access traces examined in the context of the execution of the fault detection algorithm varies, depending on the number of threads. When the complex code was executed using 4 worker threads, 34,985 variable access traces were generated and examined, while for the 5-threads execution the corresponding number of variable access traces is 162,669.

6.4.2 Performance evaluation

As noted above, in order to assess the feasibility and efficiency of the proposed intermittent fault detection algorithm, the algorithm was tested under two settings:

Table 6.3: Intermittent fault detection capabilities of the evaluated algorithms

	Proposed algorithm	Baseline algorithm [88]
Detection of intermittent faults owing to single-variable concurrency hazards	✓	✓
Detection of intermittent faults owing to indirect shared variable dependence	✓	✗
True positives	3	2
False positives	✓	✓
Rule 1	1	1
Rule 2	0	0
Rule 3	0	0

- a setting where execution traces were stored in main memory structures and
- a setting where execution traces were stored in a MongoDB database.

aiming to assess the trade-offs between execution time and memory footprint. In both cases, during program execution the execution traces were collected, and the analysis of execution traces for suspicious access pattern identification was conducted afterwards. Furthermore, we varied the number of threads, to gain insight on the effect of the degree of concurrency on the execution time of the algorithm; the number of threads affects the number of potential execution schedules that will be examined and hence the time that the algorithm will need to check these paths for suspicious access patterns. Figure 6.1 depicts the execution time under the two different settings, considering the two code configurations presented in Section 6.4 (*simple thread code* and *complex thread code*). For the *complex thread code* configuration we list only results for the MongoDB setting, because the memory requirements of the memory storage-based setting were deemed excessive.

In Figure 6.1, we can observe that the use of the database instead of memory structures for execution trace storage introduces an overhead of approximately 25% regarding execution time. This increment is deemed tolerable, while it has to be noted that MongoDB consumes approximately 15% of the processing power of an additional CPU core of the same machine. While the availability of a second core is a commodity for modern machines, this aspect has to be taken into account for machine rightsizing, especially when tests are performed on servers where resources are shared between tasks and where CPU usage charges may apply. Figure 6.1 additionally illustrates that the complexity of the thread code plays an important role in the scalability of the fault detection algorithm. While for a small number of threads (e.g. 3 or 4) the execution times for both the simple code and the complex code configurations are comparable, for higher number of threads, the fault detection algorithm execution time rises steeply, owing to the *state explosion* issue [270], i.e.

the number of distinct execution schedules that need to be examined, considering not only instructions within the method where synchronization occurs, but also instructions within methods that are invoked from this method. To tackle the state explosion issue, the algorithm implementation provides to the tester the option to designate methods as "excluded", in which case the execution of the code therein is not monitored. This can be applied to methods that are known not to access or modify (a) shared variables and (b) local variables that either depend on the values of shared variables or are used to calculate values of shared variables.

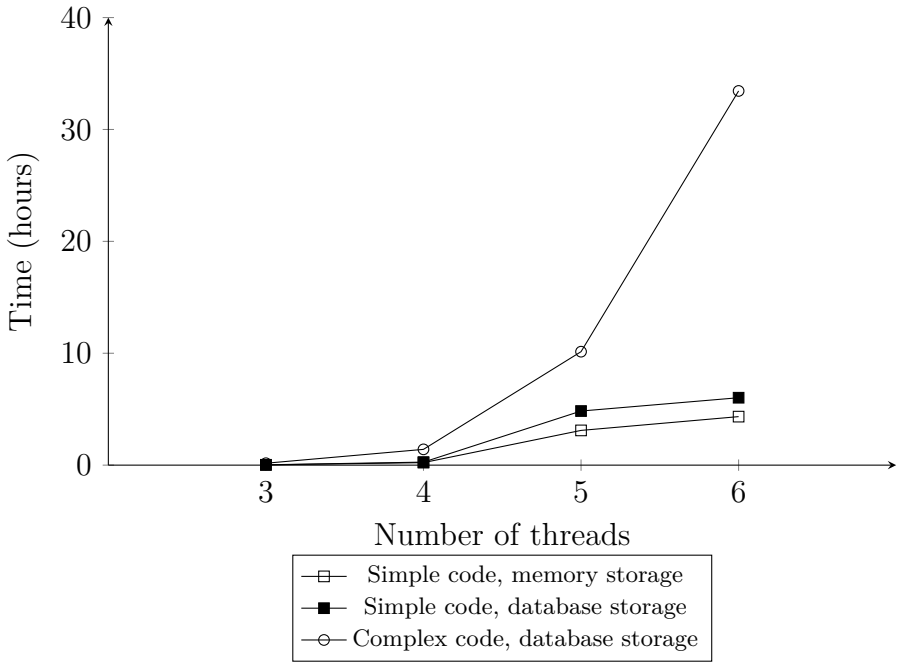

Figure 6.1: Execution time under different code configurations and storage settings.

Figure 6.2 depicts the memory footprint of the intermittent fault detection algorithm, under the code configuration and storage settings discussed above. We can observe that the number of threads has a significant impact on the memory needed to execute the algorithm under the memory-based storage setting, even in the simple-code configuration. On the other hand, when the database-oriented storage setting is employed the memory requirements exhibit only small increments when either the number of threads or the code complexity increases, offering a more scalable solution in terms of memory requirements.

Table 6.4 illustrates the experiment results, concerning the execution time and the memory footprint of the intermittent fault detection algorithm in comparison to the algorithm presented in [88] (nb. the algorithm presented in [88] uses only memory-based storage). The metrics concern the execution of six parallel threads under both the complex code configuration. We can observe that the realization of the enhanced intermittent fault detection capabilities, which are offered by the proposed algorithm, leads to the increment of processing time or memory resources requirements, as compared to the algorithm presented in [88]. We can also notice the interplay between execution time and memory requirements, where the memory-based setting is faster, but less scalable than the database-oriented setting.

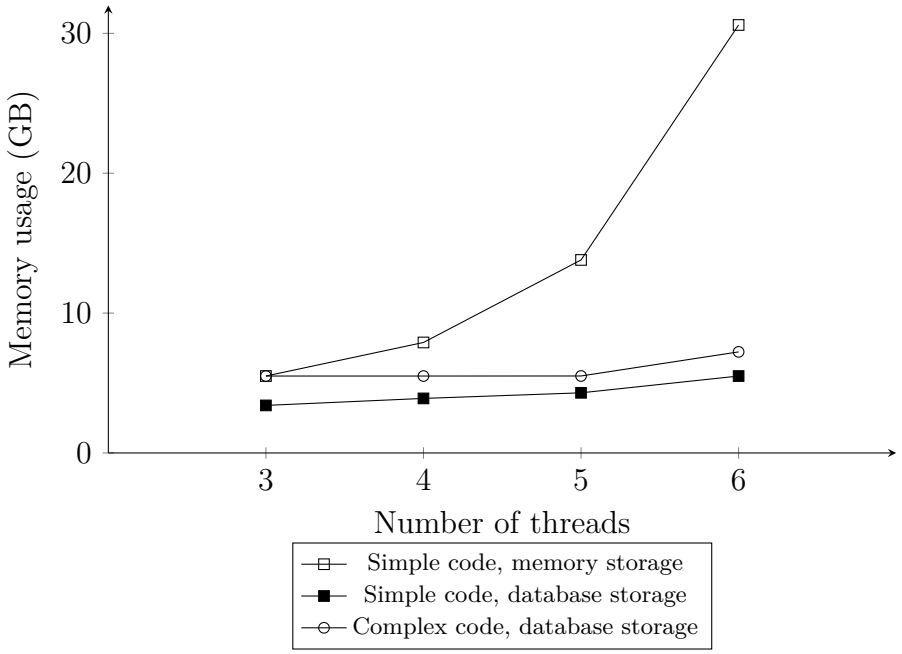

Figure 6.2: Memory usage under different configurations.

Table 6.4: Execution statistics for the fault detection process of the multithreaded Java websever under different configurations and settings

Algorithm / Setting	Simple code		Complex code	
	Runtime	Memory	Runtime	Memory
Baseline algorithm [88] / memory	2.4 h	9.2 GB	4.1 h	19.4 GB
Proposed algorithm / memory	4.3 h	30.6 GB	-	-
Proposed algorithm / DB	6.0 h	5.5 GB	33.4 h	7.5 GB

6.5 Conclusions

In this chapter we presented a novel method for intermittent fault detection. The proposed approach examines value dependencies between shared variables and related access sequences, to identify suspicious patterns, and the relevant code fragments are correspondingly flagged and presented to developers for further scrutiny. The algorithm introduced in this chapter extends previous work by considering not only direct value dependencies between shared variables, but additionally indirect ones, where local variables are involved in the value propagation between shared variables. The proposed approach augments the intermittent fault detection capability of previously published algorithms, allowing for identification of faulty code which would evade detection by previous methods. Additionally, no false positives were introduced.

The performance of the proposed algorithm has been experimentally evaluated considering runtime and memory footprint, and compared to that of previous approaches. Based on the experiments, it is shown that the algorithm is feasible, exhibiting enhanced detection capabilities as compared to baseline algorithms, requiring, however, increased processing time or memory resources, depending on the setting used (memory-based or database-oriented storage).

The proposed method can identify intermittent faults that are reflected as suspicious variable access patterns. There do exist classes of faults that are beyond the scope of the proposed work and hence cannot be identified, including faults owing to malfunctions of the underlying hardware, resource depletion, changes to memory locations (mapping to variables) by hardware components (e.g. cases of memory-mapped input/output) and lost interrupt signals. In our future work we will consider the accommodation of methods that would enable the detection of intermittent faults owing to these causes.

Our future work will focus on the optimization of the algorithm performance, the identification and evaluation of additional dependency rules (e.g. rules expressing dependencies between software variables and memory locations directly modified by hardware components, accommodated under the concept of *volatile* variables in C/C++ and Java [300]), as well as the study of the applicability of the proposed algorithm in multi-process environments. Concerning the optimization of the algorithm performance, an experimental version of the listener which optimizes the storage of access sequences in the database is being developed; the current version of the experimental version of the listener can be accessed at [301].

6.6 Optimizing the performance of the generalized, rule-based method for the detection of intermittent faults in software programs

The more the shared variables and the related local variables that are monitored with JPF, the more time is needed to excecute all the different excecution paths of the calculated tree. Since this time is increased, it is omptimal to limit the parameters to monitor in order to decrease the time of the execution. Particullarly, when local variable monitoring is involved, the time needed to execute jpf is often prohibitive because of the number of the local variables that are included in the calculations.

On the other hand, the inclusion of the local variable monitoring does not affect the final result, except in terms of time execution. Taking this into account, an optimal solution is to execute JPF monitoring only the shared variables of interest. These, for example, could be the ones that affect the outcome of the execution in the end and not the internal ones, since the shared variables preciding the ones at the outcome would affect them.

In the cases where the software program has deterministic input variables and does not include random functions during the execution, it would also be expected to have deterministic output. Thus, comparing the results at the end states of the JPF execution tree, could reveil which paths have different results, and which would contain the intermittent fault.

Following steps, would be to re-execute JPF using "the generalized, rule-based method for the detection of intermittent faults in software programs", but only for the subset of execution paths that have different results of the monitored shared variables at the end states. This way the method does not add delays because of the additional monitoring, and the execution time would be almost the same.

Let's consider the following example :

Listing 6.4: A simple code example

```
private static int a = 1;
private static int b = 2;
private static int c = 0;


public void function1() {

   a = b + 1;          (T1.1)
   int x1 = a + b;        (T1.2)
   x1++;
   c = x1*3;           (T1.3)

   int result1 = c - 4; (T1.4)

}


public void function2() {

   b = a + 1;          (T2.1)
   int x2 = a + b;        (T2.2)
   x2++;
   c = x2*3;           (T2.3)

   int result2 = c - 4; (T2.4)

}
```

In this case, some possible different execution paths would be :
(T1.1)-(T1.2)-(T1.3)-(T1.4)-(T2.1)-(T2.2)-(T2.3)-(T2.4) => (result1=14 , result2=20)
(T2.1)-(T1.1)-(T1.2)-(T1.3)-(T1.4)-(T2.2)-(T2.3)-(T2.4) => (result1=14 , result2=14)
(T1.1)-(T2.1)-(T1.2)-(T1.3)-(T1.4)-(T2.2)-(T2.3)-(T2.4) => (result1=20 , result2=20)
(T1.1)-(T1.2)-(T1.3)-(T2.1)-(T1.4)-(T2.2)-(T2.3)-(T2.4) => (result1=14 , result2=20)
(T1.1)-(T1.2)-(T1.3)-(T2.1)-(T2.2)-(T2.3)-(T1.4)-(T2.4) => (result1=14 , result2=20)
(T2.1)-(T2.2)-(T2.3)-(T2.4)-(T1.1)-(T1.2)-(T1.3)-(T1.4) => (result1=14 , result2=8)
(T1.1)-(T2.1)-(T2.2)-(T2.3)-(T2.4)-(T1.2)-(T1.3)-(T1.4) => (result1=20 , result2=20)
(T2.1)-(T1.1)-(T2.2)-(T2.3)-(T2.4)-(T1.2)-(T1.3)-(T1.4) => (result1=14 , result2=14)
(T2.1)-(T2.2)-(T2.3)-(T1.1)-(T2.4)-(T1.2)-(T1.3)-(T1.4) => (result1=14 , result2=8)
(T2.1)-(T2.2)-(T2.3)-(T1.1)-(T1.2)-(T1.3)-(T2.4)-(T1.4) => (result1=14 , result2=8)

Since the result is expected to be deterministic, and we have result sets with different values, it is evident that there exist intermittent faults in the code.

Regarding the optimal execution of the intermittent fault detection process, two options can be considered:

- execute the generalized, rule-based method for the detection of intermittent faults, for all the paths that have resulted in different outcomes.
- choose, only paths with different outcomes, then execute the generalized, rule-based method for the detection of intermittent faults and then iterate.

The benefits and concerns for these methods could be further surveyed in future research.

Chapter 7

The Tentative-Patent and Pre-License Frameworks

Abstract

A patent is an exclusive right granted for an invention, which is a product or a process that provides, in general, a new way of doing something, or offers a new technical solution to a problem. To obtain a patent, technical information about the invention must be disclosed to the public in a patent application [302]. While most patents start from new ideas, ideas cannot be patented. According to the law they should be transformed to one of the following categories in order to be considered an invention : 1. a process, 2. a machine, 3. a manufacture, 4. a new composition of matter or 5. a combination of the above [303, 304]. The distance that must be covered from the conception of the idea to the creation of the invention, and the subsequent obtaining of the patent, may be shorter or longer. An important issue is that existing procedures for obtaining patents cannot be applied to ideas only, but only to inventions, under the definitions listed above; hence, new ideas and concepts in some area cannot be protected by patents until the relevant invention has been created. However this shortcoming may leave the conceptor unprotected until the invention is built, which may be a serious issue when the conceptor may need to disclose portions or elements of the innovative idea, for instance to venture capitals of financial institutions, during efforts to secure funding for transforming ideas into inventions.

In this chapter we propose a *tentative-patent* framework, which would grant tentative-patent rights to the person who conceived and registered the idea. The tentative-patent phase covers the duration from the conception of the idea to the creation of the invention, which could result in a patent application or some other license, providing an attestation of the claims of the conceptor and thus providing guarantees for the conceptor of the idea. Tentative-patents can also be applied to inventions.

Additionally, in this chapter, we propose a *pre-license framework*. This framework allows for an adaptation period, during which the authors/creators may experiment and test in order to decide which license best suites their work. In this way, the proposed framework provides the authors/creators of the work with elevated flexibility regarding the choice of the license they will apply to it. The proposed framework includes provisions for making re-licensing smooth for the users of the pre-licensed work. The framework does not replace existing licenses, copyright schemes and patent laws and does not intend to any modification of the existing options; instead it supplements them to create a more flexible licensing scheme.

7.1 Introduction

Patent laws have been created around the globe for protecting the inventions and their creators, providing patent rights to the inventors for the specific innovative work for some period of time. Patent-related legislation varies among countries. Specifically, in European Union, there is the European Patent Convention, which describes the patent procedures in EU, and in each of the countries by referencing to the specific country laws. The European Patent Convention (EPC) can be found at the EPC page [305], while the related National Laws can be accessed in [306].

An essential and laborious task in the patent process is the step of verifying that the patent claims are original and innovative. According to the procedure scheme, the idea (or innovation) to be tentatively-patented is openly advertised for some period, during which it can be challenged by persons or enterprises who may claim that they already possess intellectual rights on the claims. If the claims are not challenged, then the conceptor is granted tentative-patent rights on the claims.

The possible-patent phase covers the time from the conception of the idea to the creation of the invention. This period is deemed sufficient, since from the creation of the invention and onward, the existing provisions for patent applications and/or licensing can be applied to secure the conceptor's rights to the tentatively patented idea, which is his/her intellectual property. As noted above, the tentative-patent framework aims to furnish innovators with tools for patenting concepts, including theoretical works and ideas, as well as the usage perspective of innovative systems, which are not currently covered by patenting and/or licensing schemes.

In all cases, the tentatively-patented work gains the so called tentative-patent rights, which include (a) recognition of the creator, (b) rendering of the work non-patentable by others, and (c) the obligation to inform the creator regarding any dependent work and include him/her to it as appropriate. In addition, it constitutes a cost-free solution, as contrasted to the patent framework. This characteristic promotes democratization and equity, permitting to any interested person to tentatively-patent their work, independently of their financial capacity. This aspect is deemed of high importance since the cost of obtaining a patent in most countries incurs elevated costs, which exclude large portions of the population from the capacity to obtain a patent. This mode of patenting is also expected to serve the purpose of protecting the conceptor's intellectual property rights when part of the idea and/or implementation plans need to be exposed to third parties, such as financial institutes or venture funds, while seeking financing to create an invention that can be patented according the existing patent workflow.

In the second part of the chapter, the *pre-license* framework is described. The *pre-license* framework aims to supplement current license frameworks that give the opportunity to the author/creator of a work to choose the desirable license, providing elevated license choice flexibility to software providers, while at the same time guaranteeing a smooth adaption of possible re-licensing by the users.

Licensing can be permissive or restrictive, depending on the authors'/creators' preferences. In the following, we describe the concept of the Open Source License and the Product License, both of which can be used as licenses of the possible-license framework.

According to the definition of the Open Source Initiative-approved licenses "Open source licenses are licenses that comply with the Open Source Definition –in brief, they allow software to be freely used, modified, and shared" [307]. In order to comply with the open source definition, a license must include clauses granting appropriate rights concerning 10 distinct areas of software characteristics, use and reuse, which are free

redistribution; (availability of) source code; derived works; integrity of the author's source code; no discrimination against persons or groups; no discrimination against fields of endeavor; distribution of license; no product specificity; no restrictions on other software; and technological neutrality [308]. The Open Source Initiative categorizes approved licenses under nine categories, namely "International", "Non-Reusable", "Other/Miscellaneous", "Popular / Strong Community", "Redundant with more popular", "Special Purpose", "Superseded", "Uncategorized" and "Voluntarily retired" [307]. An alternative definition of Open Source License is provided by Law Insider, according to which "Open Source License means any license meeting the Open Source Definition (as promulgated by the Open Source Initiative) or the Free Software Definition (as promulgated by the Free Software Foundation), or any substantially similar license, including any license approved by the Open Source Initiative or any Creative Commons License. "Open Source Licenses" shall include Copyleft Licenses" [309].

Table 7.1 describes some of the major Open Source licenses available, according to the summary of their characteristics given in [310]:

Open source software licenses can be also used for non-software work and are often the best choice, especially when the work in question can be edited and versioned as source (e.g., open source hardware designs) [311].

According to the Definition of Product License at *law-insider*, Product License means any written contract, license, lease or other agreement entered into by, on behalf of or under authority of Seller or its Affiliates in the ordinary course of business consistent with past practice, prior to the Closing, including any click-through or shrink-wrap license, that (a) accompanies the sale, servicing (including support, maintenance and installation), licensing or provision of any product or service of Seller or its Affiliates, to a Person (including re-sellers and distributors), by or on behalf of Seller or its Affiliates, or (b) is incident to the provision of the development and/or servicing (including support, maintenance and installation) services by third party contractors, consultants and other providers, and (c) in each case includes rights under the Transferred Intellectual Property consisting solely of a non exclusive grant of a license or similar rights (express or implied) to such Person under any of such Transferred Intellectual Property in favor of such Person, solely regarding such Person's rights in connection with such product or service, and in respect of re-sellers and distributors, rights to market and sell such products and services [312].

Following the definitions of Copyright Alliance, Table 7.2 indicates the differences among Patents, Copyrights and Trademarks [313]:

The pre-license framework can include all the existing or custom licenses, including the ones described in this section.

7.2 Related work

7.2.1 Patents

Patents are a cornerstone of innovation, granting inventors exclusive rights to their inventions for a limited period.

The patent landscape encompasses various types of inventions:

1. Utility Patents: Protect new and useful processes, machines, articles of manufacture, or compositions of matter. These are the most common type, encompassing inventions as diverse as smartphones and life-saving drugs.

Licenses	Permissions	Conditions	Limitations
GNU AGPLv3	Commercial use, Distribution, Modification, Patent use, Private use	Disclose source, License and copyright notice, Network use is distribution, Same license, State changes	Liability, Warranty
GNU GPLv3	Commercial use, Distribution, Modification, Patent use, Private use	Disclose source, License and copyright notice, Same license, State changes	Liability, Warranty
GNU LGPLv3	Commercial use, Distribution, Modification, Patent use, Private use	Disclose source, License and copyright notice, Same license (library), State changes	Liability, Warranty
Mozilla Public License 2.0	Commercial use, Distribution, Modification, Patent use, Private use	Disclose source, License and copyright notice, Same license (library)	Liability, Trademark use, Warranty
Apache License 2.0	Commercial use, Distribution, Modification, Patent use, Private use	License and copyright notice, State changes	Liability, Trademark use, Warranty
MIT License	Commercial use, Distribution, Modification, Private use	License and copyright notice	Liability, Warranty
Boost Software License 1.0	Commercial use, Distribution, Modification, Private use	License and copyright notice for source	Liability, Warranty
The Unlicense	Commercial use, Distribution, Modification, Private use		Liability, Warranty

Table 7.1: Typical licenses, associated with permissions, conditions and limitations

	Copyright	Patent	Trademark
What's Protected?	Original works of authorship, such as books, articles, songs, photographs, sculptures, choreography, sound recordings, motion pictures, and other works	Inventions, such as processes, machines, manufactures, compositions of matter as well as improvements to these	Any word, phrase, symbol, and/or design that identifies and distinguishes the source of the goods of one party from those of others
Requirements to be Protected	A work must be original, creative and fixed in a tangible medium	An invention must be new, useful and non-obvious	A mark must be distinctive (i.e., that is, it must be capable of identifying the source of a particular good)
Term of Protection	Author's life plus 70 more years.	20 years	For as long as the mark is used in commerce
Rights Granted	Right to control the reproduction, making of derivative works, distribution and public performance and display of the copyrighted works	Right to prevent others from making, selling using or importing the patented invention	Right to use the mark and to prevent others from using similar marks in a way that would cause a likelihood-of-confusion about the origin of the goods or services.

Table 7.2: Differences among Patents, Copyrights and Trademarks

2. Design Patents: Cover the ornamental aspects of an article, such as its shape or configuration. Examples could include the iconic Coca-Cola bottle or the sleek design of a modern chair.

3. Plant Patents: Protect new and distinct varieties of asexually reproduced plants. This incentivizes innovation in the agricultural sector by securing exclusive rights for novel plant strains.

Obtaining a patent involves a multi-step process:

1. *Conception*: The inventor conceives a new idea for a product, process, or design.

2. *Disclosure*: The invention is documented through detailed drawings, written descriptions, and potentially prototypes.

3. *Search and Patentability Analysis*: A thorough search investigates existing patents to determine novelty (being unlike anything existing) and non-obviousness (not an obvious improvement on existing inventions).

4. *Patent Application Filing*: A formal patent application is drafted, including detailed descriptions, claims (defining the scope of protection), and drawings. Fees are

associated with filing.

5. *Examination*: Patent offices like the United States Patent and Trademark Office (USPTO) meticulously examine the application to ensure it meets all legal requirements. This stage may involve communication between the examiner and the inventor/patent attorney to clarify details or address potential rejections.

6. *Issuance (or Rejection)*: Upon successful examination, the patent office grants the patent, providing the inventor with exclusive rights for a specific period (typically 20 years in the US). If rejected, the applicant can appeal or make modifications and resubmit the application.

Patents are legal documents, and infringement occurs when someone makes, uses, sells, or offers to sell a patented invention without permission. Patent holders can take legal action against infringers to prevent further infringement and potentially recover damages.

However, patents are not absolute. There are exceptions, like fair use for research or experimentation, and challenges to patent validity based on prior art (existing knowledge) or obviousness. Strong patent claims and well-documented disclosures are crucial for robust enforcement.

Patents play a vital role in driving innovation by:

1. *Incentivizing Investment*: The prospect of exclusive rights encourages inventors and companies to invest in research and development.

2. *Promoting Disclosure*: The patent process necessitates detailed disclosure, enriching the public domain with technical knowledge that can be exploited after the patent expires.

3. *Facilitating Commercialization*: Patents allow inventors to control their inventions and negotiate licensing deals, aiding commercialization and technology transfer.

4. *Fueling Economic Growth*: Innovation spurred by patents leads to new products, industries, and jobs, contributing to economic advancement.

The patent landscape is constantly evolving. Debates surround issues like software patentability, patent trolls (entities that acquire patents primarily for litigation, not innovation), and the impact of globalization on patent enforcement. As technology advances, patent offices are challenged to keep pace and ensure an efficient and fair system for all inventors.

Patents have been thoroughly analyzed in many papers. In [314] Himanshu Gupta, Suresh Kumar, Saroj Kumar Roy, and R. S. Gaud discuss patent protection strategies as well as strategies for extending/renewing them in the Pharmaceutical industry. The patent cost is examined in [315] by Bruno van Pottelsberghe and Didier François , while in [316] the need for patent cost reduction is stated. In [317], a law case is described where the need for pre-patent protection is raised, and in [318] a comparison between patents and copyrights is given.

7.2.2 Licenses

In the intricate world of intellectual property (IP), licenses play a crucial role. They act as legal agreements granting permission to use someone else's intellectual property, such as patents, copyrights, trademarks, or trade secrets.

Licenses come in a variety of flavors, each catering to specific needs and granting different levels of permission:

1. Non-Exclusive Licenses: The licensor (owner of the IP) grants permission to the licensee (user of the IP) to use the IP, but retains the right to grant licenses to other parties as well. This arrangement is common for patents and trademarks, where competition can actually stimulate innovation and market growth.

2. Exclusive Licenses: Here, the licensor grants the licensee the sole right to use the IP within a defined territory or for a specific purpose. This provides the licensee with greater control over the IP but comes at a higher cost. Exclusive licenses are often used for copyrights, where the licensee may be granted the sole right to publish or distribute a copyrighted work.

3. Sole Licenses: These represent a rare breed, granting the licensee the exclusive right to use the IP for all purposes, essentially acting like a transfer of ownership with limitations. Sole licenses are typically only granted in exceptional circumstances, as the licensor relinquishes significant control over their IP.

4. Sub-Licenses: With permission from the licensor, the licensee can grant sub-licenses to other parties, allowing them to use the IP within the scope of the original license agreement. This can be a valuable tool for expanding the reach of the IP, especially in situations where the licensee lacks the resources to fully exploit the IP themselves.

A well-crafted license agreement clearly defines the rights and obligations of both parties. Here are some key components to consider:

1. *Parties Involved*: Clearly identify the licensor and licensee, along with their respective contact information.

2. *Licensed IP*: Precisely define the intellectual property being licensed, whether it's a patent, copyright, trademark, trade secret, or a combination thereof.

3. *Scope of the License*: Clearly outline the extent of the permission granted. This includes the specific rights granted (e.g., manufacturing, distribution, sale), any geographical limitations, and the duration of the license term.

4. *Royalties and Fees*: Define the compensation structure. This may involve upfront fees, ongoing royalties based on sales or usage, or a combination of both. The specifics will depend on the type of IP, its value, and negotiation leverage.

5. *Termination Clauses*: Set clear conditions under which the license agreement can be terminated. These may include breach of contract, insolvency, or expiration of the license term.

6. *Confidentiality Provisions*: If the license involves trade secrets or other sensitive information, strong confidentiality clauses are essential to protect the licensor's proprietary knowledge.

Licenses play a significant role in the innovation ecosystem by:

1. *Facilitating Technology Transfer*: Licenses enable inventors and rights holders to share their IP with others, allowing businesses to access and utilize valuable technology without the substantial investment required for in-house development.

2. *Promoting Innovation*: By providing alternative avenues to market their inventions, licenses incentivize inventors and small businesses to invest in Research and Development activities.

3. *Fostering Collaboration*: Licenses can foster collaboration between companies, allowing them to combine their expertise and resources to develop innovative products and services.

4. *Expanding Market Reach*: Licenses allow rights holders to extend the reach of their IP by enabling others to exploit it in new markets or applications.

The landscape of licensing can vary across international borders. Differences in intellectual property laws and regulations require careful consideration when negotiating international licensing agreements. Seeking legal counsel with expertise in international IP law is crucial to navigate these complexities.

Regarding licenses, and especially open source ones, there are a lot of related studies. We list here only a few of them as reference. In [319] a survey is presented explaining When and Why Developers Adopt and Change Software Licenses. In [320] Free and Open Source Software Licenses are described, while in [321] the differences between Open Source Software and Proprietary Software are presented. License Selection and Comparison is performed in [322] as well as Open Source Licenses and the Creative Commons Framework. Finally, The Role of Software Licenses in Open Architecture Ecosystems is designated in [323].

7.3 The Tentative-patent framework

The current patent framework has received criticism on the grounds of unfairness and reduced inclusiveness. The patent creation cost is considerable and demotes the potential of all interested parties to partake the process. This creates unfairness, since less financially robust parties are effectively excluded. The costs for obtaining a patent concern different aspects of the procedure, and the required amounts vary by country, while typically patents offering more global protection are usually associated with higher costs. Indicatively, patent costs may arise from the following aspects [324]:

1. Patent office fees during the application process,

2. Patent attorney fees for legal advice and preparation of the application,

3. Patent protection maintenance fees.

The “tentative-patent" framework proposed in this thesis aims to address issues stemming from the increased cost of patents. This framework includes a filing procedure, which is followed by a basic comparison with existing patents, sourced from patent databases. If no matches are identified, the applicant is granted a “tentative patent", which offers to the inventor a protection status on the invention. The tentative patent may be revoked if the originality claims are shown to be not adequately grounded. This scheme can be realized at a much lower cost than the current patenting scheme, giving the opportunity to any interested party to obtain tentative patents, without seeking or obtaining any external financial support, which might include hypothecating some IP rights/benefits to secure the required funds.

An additional issue with the current patent framework is that it offers no protection to ideas, since only inventions are allowed to be patented. However inventions require some form of implementation, and parties that have conceived ideas but either (a) do not have enough funds and/or (b) lack technical skills and infrastructure to proceed with an implementation are not adequately protected in the process of materializing their ideas in the form of patentable inventions. More specifically, in the process of fund acquisition and/or securing of technical infrastructure or expertise, the party that has conceived the idea must disclose some aspects of it to potential funders or technical partners, while at that stage their intellectual property rights on the idea are not adequately secured. A level of protection can be achieved by having negotiating parties (the conceptor of the idea and potential funders and/or technical expertise providers) enter mutual agreements through which (a) the IPR of the conceptor of the idea are recognized, (b) potential funders and/or technical expertise providers agree to refrain from arrogating the conceptor's idea and (c) potential funders and/or technical expertise commit to not disclose any information regarding the conceptor's idea. However, mutual agreements of this form have a number of shortcomings, including:

- the need to create these agreements for each distinct negotiation, incurring time delays and administrative burden,
- the lack of a universal-wide and transparent framework regarding the terms and conditions governing these agreements and the level of protection they offer, necessitating thus substantial legal expertise on the side of the creator; on the other hand, potential funders and/or technical expertise providers typically have access to legal advise as a matter of their standard operating procedures,
- a power imbalance in the negotiation, since potential funders and/or technical expertise providers have considerably more influence, resources, or leverage than the conceptor, and therefore negotiations may lead to unfair outcomes and/or entail unequal bargaining power.

The same concerns apply to theoretical work, which can be used as basis for inventions at a later stage.

In the proposed tentative-patent framework, the ideas (proven theoretical work) can be tentatively-patented, widening the scope of the protection offered by the existing patent system. In this case, the patent application should describe in an adequate level of detain the possible implementation(s) of the idea or the theoretical work in a particular domain. If the idea/theoretical work is implementable in different areas it is up to the author to provide relevant descriptions, in order to ensure the protection level stemming from the tentative patent.

Once a tentative-patent is obtained, the work covered by the tentative-patent, should be treated as patented. Naturally, inventions, ideas or theoretical work covered by tentative patents cannot be patented or tentatively patented by other parties, while the tentative patent is valid. Parties are given the opportunity to challenge the patent, asserting claims that the work is not original, and in this case the challenge is reviewed and decided upon. If the challenge is successful, then the tentative-patent status is revoked and interested parties may use the concepts therein, including the potential to apply themselves for a tentative patent or a patent. Improper use of inventions, ideas or theoretical work covered by tentative patents corresponds to the infringement of the conceptor's rights, and incurs appropriate criminal and/or civil penalties and sanctions.

Summarizing, the tentative-patent framework proposed in this section addresses the three aforementioned issues. More specifically the tentative patent framework tackles aspects related to:

- the patent cost; more specifically, no costs are charged for patent office applications, or these costs can be substantially reduced, owing to the more lightweight processing of applications associated with tentative patents. Additionally, the streamlining of the operations and standardization of application forms is expected to reduce or eliminate attorneys and maintenance/renewal fees.
- the recognition of the idea and its creator, while at the same time providing a framework for protecting ideas until they are implemented and,
- the inclusion of innovative theoretical work as potential grantees of tentative patents.

Notably, a conceptor that has acquired a tentative patent may subsequently apply for a “full" patent for the same concept or an implementation of it.

The workflow of the tentative-patent framework is illustrated in figure 7.1 :

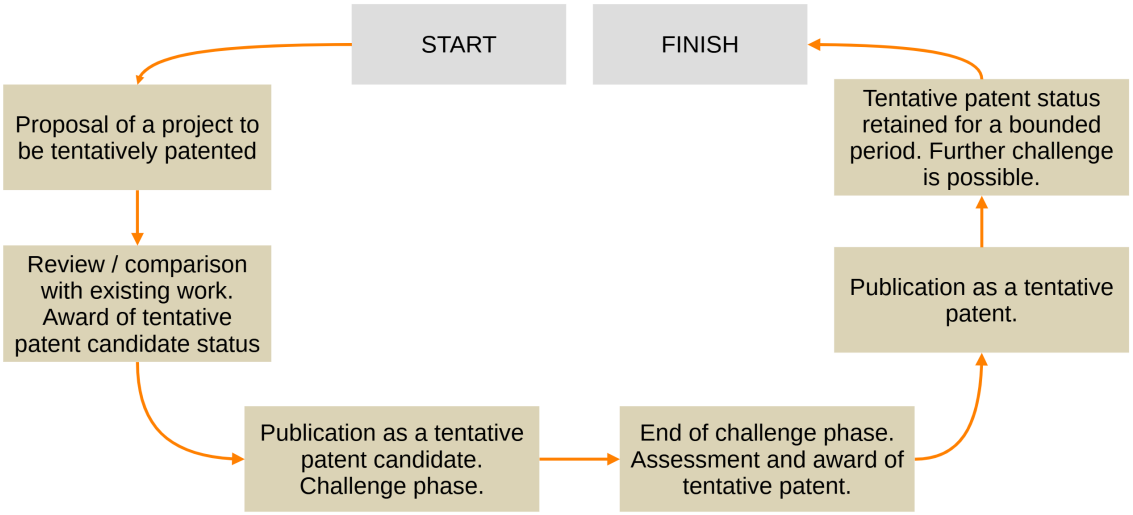

Figure 7.1: The tentative-patent workflow.

In more detail the tentative patent framework encompasses the following steps:

1. Filing of a proposal of a project to be tentatively patented.
2. The proposal is assessed. A basic comparison with existent work is performed and, if no matches with existing patents are identified, the project / product is awarded a *tentative patent candidate* status.
3. The project / product description is uploaded on the internet and published as a *tentative-patent candidate*, for a specified period. During this period, the innovativeness of the project / product can be challenged by any interested party. Filed challenges are assessed, and an assessment concludes that the innovativeness claims of the project collide with previously patented ones, the tentative-patent candidate is revoked.

4. After the challenge phase has concluded, if either (a) no challenges have been submitted or (b) all submitted challenges have been rejected, then the project/product is awarded a tentative patent.

5. The tentatively patented project/product is published as a tentative patent, in a relevant repository.

6. The tentative-patent status is retained for a bounded period. During this period, the patent may be further challenged. Challenges submitted at this stage are processed as described in step 3.

7.4 The versatile license framework

In the software engineering domain, artifacts (libraries, modules, applications or systems) are accompanied by respective licenses. The exact type of license, which mandates the terms and conditions under which the software artifact can be used, as well as liabilities and other aspects pertaining to the usage of the artifact are chosen by the creator of the software. In some cases, however, the choice of the license that best suits the creator's intentions and the nature of the software artifact is not clear from the beginning. Under this viewpoint, the license that accompanies a software artifact may need to change. On the other hand, users of the software artifact may depend on the warranties provided to them by the license that has been published, and changes to these provisions may adversely affect their operations. In order to tackle this issue, we propose a *versatile license framework*, which allows licensing terms to change and evolve, while safeguarding the rights of users that depend on the previously published license.

In general, the domain of projects licensing is dynamic and evolves in numerous ways. Currently, a variety of licenses of different types (e.g. permissive or restrictive) are available, from which a developer may choose for his/her project. However, these licenses get updated over the time (*license versioning*) and newer versions are usually adapted by the projects, while at the same time more licenses are created and made available, expanding this way the license choice set. Developers may also craft their own licenses for their projects, albeit this choice is mostly exercised by enterprises which have specialized and organized legal departments, while individuals typically resort to the use of standard licensing templates. For the case that no license assignment is required, there is the no-license specification (e.g. the Unlicense in open source), with its properties defined. In addition, when a license is chosen for a project, this license remains effective until the project is revoked or until the license is modified. Currently, there are frameworks like OSF (Open Science Framework) which makes possible to make a project public attaching a license, either an existing one or a new one [325]. As noted above however, the license is chosen once and attached permanently (or until the project is revoked), with no provisions for revisions. Under the versatile license framework, the minimum duration of the license effect is specified, and during this period no changes to the licensing model are permitted. When this period expires, the license assignment may be reconsidered/re-evaluated (e.g. to be upgraded to a newer version). It is noted here that cases of re-licensing of software have been witnessed in the past (c.f. Wikipedia page [326]; some notable cases are the re-licensing of the GNU TLS project [327] and the re-licensing of the MariaDB database [328]); however, the proposed versatile licensing framework systematizes the re-licensing process, providing safeguards to warrant the rights of the licensees and giving appropriate directions to licensors.

It is important to note here that license updates to a more recent version or to another license, can have direct of indirect consequences for the users who usually assume a non-modifiable license for a specific project on which they could be dependent. For example, an update of license from a permissive type to a more restrictive one, could impose additional costs or even revocation of the dependent project, while an update from a restrictive type to a more permissive one, may have consequences making the dependent project less competitive, as the sub-projects usage could be easily replicated. In any case, the versatile-license framework determines a mandatory license period which ensures the license conditions for a predefined period.

Currently, there is no location or repository where potential users of software artifacts can be informed about the license period of the different projects available. The proposed versatile licensing framework mandates such a provision, allowing users to compare the available projects in respect to their features and licenses, and co-evaluate these aspects in conjunction with the mandatory validity period of the licenses. The implementation of such a repository would also enable users to register to receive updates when licensing changes occur to software artifacts of their choice, or when the validity period of license assignments to these artifacts is about to expire. These provisions facilitate artifact choice and license management on behalf of artifact users. This is of particular importance, especially considering that license updates to a software artifact may result in the need to re-evaluate the licensing model of projects utilizing this artifact, due to potential incompatibilities of updated terms with obligations stemming from licenses related to other artifacts used in the project [329]. The versatile license framework, through ensuring the minimum effective period of the licenses of the individual artifacts and through the provisions for remaining under the previous license terms, ensures stability in terms and conditions of using software and a smooth transition where appropriate.

The versatile license framework proposes a *license life-cycle management*, in order to manage re-licensing of projects and products; the proposed provisions can be applied to projects and products either in the open domain, or in the proprietary domain. In more detail, the versatile license framework allows for the following evolution paths, regarding the license that is associated with a software artifact:

1. The *fixed-license type*: The project cannot be re-licensed, i.e. the initial license assignment remains fixed and cannot be modified.

2. The *version-re-license type*: The project can be re-licensed only to migrate under a new version of the effective license.

3. The *permissive re-license type*: The project can be re-licensed only to a more open/permissive license. Among two licenses L_1 and L_2, L_2 is considered more open/permissive than L_1 iff all the following conditions hold:

   i) L_2 reserves for the users of the artifact at least all the rights that are reserved by L_1,

   ii) L_2 does not impose for the users of the artifact any additional restrictions than those specified by L_1,

   iii) L_2 mandates for the licensor at least the same obligations that are mandated by L_1.

4. The *any direction re-license type*: The project can be re-licensed towards any license, including the case of moving from a more permissive to more a restrictive type. In

this case, the license to use a different implementation of the latest functionality of the software artifact should be automatically assumed, in order to avoid the creation of usage gaps for existing usages.

The specification and public availability of the evolution path of the license can provide valuable information for potential users of the software. For instance, when selecting a software library on which a project will be based among two candidates, if one candidate library is guaranteed to always be licensed under some free license, while the second one is not, this can be a key element towards the final decision.

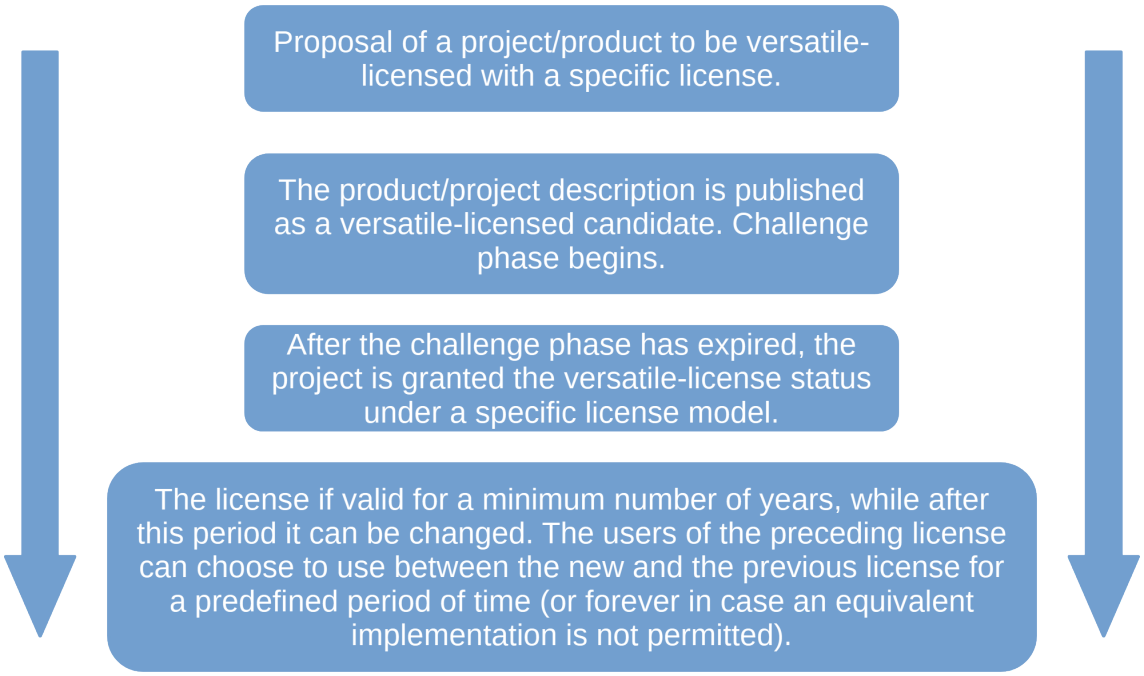

Figure 7.2: The versatile-license workflow.

The flow diagram in figure 7.2 presents the different stages that of the versatile license framework, including the steps that are followed for an artifact to enter the versatile license framework and the steps for (re)assigning a specific license. In more detail, the workflow is as follows:

1. During the first phase, the creator of the project proposes an initial license assignment for their work, also specifying the minimum period that this license assignment will be fixed and unchangeable. Furthermore, the creator of the project specifies the evolution path associated with the licensed artifact.

2. During the second phase, the description of the implementation of the project, the license evolution path and the initial license selection is uploaded on the internet in order to be reviewed and challenged by anyone, for a pre-determined period of time. This challenge phase allows interested parties to designate cases where the project is allegedly not a new one, but essentially constitutes a fork/copy of a previous project which had been released under a different license and/or license upgrade path, and the new licensing settings are not compatible with the previous ones. For instance, if a project was initially released under GPLv3 and a permissive re-license type, releasing a fork of this project (possibly slightly modified) under a proprietary license

cannot be accepted, since this practice effectively violates the terms of the initial licensing model.

3. During the third phase, the project is granted admission to the versatile licensing framework, and it is assigned the initial license (which will not be changed for a minimum number of years). Additionally, the project license assignment is posted on the internet, making it thus effectively accessible by interested parties.

4. During the fourth phase, the chosen license takes effect. The initial license assignment remains effective for the specified minimum number of years, while after this period it can be changed, provided that the change is allowed by the license evolution path. When a license is changed, existing users of the previous license can choose whether they will remain in the old license, or accept the new one. The choice of the existing users may be accompanied with a specification of the time period for which their choice remains in effect. Existing users may opt to remain under the old license for an indefinite amount of time, in case they consider the new license as being not satisfactory for them.

7.5 Conclusions and Future work

In this chapter, a tentative patent framework is proposed for an object where the cost is minimized, thus making this process accessible to anyone who is interested in following it. In addition, there is a reference to the existing licenses for the use of the software and a new versatile license procedure is proposed, which gives more flexibility both in the choice of the license and provides more information and guarantees about its duration and the effective terms of use to the users.

Regarding future work in the area, the following dimensions will be considered:

1. the implementation of the proposed frameworks and the related tools.

2. the assessment of the proposed frameworks.

Chapter 8

Conclusions - Future work

Application testing and test management

First chapter In the first chapter we have presented the Additional Testsuite Framework, which upgrades the classic one-to-one mapping between a program version and a statically specified corresponding testsuite into a dynamic testsuite specification, where tests to be included are designated through declarative specifications that are evaluated on-the-fly to derive the set of tests that need to be applied on the particular case. Developers create tests and annotate them appropriately to designate the software artifact/version to which they apply, the deployment environment in which they will be used, or any other pertinent aspect that may be related to testing; then upon test execution, the AT framework instrumentation extracts the test annotations and the test execution specification (which includes information on the deployment environment, software version, features included etc.) to automatically determine which tests should be applied in the particular case. The AT Framework can support all the types and levels of testing, such as unit testing, component testing and integration testing. Furthermore, the Additional Testsuite Framework offers a number of features, including multi-configuration testing, dynamic program builds, test-base software development, custom software builds and source code analysis support.

The AT Framework undertakes a different point of view in the area of testsuites. This approach upgrades the status of testsuites to “first class citizens" in the software engineering domain, rather than considering them as “subordinates" of software artifacts. The AT Framework is generic, and can be applied for any software program written in any software programming language, and can accommodate any type of testing procedure.

The AT Framework has been implemented and has been applied in a number of real-world software cases, notably including the JBOSS server, Spring Boot, Active MQ and so forth. The results are reported in section 1.6 and demonstrate that the AT Framework improves the test code base maintenance process, eliminating the need for multiple code bases and test method duplication, as well as the probability of inconsistencies due to imperfect update of copies and the extra effort that these updates incur. A list of cases in which the AT Framework has been successfully applied can be found at [78]. A tutorial covering the installation and use of the AT framework is available at [86].

In our future work we will explore additional potential of the AT Framework, including the implementation of support for model-based testing methods [87] and techniques for intermittent fault detection [88], as well as the implementation of tools for cross-analyzing test annotations and code versions to check for mis-annotated tests, the existence of which may lead to the omission of some tests in certain versions, deployment environments

or features. Combination of model based testing and test based source code generation iteratively will be studied. Generation of statistics regarding code usage in relation to the available tests and exploitation of these statistics in the context of software engineering tasks, such as code refactoring and optimization, will be also considered. Finally, the combination of the AT framework with works on test case prioritization [89] will be examined.

Second chapter In the second chapter we introduced the ATF/MA framework, which can be used to support the testing of mobile applications, meeting the real-world requirements for testing multi-version, multi-configuration, multi-device apps, and furthermore underpinning the process of testing feature-based builds and mobile apps exhibiting dynamic behavior. The ATF/MA framework extends the capabilities of the Additional Testsuite Framework [131] considering the particularities of the mobile application domain, effectively tackles the issues related with device fragmentation and change of the mobile ecosystem and reduces the complexity of test management and the workload of developers and testers.

The ATF/MA framework provides the necessary instrumentation and procedures to assist testers in setting up, developing and maintaining the test code base. In utilizes an annotation-based approach to enable a declarative specification of the association between individual tests and the pertinent builds or environment and deployment configurations. The effectiveness of the ATF/MA framework has been demonstrated through its application on a number of test cases. Similarly to the Additional Testsuite Framework [131], ATF/MA is applicable on any app written in any programming language, including both native languages such as Java and Objective-C, as well as cross-platform languages such as Flutter [138]. Moreover, the ATF/MA framework may serve all types of tests, including unit testing, integration testing, functional testing, performance testing etc.

In our future work we will explore the incorporation of model-based testing methods [87] and techniques for intermittent fault detection [88]; this is particularly important for mobile apps, since conditions which trigger the manifestation of intermittent faults may be correlated with aspects of the mobile app environment, such as change of network connectivity or switching between power consumption modes. Taking into account that for mobile apps, the number of tests that need to be conducted to ensure seamless operation is higher than traditional applications, due to the complexities of the mobile ecosystem and environment, the incorporation of test case prioritization [89] will be considered. Finally, in order to assist testers for identifying tests that are executed for each particular build and validating test annotations, a relevant tool will be implemented.

Third chapter In the third chapter, we have presented a software vulnerability management framework which supports all the stages of a pipeline for the management of IoT platform software vulnerabilities. More specifically, the framework supports (a) the configuration of software to include only the necessary features, (b) the execution of security-related tests and the compilation of platform-wide software vulnerability lists, (c) the estimation of the impact and the associated fixing cost for each vulnerability, and (d) the prioritization of vulnerability addressing (considering the impact of each vulnerability) the associated technical debt for its remediation and the available security budget.

The work presented in this chapter advances the state-of-the-art by (i) proposing a statistics-based method for the estimation of impact of detected vulnerabilities, (ii) proposing an integer programming-based algorithm for prioritizing security fixes with the goal of minimizing the residual risk level, and (iii) proposing a comprehensive framework for

the security analysis of platform software which formulates proposals on the prioritization of security issue addressing, taking into account all the aspects (a)–(d) listed in the previous paragraph.

Our future work will focus on the incorporation of dynamic security code analysis and attack graphs into the workflow, as well as the refinement of vulnerability impact estimation.

Fourth chapter In the fourth chapter, we have presented an approach for automating the task of minimizing the risk level of IoT systems that is owing to the vulnerabilities of libraries required by software artefacts. The proposed method exploits knowledge on which libraries provide equivalent implementations of the same functionalities, and when some library is utilized, the system identifies candidate libraries realizing the same functionality, automatically assesses the risk level that corresponds to each library implementation and finally selects the library exhibiting the minimum risk level to bundle into the executable software artefact. Additionally, the risk level of candidate implementations is constantly monitored for new vulnerability identifications or fixes in the implementations, triggering new risk assessments and producing new executables as appropriate. The proposed methodology can be used in IoT platform deployment to minimize the software-rooted risk level.

In our future work we will consider methods that will facilitate the creation and update of the library functional equivalence database, including the introduction of a compatibility/abstraction layer which will be used on top of functionally-equivalent libraries, in the same fashion that Hibernate [223] provides a compatibility/abstraction on top of database drivers. Methods for the more accurate estimation of the risk level of libraries will be explored, including the matching of the vulnerabilities whose impact needs to be assessed against other vulnerabilities of known impact, either at textual description level or at code level. Finally, the use of hotswapping mechanisms for the in-place update of the deployed and running executables will be investigated.

Intermittent fault detection

Fifth chapter In the fifth chapter we presented a methodology for intermittent fault detection that is based on the identification of suspicious shared variable access patterns in the code execution traces. Execution traces are generated using the JPF tool, which has been enhanced by a customized listener, while the suspicious access patterns that are searched for correspond to well-known parallel programming hazards. Out method has been shown to be capable of detecting intermittent faults that evade detection when other methods are used, while on the other hand introducing some false positives. In this sense, the programmer is asked to review the potential intermittent fault root causes and accept or reject them. In order to leverage the efficiency of the proposed method, we have introduced optimization methods which can exploit structural properties of the code, such as thread independence and thread subtree isolation, as well as parallel hardware capabilities. Our experiments on optimizations exploiting structural properties of the code have demonstrated that significant performance gains can be reaped.

In the context of our future work we plan to examine the following dimensions:

1. fully implement and evaluate the optimization method for exploiting parallel hardware capabilities.

2. take into account dependencies between global and local variables, which are established via assignment statements.

3. Identify and evaluate additional dependency rules.

4. study the computational complexity of the proposed algorithm, computing a theoretical upper bound for the number of possible execution paths that need to be explored.

Sixth chapter In the sixth chapter we presented a novel method for intermittent fault detection. The proposed approach examines value dependencies between shared variables and related access sequences, to identify suspicious patterns, and the relevant code fragments are correspondingly flagged and presented to developers for further scrutiny. The algorithm introduced in this chapter extends previous work by considering not only direct value dependencies between shared variables, but additionally indirect ones, where local variables are involved in the value propagation between shared variables. The proposed approach augments the intermittent fault detection capability of previously published algorithms, allowing for identification of faulty code which would evade detection by previous methods. Additionally, no false positives were introduced.

The performance of the proposed algorithm has been experimentally evaluated considering runtime and memory footprint, and compared to that of previous approaches. Based on the experiments, it is shown that the algorithm is feasible, exhibiting enhanced detection capabilities as compared to baseline algorithms, requiring, however, increased processing time or memory resources, depending on the setting used (memory-based or database-oriented storage).

The proposed method can identify intermittent faults that are reflected as suspicious variable access patterns. There do exist classes of faults that are beyond the scope of the proposed work and hence cannot be identified, including faults owing to malfunctions of the underlying hardware, resource depletion, changes to memory locations (mapping to variables) by hardware components (e.g. cases of memory-mapped input/output) and lost interrupt signals. In our future work we will consider the accommodation of methods that would enable the detection of intermittent faults owing to these causes.

Our future work will focus on the optimization of the algorithm performance, the identification and evaluation of additional dependency rules (e.g. rules expressing dependencies between software variables and memory locations directly modified by hardware components, accommodated under the concept of *volatile* variables in C/C++ and Java [300]), as well as the study of the applicability of the proposed algorithm in multi-process environments. Concerning the optimization of the algorithm performance, an experimental version of the listener which optimizes the storage of access sequences in the database is being developed; the current version of the experimental version of the listener can be accessed at [301].

Software licensing

Seventh chapter In the seventh chapter, a tentative patent method is proposed for an object where the cost is minimized, thus making this process accessible to anyone who is interested in following it. In addition, there is a reference to the existing licenses for the

use of the software and a new versatile license procedure is proposed, which gives more flexibility in the choice of the license and provides more information about its duration to the users.

In the context of our future work we plan to examine the following dimensions:

1. ways to implement the frameworks proposed.
2. creation of examples using the proposed frameworks.

Bibliography

[1] B. Beizer, Software Testing Techniques (2nd Ed.), Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[2] IEEE Software and Systems Engineering Standards Committee, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE Std 982.2-1988 (1989) 1–153doi:10.1109/IEEESTD.1989.122630.

[3] A. Monden, M. Tsunoda, M. Barker, K. Matsumoto, Examining Software Engineering Beliefs about System Testing Defects, IT Professional 19 (2) (2017) 58–64. doi:10.1109/MITP.2017.31.

[4] J. Gaur, A. Goyal, T. Choudhury, S. Sabitha, A walk through of software testing techniques, in: 2016 International Conference System Modeling Advancement in Research Trends (SMART), 2016, pp. 103–108. doi:10.1109/SYSMART.2016.7894499.

[5] W. E. Lewis, Software Testing and Continuous Quality Improvement, Third Edition, 3rd Edition, Auerbach Publications, Boston, MA, USA, 2008.

[6] A. Bertolino, Software Testing Research: Achievements, Challenges, Dreams, in: Future of Software Engineering (FOSE '07), 2007, pp. 85–103. doi:10.1109/FOSE.2007.25.

[7] D. Campbell, C. Grevstad, A Tutorial for Make, in: Proceedings of the 1985 ACM Annual Conference on The Range of Computing : Mid-80's Perspective: Mid-80's Perspective, ACM '85, ACM, New York, NY, USA, 1985, pp. 374–380. doi:10.1145/320435.320545.
URL http://doi.acm.org/10.1145/320435.320545

[8] F. P. Miller, A. F. Vandome, J. McBrewster, Apache Maven, Alpha Press, 2010.

[9] C. Russell, PHP Development Tool Essentials, Apress, Berkeley, CA, 2016. doi:https://doi.org/10.1007/978-1-4842-0683-6.

[10] G. Hinkel, NMF: A Multi-platform Modeling Framework, in: A. Rensink, J. Sánchez Cuadrado (Eds.), Theory and Practice of Model Transformation, Springer International Publishing, Cham, 2018, pp. 184–194.

[11] A. M. Hart, Hibernate in the Classroom, J. Comput. Sci. Coll. 20 (4) (2005) 98–100.
URL http://dl.acm.org/citation.cfm?id=1047846.1047860

[12] .NET Persistence API Team, .NET Persistence API, https://www.npersistence.org/index.html (2012).
URL https://www.npersistence.org/index.html

[13] Apache Group, Apache Maven Assembly Plugin, http://maven.apache.org/plugins/maven-assembly-plugin/ (2019).
URL http://maven.apache.org/plugins/maven-assembly-plugin/

[14] Apache Group, Apache Tomcat® Maven Plugin, http://tomcat.apache.org/maven-plugin.html (2013).
URL http://tomcat.apache.org/maven-plugin.html

[15] N. Seth, R. Khare, ACI (automated Continuous Integration) using Jenkins: Key for successful embedded Software development, in: 2015 2nd International Conference on Recent Advances in Engineering Computational Sciences (RAECS), 2015, pp. 1–6. doi:10.1109/RAECS.2015.7453279.

[16] R. Kratky, Documentation based on user stories, https://opensource.com/article/17/6/documentation-based-user-stories (2017).
URL https://opensource.com/article/17/6/documentation-based-user-stories

[17] L. Chen, Q. Li, Automated test case generation from use case: A model based approach, in: 3rd International Conference on Computer Science and Information Technology, Vol. 1, 2010, pp. 372–377. doi:10.1109/ICCSIT.2010.5563772.

[18] A. Panichella, F. M. Kifetew, P. Tonella, Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets, IEEE Transactions on Software Engineering 44 (2) (2018) 122–158. doi:10.1109/tse.2017.2663435.
URL https://doi.org/10.1109/tse.2017.2663435

[19] D. D. Nardo, N. Alshahwan, L. Briand, Y. Labiche, Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system, Software Testing, Verification and Reliability 25 (4) (2015) 371–396. doi:10.1002/stvr.1572.
URL https://doi.org/10.1002/stvr.1572

[20] W. Qiang, F. Chen, L. T. Yang, H. Jin, MUC: Updating cloud applications dynamically via multi-version execution, Future Generation Computer Systems 74 (2017) 254–264. doi:10.1016/j.future.2015.12.003.
URL https://doi.org/10.1016/j.future.2015.12.003

[21] Ubuntu Linux, The Ubuntu lifecycle and release cadence, https://ubuntu.com/about/release-cycle (2020).
URL https://ubuntu.com/about/release-cycle

[22] Microsoft Corporation, Windows lifecycle fact sheet, https://support.microsoft.com/en-us/help/13853/windows-lifecycle-fact-sheet (2020).
URL https://support.microsoft.com/en-us/help/13853/windows-lifecycle-fact-sheet

[23] Red Hat, Red Hat JBoss Middleware Product Update and Support Policy, https://access.redhat.com/support/policy/updates/jboss_notes (2020).
URL https://access.redhat.com/support/policy/updates/jboss_notes

[24] Oracle, Oracle Lifetime Support Policy for Technology Products Guide, http://www.oracle.com/us/support/library/lifetime-support-technology-069183.pdf (2020).
URL http://www.oracle.com/us/support/library/lifetime-support-technology-069183.pdf

[25] Joomla, Joomla! Project Roadmap, https://developer.joomla.org/roadmap.html (2020).
URL https://developer.joomla.org/roadmap.html

[26] R. Ramsauer, D. Lohmann, W. Mauerer, Observing Custom Software Modifications: A Quantitative Approach of Tracking the Evolution of Patch Stacks, in: Proceedings of the 12th International Symposium on Open Collaboration, OpenSym '16, Association for Computing Machinery, New York, NY, USA, 2016. doi:10.1145/2957792.2957810.
URL https://doi.org/10.1145/2957792.2957810

[27] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, R. Tumeng, Test case prioritization approaches in regression testing: A systematic literature review, Information and Software Technology 93 (2018) 74–93. doi:https://doi.org/10.1016/j.infsof.2017.08.014.
URL http://www.sciencedirect.com/science/article/pii/S0950584916304888

[28] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, T. Xie, Dependent-test-aware regression testing techniques, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, 2020. doi:10.1145/3395363.3397364.
URL https://doi.org/10.1145/3395363.3397364

[29] J. Wang, W. Tepfenhart, Formal Methods in Computer Science, Chapman and Hall/CRC, 2019.

[30] S. Chen, J. Huang, Y. Gong, Static Testing as a Service on Cloud, in: 27th International Conference on Advanced Information Networking and Applications Workshops, 2013, pp. 638–642. doi:10.1109/WAINA.2013.257.

[31] F. Asplund, Exploratory testing: Do contextual factors influence software fault identification?, Information and Software Technology 107 (2019) 101–111. doi:https://doi.org/10.1016/j.infsof.2018.11.003.
URL http://www.sciencedirect.com/science/article/pii/S0950584918302325

[32] M. E. Khan, F. Khan, A Comparative Study of White Box, Black Box and Grey Box Testing Techniques, International Journal of Advanced Computer Science and Applications 3 (6) (2012) 22–25.

[33] P. Runeson, A survey of unit testing practices, IEEE Software 23 (4) (2006) 22–29. doi:10.1109/MS.2006.91.

[34] M. Jaffar-ur Rehman, F. Jabeen, A. Bertolino, A. Polini, Testing software components for integration: a survey of issues and techniques, Software Testing, Verification and Reliability 17 (2) (2007) 95–133. arXiv:https://onlinelibrary.wiley.com/doi/

pdf/10.1002/stvr.357, doi:10.1002/stvr.357.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.357

[35] L. Briand, Y. Labiche, A UML-Based Approach to System Testing, Software and Systems Modeling 1 (1) (2002) 10–42. doi:10.1007/s10270-002-0004-8.
URL https://doi.org/10.1007/s10270-002-0004-8

[36] P. Hsia, D. Kung, C. Sell, Software requirements and acceptance testing, Annals of Software Engineering 3 (1) (1997) 291–317. doi:10.1023/A:1018938021528.
URL https://doi.org/10.1023/A:1018938021528

[37] M. Gligoric, L. Eloussi, D. Marinov, Practical Regression Test Selection with Dynamic File Dependencies, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, ACM, New York, NY, USA, 2015, pp. 211–222. doi:10.1145/2771783.2771784.
URL http://doi.acm.org/10.1145/2771783.2771784

[38] J. S. Vetter, B. R. de Supinski, Dynamic Software Testing of MPI Applications with Umpire, in: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, SC '00, IEEE Computer Society, USA, 2000, pp. 51—es.

[39] G. team, Git, https://git-scm.com/ (2021).
URL https://git-scm.com/

[40] A. group, Apache® Subversion®, https://subversion.apache.org/ (2021).
URL https://subversion.apache.org/

[41] G. Downs, Lean-agile acceptance test-driven development, ACM SIGSOFT Software Engineering Notes 36 (4) (2011) 34–34. doi:10.1145/1988997.1989006.
URL https://doi.org/10.1145/1988997.1989006

[42] B. George, L. Williams, A structured experiment of test-driven development, Information and Software Technology 46 (5) (2004) 337–342, special Issue on Software Engineering, Applications, Practices and Tools from the ACM Symposium on Applied Computing 2003. doi:https://doi.org/10.1016/j.infsof.2003.09.011.
URL http://www.sciencedirect.com/science/article/pii/S0950584903002040

[43] T. D. Hellmann, A. Hosseini-Khayat, F. Maurer, Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design, in: Third International Conference on Software Testing, Verification, and Validation Workshops, 2010, pp. 444–447.

[44] C. Wang, F. Pastore, A. Goknil, L. Briand, Z. Iqbal, Automatic generation of system test cases from use case specifications, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ACM, 2015. doi:10.1145/2771783.2771812.
URL https://doi.org/10.1145/2771783.2771812

[45] P. Tonella, Evolutionary testing of classes, ACM SIGSOFT Software Engineering Notes 29 (4) (2004) 119–128. doi:10.1145/1013886.1007528.
URL https://doi.org/10.1145/1013886.1007528

[46] P. McMinn, Search-based software test data generation: a survey, Software Testing, Verification and Reliability 14 (2) (2004) 105–156. doi:10.1002/stvr.294.
URL https://doi.org/10.1002/stvr.294

[47] N. Jain, R. Porwal, Automated Test Data Generation Applying Heuristic Approaches—A Survey, in: Advances in Intelligent Systems and Computing, Springer Singapore, 2018, pp. 699–708. doi:10.1007/978-981-10-8848-3_68.
URL https://doi.org/10.1007/978-981-10-8848-3_68

[48] Bugespy, How much does Software Testing Cost, https://www.bugespy.com/how-much-does-software-testing-cost/ (2020).
URL https://www.bugespy.com/how-much-does-software-testing-cost/

[49] O. Litvinova, How to reduce testing costs in agile projects, https://techbeacon.com/app-dev-testing/how-reduce-testing-costs-agile-projects (2020).
URL https://techbeacon.com/app-dev-testing/how-reduce-testing-costs-agile-projects

[50] E. Papatheocharous, S. Bibi, I. Stamelos, A. S. Andreou, An investigation of effort distribution among development phases: A four-stage progressive software cost estimation model, Journal of Software: Evolution and Process 29 (10) (2017) e1881. doi:10.1002/smr.1881.
URL https://doi.org/10.1002/smr.1881

[51] P. Sotiropoulos, JBoss Server Additional Testsuite, https://github.com/panossot/EAT-1 (2022).
URL https://github.com/panossot/EAT-1

[52] EDUCBA, WebLogic vs JBoss, https://www.educba.com/weblogic-vs-jboss/ (2020).
URL https://www.educba.com/weblogic-vs-jboss/

[53] A. T. Group, Migrating from 8.0.x or 8.5.x to 9.0.x, https://tomcat.apache.org/migration-9.html (2020).
URL https://tomcat.apache.org/migration-9.html

[54] M. Corporation, Deploy an Azure Web App, https://docs.microsoft.com/en-us/azure/devops/pipelines/targets/webapp?view=azure-devops&tabs=yaml (2020).
URL https://docs.microsoft.com/en-us/azure/devops/pipelines/targets/webapp?view=azure-devops&tabs=yaml

[55] AWS, AWS CodeDeploy resources, https://aws.amazon.com/codedeploy/resources/?nc=sn&loc=5 (2020).
URL https://aws.amazon.com/codedeploy/resources/?nc=sn&loc=5

[56] AWS, AWS AppSpec Files, https://docs.aws.amazon.com/codedeploy/latest/userguide/reference-appspec-file.html (2020).
URL https://docs.aws.amazon.com/codedeploy/latest/userguide/reference-appspec-file.html

[57] Kubernetes, Deployments (Kubernetes documentation on controllers), https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

(2020).
URL https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[58] Rancher, Building an Automated Deployment Process with Kubernetes CI/CD, https://rancher.com/learning-paths/building-an-automated-deployment-process-for-kubernetes-with-cicd/ (2020).
URL https://rancher.com/learning-paths/building-an-automated-deployment-process-for-kubernetes-with-cicd/

[59] C. Prehofer, Feature-oriented programming: A fresh look at objects, in: M. Akşit, S. Matsuoka (Eds.), ECOOP'97 — Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 419–443.

[60] OpenLiberty, Feature overview, https://openliberty.io/docs/ref/feature/ (2020).
URL https://openliberty.io/docs/ref/feature/

[61] P. Sotiropoulos, OpenLiberty test cases, https://github.com/panossot/OAT/tree/master/modules/testcases/OpenLiberty (2020).
URL https://github.com/panossot/OAT/tree/master/modules/testcases/OpenLiberty

[62] M. El-Attar, J. Miller, Developing comprehensive acceptance tests from use cases and robustness diagrams, Requirements Engineering 15 (3) (2009) 285–306. doi:10.1007/s00766-009-0088-6.
URL https://doi.org/10.1007/s00766-009-0088-6

[63] E. M. Maximilien, L. Williams, Assessing test-driven development at IBM, in: Proceedings of the 25th International Conference on Software Engineering, 2003, pp. 564–569.

[64] D. Janzen, H. Saiedian, Test-driven development concepts, taxonomy, and future direction, Computer 38 (9) (2005) 43–50.

[65] L. Koskela, Test Driven: Practical Tdd and Acceptance Tdd for Java Developers, Manning Publications Co., USA, 2007.

[66] A. Cauevic, S. Punnekkat, D. Sundmark, Quality of Testing in Test Driven Development, in: Eighth International Conference on the Quality of Information and Communications Technology, 2012, pp. 266–271.

[67] Y. Sui, D. Ye, J. Xue, Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, Association for Computing Machinery, New York, NY, USA, 2012, p. 254–264. doi:10.1145/2338965.2336784.
URL https://doi.org/10.1145/2338965.2336784

[68] D. Yan, S. Yang, A. Rountev, Systematic testing for resource leaks in Android applications, in: IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), 2013, pp. 411–420.

[69] D. Baca, B. Carlsson, K. Petersen, L. Lundberg, Improving software security with static automated code analysis in an industry setting, Software: Practice and Experience 43 (3) (2013) 259–279. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2109, doi:10.1002/spe.2109.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2109

[70] G. Rasool, Z. Arshad, A review of code smell mining techniques, Journal of Software: Evolution and Process 27 (11) (2015) 867–895. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1737, doi:10.1002/smr.1737.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1737

[71] T. Arendt, G. Taentzer, Integration of Smells and Refactorings within the Eclipse Modeling Framework, in: Proceedings of the Fifth Workshop on Refactoring Tools, WRT '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 8–15. doi:10.1145/2328876.2328878.
URL https://doi.org/10.1145/2328876.2328878

[72] E. Torlak, S. Chandra, Effective Interprocedural Resource Leak Detection, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 535–544. doi:10.1145/1806799.1806876.
URL https://doi.org/10.1145/1806799.1806876

[73] B. Pietrzak, B. Walter, Leveraging Code Smell Detection with Inter-smell Relations, in: P. Abrahamsson, M. Marchesi, G. Succi (Eds.), Extreme Programming and Agile Processes in Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 75–84.

[74] G. A. Campbell, P. P. Papapetrou, SonarQube in Action, 1st Edition, Manning Publications Co., USA, 2013.

[75] S. Chiba, Load-Time Structural Reflection in Java, in: Proceedings of the 14th European Conference on Object-Oriented Programming, ECOOP '00, Springer-Verlag, Berlin, Heidelberg, 2000, p. 313–336.

[76] Javassist, Java Programming Assistant, https://www.javassist.org/ (2021).
URL https://www.javassist.org/

[77] J. Evain, Mono.Cecil library, https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/ (2021).
URL https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/

[78] P. Sotiropoulos, C. Vassilakis, Application cases of the Additional Testsuites Framework, https://soda.dit.uop.gr/?q=publications/trs/soda-tr-2020-01 (2022).
URL https://soda.dit.uop.gr/?q=publications/trs/soda-tr-2020-01

[79] WildFly, WildFly home page, https://www.wildfly.org/ (2021).
URL https://www.wildfly.org/

[80] OpenLiberty, OpenLiberty, https://openliberty.io/ (2020).
URL https://openliberty.io/

[81] Spring, Spring Boot web page, https://spring.io/projects/spring-boot (2021).
URL https://spring.io/projects/spring-boot

[82] L. Kopanias, 2nd Chance, https://github.com/Lkop/Project2ndChance (2021).
URL https://github.com/Lkop/Project2ndChance

[83] A. group, ActiveMQ Broker, https://github.com/apache/activemq/tree/master/activemq-broker (2021).
URL https://github.com/apache/activemq/tree/master/activemq-broker

[84] A. group, ActiveMQ Client Tests, https://github.com/apache/activemq (2021).
URL https://github.com/apache/activemq

[85] E. S. Foundation, Eclipse Vertx, https://github.com/vert-x3 (2021).
URL https://github.com/vert-x3

[86] P. Sotiropoulos, EAT Workshop, https://drive.google.com/file/d/1iRT_FoRxj2tOcIPD42mUQpqcbDxuczCy/view (2020).
URL https://drive.google.com/file/d/1iRT_FoRxj2tOcIPD42mUQpqcbDxuczCy/view

[87] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A Survey on Software Fault Localization, IEEE Transactions on Software Engineering 42 (8) (2016) 707–740. doi:10.1109/TSE.2016.2521368.

[88] P. Sotiropoulos, C. Vassilakis, Detection of intermittent faults in software programs through identification of suspicious shared variable access patterns, Journal of Systems and Software 159 (2020) 110455. doi:10.1016/j.jss.2019.110455.
URL https://www.sciencedirect.com/science/article/pii/S0164121219302298?via%3Dihub

[89] H. Jahan, Z. Feng, S. H. Mahmud, P. Dong, Version specific test case prioritization approach based on artificial neural network, Journal of Intelligent and Fuzzy Systems 36 (6) (2019) 6181–6194. doi:10.3233/JIFS-181998.
URL https://doi.org/10.3233/JIFS-181998

[90] N. G. Berihun, C. Dongmo, J. A. V. der Poll, The Applicability of Automated Testing Frameworks for Mobile Application Testing: A Systematic Literature Review, International Journal of Web Engineering and Technology (May 2023). doi:10.3390/computers12050097.
URL https://doi.org/10.3390/computers12050097

[91] F. Pecorelli, G. Catolino, F. Ferrucci, A. De Lucia, F. Palomba, Testing of Mobile Applications in the Wild: A Large-Scale Empirical Study on Android Apps, in: Proceedings of the 28th International Conference on Program Comprehension, ICPC '20, ACM, 2020. doi:10.1145/3387904.3389256.
URL http://dx.doi.org/10.1145/3387904.3389256

[92] J. Kaasila, D. Ferreira, V. Kostakos, T. Ojala, Testdroid: automated remote UI testing on Android, in: Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia, MUM '12, Association for Computing Machinery, New York, NY, USA, 2012. doi:10.1145/2406367.2406402.
URL https://doi.org/10.1145/2406367.2406402

[93] F. Lettner, C. Holzmann, Automated and Unsupervised User Interaction Logging as Basis for Usability Evaluation of Mobile Applications, in: Proceedings of the 10th International Conference on Advances in Mobile Computing & Multimedia, MoMM '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 118–127. doi:10.1145/2428955.2428983.
URL https://doi.org/10.1145/2428955.2428983

[94] W. Kluth, K.-H. Krempels, C. Samsel, Automated Usability Testing for Mobile Applications, in: Proceedings of the 10th International Conference on Web Information Systems and Technologies, SCITEPRESS - Science and and Technology Publications, 2014. doi:10.5220/0004985101490156.
URL https://doi.org/10.5220%2F0004985101490156

[95] J. Grigera, A. Garrido, J. M. Rivero, G. Rossi, Automatic detection of usability smells in web applications, International Journal of Human-Computer Studies 97 (2017) 129–148. doi:10.1016/j.ijhcs.2016.09.009.
URL https://doi.org/10.1016/j.ijhcs.2016.09.009

[96] P. Harms, Automated Usability Evaluation of Virtual Reality Applications, ACM Trans. Comput.-Hum. Interact. 26 (3) (apr 2019). doi:10.1145/3301423.
URL https://doi.org/10.1145/3301423

[97] ISO/IEC, IEEE, ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary, ISO/IEC/IEEE 24765:2017(E) (2017) 1–541doi:10.1109/IEEESTD.2017.8016712.

[98] H. Alsolai, M. Roper, A systematic literature review of machine learning techniques for software maintainability prediction, Information and Software Technology 119 (2020) 106214. doi:10.1016/j.infsof.2019.106214.
URL https://doi.org/10.1016/j.infsof.2019.106214

[99] W. W. Everett, J. D. Musa, J. D. Musa, W. W. Everett, A. F. Ackerman, G. A. Wilson, Software Reliability Engineering, in: J. Marciniak (Ed.), Encyclopedia of Software Engineering, Wiley, 2002. doi:10.1002/0471028959.sof327.
URL https://doi.org/10.1002/0471028959.sof327

[100] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, J. Liu, Dictionary Learning Based Software Defect Prediction, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Association for Computing Machinery, New York, NY, USA, 2014, p. 414–423. doi:10.1145/2568225.2568320.
URL https://doi.org/10.1145/2568225.2568320

[101] L. Qiao, X. Li, Q. Umer, P. Guo, Deep learning based software defect prediction, Neurocomputing 385 (2020) 100–110. doi:10.1016/j.neucom.2019.11.067.
URL https://doi.org/10.1016/j.neucom.2019.11.067

[102] P. Jung, S. Kang, J. Lee, Automated code-based test selection for software product line regression testing, Journal of Systems and Software 158 (2019) 110419. doi:10.1016/j.jss.2019.110419.
URL https://doi.org/10.1016/j.jss.2019.110419

[103] S. Lity, M. Nieke, T. Thüm, I. Schaefer, Retest test selection for product-line regression testing of variants and versions of variants, Journal of Systems and

Software 147 (2019) 46–63. doi:10.1016/j.jss.2018.09.090.
URL https://doi.org/10.1016/j.jss.2018.09.090

[104] E. Whiting, Granular Modeling of User Experience in Load Testing with Automated UI Tests, in: 2021 International Conference on Data and Software Engineering (ICoDSE), IEEE, 2021. doi:10.1109/icodse53690.2021.9648440.
URL https://doi.org/10.1109/icodse53690.2021.9648440

[105] M. H. Moghadam, M. Saadatmand, M. Borg, M. Bohlin, B. Lisper, Machine Learning to Guide Performance Testing: An Autonomous Test Framework, in: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2019. doi:10.1109/icstw.2019.00046.
URL https://doi.org/10.1109/icstw.2019.00046

[106] O. Huerta-Guevara, V. Ayala-Rivera, L. Murphy, A. O. Portillo-Dominguez, DYNAMOJM: A JMeter Tool for Performance Testing Using Dynamic Workload Adaptation, in: Testing Software and Systems, Springer International Publishing, 2019, pp. 234–241. doi:10.1007/978-3-030-31280-0_14.
URL https://doi.org/10.1007/978-3-030-31280-0_14

[107] M. Camilli, A. Guerriero, A. Janes, B. Russo, S. Russo, Microservices integrated performance and reliability testing, in: Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test, ACM, 2022. doi:10.1145/3524481.3527233.
URL https://doi.org/10.1145/3524481.3527233

[108] R. S. Herlim, Y. Kim, M. Kim, CITRUS: Automated Unit Testing Tool for Real-world C++ Programs, in: 2022 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE, 2022. doi:10.1109/icst53961.2022.00046.
URL https://doi.org/10.1109/icst53961.2022.00046

[109] A. Janes, B. Russo, Automatic Performance Monitoring and Regression Testing During the Transition from Monolith to Microservices, in: 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2019. doi:10.1109/issrew.2019.00067.
URL https://doi.org/10.1109/issrew.2019.00067

[110] M. Kreitz, Security by Design in Software Engineering, ACM SIGSOFT Software Engineering Notes 44 (3) (2019) 23–23. doi:10.1145/3356773.3356798.
URL https://doi.org/10.1145/3356773.3356798

[111] M. B. Freitas, V. M. Araújo, J. P. Magalhães, Process SDLC-GDPR: Towards the Development of Secure and Compliant Applications, in: 2023 1st International Conference on Advanced Innovations in Smart Cities (ICAISC), IEEE, 2023. doi:10.1109/icaisc56366.2023.10085308.
URL https://doi.org/10.1109/icaisc56366.2023.10085308

[112] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, A. Pretschner, Security Testing, in: Advances in Computers, Elsevier, 2016, pp. 1–51. doi:10.1016/bs.adcom.2015.11.003.
URL https://doi.org/10.1016/bs.adcom.2015.11.003

[113] J. Li, Vulnerabilities Mapping based on OWASP-SANS: a Survey for Static Application Security Testing (SAST), Computing Research Repository (CoRR) (2020). doi:10.48550/ARXIV.2004.03216.
URL https://arxiv.org/abs/2004.03216

[114] C.-M. Mathas, C. Vassilakis, Reconnaissance, in: N. Kolokotronis, S. Shiaeles (Eds.), Cyber-Security Threats, Actors, and Dynamic Mitigation, 1st Edition, CRC Press, 2021, Ch. 2, pp. 27–80. doi:10.1201/9781003006145-2.

[115] OWASP Foundation, OWASP Web Security Testing Guide, https://owasp.org/www-project-web-security-testing-guide/ (2020).
URL https://owasp.org/www-project-web-security-testing-guide/

[116] B. W. Boehm, J. R. Brown, M. Lipow, Quantitative Evaluation of Software Quality, in: Proceedings of the 2nd International Conference on Software Engineering, ICSE '76, IEEE Computer Society Press, Washington, DC, USA, 1976, p. 592–605.

[117] ISO/IEC 25010, ISO/IEC 25010:2011, Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, ISO/IEC, 2011.

[118] I.-C. Yoon, A. Sussman, A. Memon, A. Porter, Effective and scalable software compatibility testing, in: Proceedings of the 2008 international symposium on Software testing and analysis, ACM, 2008. doi:10.1145/1390630.1390640.
URL https://doi.org/10.1145/1390630.1390640

[119] T. Zhang, J. Gao, J. Cheng, T. Uehara, Compatibility Testing Service for Mobile Applications, in: 2015 IEEE Symposium on Service-Oriented System Engineering, IEEE, 2015. doi:10.1109/sose.2015.35.
URL https://doi.org/10.1109/sose.2015.35

[120] C. Prathibhan, A. Malini, N. Venkatesh, K. Sundarakantham, An automated testing framework for testing Android mobile applications in the cloud, in: 2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies, 2014, pp. 1216–1219. doi:10.1109/ICACCCT.2014.7019292.

[121] J. Cheng, Y. Zhu, T. Zhang, C. Zhu, W. Zhou, Mobile Compatibility Testing Using Multi-objective Genetic Algorithm, in: 2015 IEEE Symposium on Service-Oriented System Engineering, 2015, pp. 302–307. doi:10.1109/SOSE.2015.36.

[122] Q. Naith, F. Ciravegna, Mobile devices compatibility testing strategy via crowdsourcing, International Journal of Crowd Science 2 (3) (2018) 225–246. doi:10.1108/IJCS-09-2018-0024.

[123] P. Tramontana, D. Amalfitano, N. Amatucci, A. R. Fasolino, Automated functional testing of mobile applications: a systematic mapping study, Software Quality Journal 27 (1) (2018) 149–201. doi:10.1007/s11219-018-9418-6.
URL https://doi.org/10.1007/s11219-018-9418-6

[124] A. Méndez-Porras, C. Quesada-López, M. Jenkins, Automated testing of mobile applications: A systematic map and review, in: IBSE 2015 - XVIII Ibero-American Conference on Software Engineering, 2015, pp. 195–208.

[125] N. G. Berihun, C. Dongmo, J. A. Van der Poll, The Applicability of Automated Testing Frameworks for Mobile Application Testing: A Systematic Literature Review, Computers 12 (5) (2023) 97. doi:10.3390/computers12050097.
URL http://dx.doi.org/10.3390/computers12050097

[126] StatCounter, OWASP Web Security Testing Guide, https://gs.statcounter.com/os-market-share/mobile/worldwide (2023).
URL https://gs.statcounter.com/os-market-share/mobile/worldwide

[127] Android Authority, The most popular Android version isn't even Android 12, https://www.androidauthority.com/android-13-version-distribution-may-2023-3332337/ (2023).
URL https://www.androidauthority.com/android-13-version-distribution-may-2023-3332337/

[128] TelemetryDeck, iOS versions market share in October 2023, https://telemetrydeck.com/blog/ios-market-share-10-23/ (2023).
URL https://telemetrydeck.com/blog/ios-market-share-10-23/

[129] GizmoChina, Apple Reveals Usage Statistics Of iOS 16 And iPadOS 16 Ahead Of WWDC 2023, https://www.gizmochina.com/2023/06/03/apple-reveals-usage-statistics-of-ios-16-and-ipados-16-ahead-of-wwdc-2023/ (2023).
URL https://www.gizmochina.com/2023/06/03/apple-reveals-usage-statistics-of-ios-16-and-ipados-16-ahead-of-wwdc-2023/

[130] J. Bo, L. Xiang, G. Xiaopeng, MobileTest: A Tool Supporting Automatic Black Box Test for Software on Smart Mobile Devices, in: Second International Workshop on Automation of Software Test (AST '07), 2007, pp. 8–8. doi:10.1109/AST.2007.9.

[131] P. Sotiropoulos, C. Vassilakis, The additional testsuite framework: facilitating software testing and test management, International Journal of Web Engineering and Technology 17 (3) (2022) 296–334. doi:https://doi.org/10.1504/ijwet.2022.127876.

[132] R. R. Nimbalkar, Mobile Application Testing and Challenges, International Journal of Science and Research 2 (Jul. 2013).

[133] M. E. Joorabchi, A. Mesbah, P. Kruchten, Real Challenges in Mobile App Development, in: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, IEEE, 2013. doi:10.1109/ESEM.2013.9.
URL https://doi.org/10.1109/ESEM.2013.9

[134] S. Zein, N. Salleh, J. Grundy, A Systematic Mapping Study of Mobile Application Testing Techniques, Journal of systems and software 117 (2016) 334—-356. doi:10.1016/j.jss.2016.03.065.
URL https://doi.org/10.1016/j.jss.2016.03.065

[135] P. H. Kuroishi, J. C. Maldonado, A. M. R. Vincenzi, Towards the definition of a research agenda on mobile application testing based on a tertiary study, Information and Software Technology 167 (2024) 107363. doi:10.1016/j.infsof.2023.107363.
URL http://dx.doi.org/10.1016/j.infsof.2023.107363

[136] D. J. Kim, N. Tsantalis, T.-H. Chen, J. Yang, Studying Test Annotation Maintenance in the Wild, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 62–73. doi:10.1109/ICSE43902.2021.00019.

[137] A. K. Jha, S. Nadi, Annotation practices in Android apps, in: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2020, pp. 132–142. doi:10.1109/SCAM51674.2020.00020.

[138] G. Inc., Flutter documentation, https://docs.flutter.dev/ (2024).
URL https://docs.flutter.dev/

[139] G. Inc., SDK Platform release notes, https://developer.android.com/tools/releases/platforms/ (2024).
URL https://developer.android.com/tools/releases/platforms/

[140] G. Inc., People and conversations, https://developer.android.com/develop/ui/views/notifications/conversations/ (2024).
URL https://developer.android.com/develop/ui/views/notifications/conversations/

[141] A. developers, Wi-Fi suggestion API for internet connectivity, https://developer.android.com/develop/connectivity/wifi/wifi-suggest#java (2024).
URL https://developer.android.com/develop/connectivity/wifi/wifi-suggest#java

[142] A. O. S. Project, Add setting that controls network rate limit, https://android-review.googlesource.com/c/platform/packages/modules/Connectivity/+/1955583/10 (2024).
URL https://android-review.googlesource.com/c/platform/packages/modules/Connectivity/+/1955583/10

[143] M. Studio, Mastering Feature-First Architecture: Building Scalable Flutter Mobile Apps, https://mobterest.medium.com/mastering-feature-first-architecture-building-scalable-flutter-mobile-apps-5c706b6e42be (2024).
URL https://mobterest.medium.com/mastering-feature-first-architecture-building-scalable-flutter-mobile-apps-5c706b6e42be

[144] A. developers, Power management, https://developer.android.com/about/versions/pie/power (2024).
URL https://developer.android.com/about/versions/pie/power

[145] P. G. user, adb-wifi-setting-manager, https://github.com/pr4bh4sh/adb-wifi-setting-manager (2024).
URL https://github.com/pr4bh4sh/adb-wifi-setting-manager

[146] P. Sotiropoulos, Source code of a basic mobile application with e-mail address validity checks, https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/testcases/jdkAll/Android/junitbasicsample (2024).
URL https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/testcases/jdkAll/Android/junitbasicsample

[147] P. Sotiropoulos, Test cases of a basic mobile application with e-mail address validity checks, https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/src/main/java/org/example/unit (2024).
URL https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/src/main/java/org/example/unit

[148] P. Sotiropoulos, Source code of a multi-configuration mobile app, https://github.com/EAT-JBCOMMUNITY/EAT/blob/master/modules/src/main/java/org/example/themesample/ (2024).
URL https://github.com/EAT-JBCOMMUNITY/EAT/blob/master/modules/src/main/java/org/example/themesample/

[149] P. Sotiropoulos, Test cases of a multi-configuration mobile app, https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/testcases/jdkAll/Android/themesample (2024).

[150] farizma GitHub user, Original code for the multi-environment mobile app, https://github.com/fari-zma/JKSLinks (2024).
URL https://github.com/fari-zma/JKSLinks

[151] P. Sotiropoulos, Source code of a multi-environment mobile app, https://github.com/EAT-JBCOMMUNITY/EAT/blob/master/modules/src/main/java/org/example/jkslinks/ (2024).
URL https://github.com/EAT-JBCOMMUNITY/EAT/blob/master/modules/src/main/java/org/example/jkslinks/

[152] P. Sotiropoulos, Test cases of a multi-environment mobile app, https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/testcases/jdkAll/Android/jkslinks (2024).
URL https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/testcases/jdkAll/Android/jkslinks

[153] A. G. community, Original code for the multi-feature mobile app, https://github.com/android/testing-samples/tree/main/runner/AndroidTestOrchestratorWithTestCoverageSample (2024).
URL https://github.com/android/testing-samples/tree/main/runner/AndroidTestOrchestratorWithTestCoverageSample

[154] P. Sotiropoulos, Source code of a multi-feature mobile app, https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/src/main/java/org/example/runner (2024).
URL https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/src/main/java/org/example/runner

[155] P. Sotiropoulos, Test cases of a multi-feature mobile app, https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/testcases/jdkAll/Android/testorchestratorwithtestcoveragesample (2024).
URL https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/testcases/jdkAll/Android/testorchestratorwithtestcoveragesample

[156] SenseMusic, Original code for the media payer mobile app, https://github.com/SenseMusic/Sense (2024).
URL https://github.com/SenseMusic/Sense

[157] P. Sotiropoulos, Source code of a media player mobile app, https://github.com/EAT-JBCOMMUNITY/Sense (2024).
URL https://github.com/EAT-JBCOMMUNITY/Sense

[158] P. Sotiropoulos, Test cases of a multi-feature mobile app, https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/testcases/jdkAll/Android/mediaplayer (2024).
URL https://github.com/EAT-JBCOMMUNITY/EAT/tree/master/modules/testcases/jdkAll/Android/mediaplayer

[159] N. limbu, Maximizing Efficiency with Multicapabilities in Appium WebDriverIO for Mobile Automation Testing, https://medium.com/\spacefactor\@m{}limbuneeshen/maximizing-efficiency-with-multicapabilities-in-appium-webdriverio-for-mobile-automation-testing-a9aa20f63cf9 (2023).
URL https://medium.com/@limbuneeshen/maximizing-efficiency-with-multicapabilities-in-appium-webdriverio-for-mobile-automation-testing-a9aa20f63cf9

[160] A. Grau, M. Indri, L. L. Bello, T. Sauter, Industrial robotics in factory automation: From the early stage to the Internet of Things, in: IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society, IEEE, 2017. doi:10.1109/iecon.2017.8217070.
URL https://doi.org/10.1109/iecon.2017.8217070

[161] A. Grau, M. Indri, L. L. Bello, T. Sauter, Robots in Industry: The Past, Present, and Future of a Growing Collaboration With Humans, IEEE Industrial Electronics Magazine 15 (1) (2021) 50–61. doi:10.1109/mie.2020.3008136.
URL https://doi.org/10.1109/mie.2020.3008136

[162] G. R. Barai, S. Krishnan, B. Venkatesh, Smart metering and functionalities of smart meters in smart grid - a review, in: 2015 IEEE Electrical Power and Energy Conference (EPEC), IEEE, 2015. doi:10.1109/epec.2015.7379940.
URL https://doi.org/10.1109/epec.2015.7379940

[163] R. Coppola, M. Morisio, Connected Car, ACM Computing Surveys 49 (3) (2016) 1–36. doi:10.1145/2971482.
URL https://doi.org/10.1145/2971482

[164] R. Hussain, S. Zeadally, Autonomous Cars: Research Results, Issues, and Future Challenges, IEEE Communications Surveys & Tutorials 21 (2) (2019) 1275–1313. doi:10.1109/comst.2018.2869360.
URL https://doi.org/10.1109/comst.2018.2869360

[165] Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030, https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/, accessed: 2023-02-04 (2022).

[166] The Internet of Things: A Movement, Not a Market, https://cdn.ihs.com/www/pdf/IoT-ebook.pdf, accessed: 2023-02-04 (2017).

[167] M. binti Mohamad Noor, W. H. Hassan, Current research on Internet of Things (IoT) security: A survey, Computer Networks 148 (2019) 283–294. doi:10.1016/j.comnet.2018.11.025.
URL https://doi.org/10.1016/j.comnet.2018.11.025

[168] R. F. Ali, A. Muneer, P. D. D. Dominic, S. M. Taib, E. A. A. Ghaleb, Internet of Things (IoT) Security Challenges and Solutions: A Systematic Literature Review, in: Communications in Computer and Information Science, Springer Singapore, 2021, pp. 128–154. doi:10.1007/978-981-16-8059-5_9.
URL https://doi.org/10.1007/978-981-16-8059-5_9

[169] H. HaddadPajouh, A. Dehghantanha, R. M. Parizi, M. Aledhari, H. Karimipour, A survey on internet of things security: Requirements, challenges, and solutions, Internet of Things 14 (2021) 100129. doi:10.1016/j.iot.2019.100129.
URL https://doi.org/10.1016/j.iot.2019.100129

[170] A. E. Omolara, A. Alabdulatif, O. I. Abiodun, M. Alawida, A. Alabdulatif, W. H. Alshoura, H. Arshad, The internet of things security: A survey encompassing unexplored areas and new insights, Computers & Security 112 (2022) 102494. doi:10.1016/j.cose.2021.102494.
URL https://doi.org/10.1016/j.cose.2021.102494

[171] I. S. Evaluators, SOHOpelessly Broken 2.0, https://www.ise.io/casestudies/sohopelessly-broken-2-0/, accessed: 2023-02-04 (2019).

[172] S. Herwig, K. Harvey, G. Hughey, R. Roberts, D. Levin, Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet, in: Proceedings 2019 Network and Distributed System Security Symposium, Internet Society, 2019. doi:10.14722/ndss.2019.23488.
URL https://doi.org/10.14722/ndss.2019.23488

[173] G. Bastos, A. Marzano, O. Fonseca, E. Fazzion, C. Hoepers, K. Steding-Jessen, C. M. H. P. C., I. Cunha, D. Guedes, W. Meira, Identifying and Characterizing Bashlite and Mirai C&C Servers, in: 2019 IEEE Symposium on Computers and Communications (ISCC), IEEE, 2019. doi:10.1109/iscc47284.2019.8969728.
URL https://doi.org/10.1109/iscc47284.2019.8969728

[174] R. Hiesgen, M. Nawrocki, T. C. Schmidt, M. Wählisch, The Race to the Vulnerable: Measuring the Log4j Shell Incident (2022). doi:10.48550/ARXIV.2205.02544.
URL https://arxiv.org/abs/2205.02544

[175] O. group, Open Liberty, https://openliberty.io/, accessed: 2023-02-04 (2023).

[176] O. group, Open Liberty: Feature overview, https://openliberty.io/docs/latest/reference/feature/feature-overview.html, accessed: 2023-02-04 (2023).

[177] A. Al-boghdady, K. Wassif, M. El-ramly, The presence, trends, and causes of security vulnerabilities in operating systems of iot's low-end devices, Sensors 21 (7) (2021). doi:10.3390/s21072329.

[178] A. Kaur, R. Nayyar, A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code, Procedia Computer Science 171 (2019) (2020) 2023–2029. doi:10.1016/j.procs.2020.04.217.

[179] OWASP, OWASP Code Review Guide v2, Tech. rep., OWASP (2017).
URL https://owasp.org/www-project-code-review-guide/

[180] C. M. Mathas, C. Vassilakis, N. Kolokotronis, C. C. Zarakovitis, M. A. Kourtis, On the design of IoT security: Analysis of software vulnerabilities for smart grids, Energies 14 (10) (2021). doi:10.3390/en14102818.

[181] E. Schiller, A. Aidoo, J. Fuhrer, J. Stahl, M. Ziörjen, B. Stiller, Landscape of IoT security, Computer Science Review 44 (2022) 100467. doi:10.1016/j.cosrev.2022.100467.
URL https://doi.org/10.1016/j.cosrev.2022.100467

[182] B. M. Calatayud, L. Meany, A comparative analysis of Buffer Overflow vulnerabilities in High-End IoT devices, 2022 IEEE 12th Annual Computing and Communication Workshop and Conference, CCWC 2022 (2022) 694–701doi:10.1109/CCWC54503.2022.9720884.

[183] de Vicente Mohino, B. Higuera, B. Higuera, S. Montalvo, The Application of a New Secure Software Development Life Cycle (S-SDLC) with Agile Methodologies, Electronics 8 (11) (2019) 1218. doi:10.3390/electronics8111218.
URL https://doi.org/10.3390/electronics8111218

[184] SAFECode, Fundamental Practices for Secure Software Development, Tech. Rep. 3rd, SAFEcode (2018).

[185] A. Rashid, H. Chivers, G. Danezis, E. Lupu, A. Martin, CyBok Version 1.0, Tech. rep., CyBok (2019).

[186] R. Dewhurst, OWASP Static Code Analysis, Tech. rep., OWASP (2023).
URL https://owasp.org/www-community/controls/Static_Code_Analysis

[187] V. Sachidananda, S. Bhairav, N. Ghosh, Y. Elovici, PIT: A Probe Into Internet of Things by Comprehensive Security Analysis, in: 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), IEEE, 2019, pp. 522–529. doi:10.1109/TrustCom/BigDataSE.2019.00076.
URL https://ieeexplore.ieee.org/document/8887378/

[188] S. Samtani, S. Yu, H. Zhu, M. Patton, H. Chen, Identifying SCADA vulnerabilities using passive and active vulnerability assessment techniques, in: 2016 IEEE Conference on Intelligence and Security Informatics (ISI), IEEE, 2016, pp. 25–30. doi:10.1109/ISI.2016.7745438.
URL http://ieeexplore.ieee.org/document/7745438/

[189] D. Geneiatakis, I. Kounelis, R. Neisse, I. Nai-Fovino, G. Steri, G. Baldini, Security and privacy issues for an IoT based smart home, in: 2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), IEEE, 2017, pp. 1292–1297. doi:10.23919/MIPRO.2017.7973622.
URL http://ieeexplore.ieee.org/document/7973622/

[190] D. Overstreet, H. Wimmer, R. J. Haddad, Penetration Testing of the Amazon Echo Digital Voice Assistant Using a Denial-of-Service Attack, in: 2019 SoutheastCon, IEEE, 2019, pp. 1–6. doi:10.1109/SoutheastCon42311.2019.9020329.
URL https://ieeexplore.ieee.org/document/9020329/

[191] D. He, X. Yu, T. Li, S. Chan, M. Guizani, Firmware Vulnerabilities Homology Detection Based on Clonal Selection Algorithm for IoT Devices, IEEE Internet of Things Journal 9 (17) (2022) 16438–16445. doi:10.1109/JIOT.2022.3152364.

[192] I. Kotenko, K. Izrailov, M. Buinevich, Static Analysis of Information Systems for IoT Cyber Security: A Survey of Machine Learning Approaches, Sensors 22 (4) (2022). doi:10.3390/s22041335.

[193] R. Akhilesh, O. Bills, N. Chilamkurti, M. J. M. Chowdhury, Automated Penetration Testing Framework for Smart-Home-Based IoT Devices, Future Internet 14 (10) (2022) 276. doi:10.3390/fi14100276.
URL https://doi.org/10.3390/fi14100276

[194] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, L. Sun, Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation, ISSTA 2022 - Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (2022) 417–428doi:10.1145/3533767.3534414.
URL https://doi.org/10.1145/3533767.3534414

[195] C. Prehofer, Feature-oriented programming: A new way of object composition, Concurrency and Computation: Practice and Experience 13 (6) (2001) 465–501. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.583, doi:10.1002/cpe.583.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.583

[196] P. Zave, Requirements for evolving systems: a telecommunications perspective, in: Proceedings Fifth IEEE International Symposium on Requirements Engineering, IEEE Comput. Soc, 2001. doi:10.1109/isre.2001.948535.
URL https://doi.org/10.1109/isre.2001.948535

[197] S. Apel, D. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines, Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-37521-7.
URL https://doi.org/10.1007/978-3-642-37521-7

[198] G. B. Dantzig, Linear Programming, Operations Research 50 (1) (2002) 42–47. arXiv:https://doi.org/10.1287/opre.50.1.42.17798, doi:10.1287/opre.50.1.42.17798.
URL https://doi.org/10.1287/opre.50.1.42.17798

[199] TechTarget, Top 12 most commonly used IoT protocols and standards, https://www.techtarget.com/iotagenda/tip/Top-12-most-commonly-used-IoT-protocols-and-standards (2022).

[200] SonarQube, Issues, https://docs.sonarqube.org/latest/user-guide/issues/ (2023).

[201] O. group, AMQP v1.0, https://www.amqp.org/sites/amqp.org/files/amqp.pdf (2011).

[202] R. Heydon, Bluetooth low energy, Prentice Hall, Philadelphia, PA, 2012.

[203] C. Bormann, A. P. Castellani, Z. Shelby, CoAP: An Application Protocol for Billions of Tiny Internet Nodes, IEEE Internet Computing 16 (2) (2012) 62–67. doi:10.1109/mic.2012.29.
URL https://doi.org/10.1109/mic.2012.29

[204] J. Yang, K. Sandstrom, T. Nolte, M. Behnam, Data Distribution Service for industrial automation, in: Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies &amp Factory Automation (ETFA 2012), IEEE, 2012. doi:10.1109/etfa.2012.6489544.
URL https://doi.org/10.1109/etfa.2012.6489544

[205] IPCisco, Small Office / Home Office (SOHO) Architecture, https://ipcisco.com/lesson/network-topology-architectures/ (2018).

[206] R. Penz, Ready your home network for IoT, https://robert.penz.name/1341/ready-your-home-network-for-iot/ (2016).

[207] M. Ozkaya, Teaching Design-by-Contract for the Modeling and Implementation of Software Systems, in: Proceedings of the 14th International Conference on Software Technologies, SCITEPRESS - Science and Technology Publications, 2019. doi:10.5220/0007950904990507.
URL https://doi.org/10.5220/0007950904990507

[208] C. Silva, S. Guérin, R. Mazo, J. Champeau, Contract-based design patterns, in: Proceedings of the 15th International Conference on Availability, Reliability and Security, ACM, 2020. doi:10.1145/3407023.3409185.
URL https://doi.org/10.1145/3407023.3409185

[209] B. Wang, N. Z. Gong, Attacking Graph-based Classification via Manipulating the Graph Structure, in: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2019. doi:10.1145/3319535.3354206.
URL https://doi.org/10.1145/3319535.3354206

[210] A. T. A. Ghazo, M. Ibrahim, H. Ren, R. Kumar, A2G2V: Automatic Attack Graph Generation and Visualization and Its Applications to Computer and SCADA Networks, IEEE Transactions on Systems, Man, and Cybernetics: Systems 50 (10) (2020) 3488–3498. doi:10.1109/tsmc.2019.2915940.
URL https://doi.org/10.1109/tsmc.2019.2915940

[211] M. O'Leary, Privilege Escalation in Linux, in: Cyber Operations, Apress, 2019, pp. 419–453. doi:10.1007/978-1-4842-4294-0_9.
URL https://doi.org/10.1007/978-1-4842-4294-0_9

[212] T. Rangnau, R. v. Buijtenen, F. Fransen, F. Turkmen, Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines, in: 2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC), IEEE, 2020. doi:10.1109/edoc49727.2020.00026.
URL https://doi.org/10.1109/edoc49727.2020.00026

[213] G. Zhao, J. Huang, DeepSim: deep learning code functional similarity, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 2018. doi:10.1145/3236024.3236068.
URL https://doi.org/10.1145/3236024.3236068

[214] Apache Group, Apache Log4j™ 2 (2023).
URL https://logging.apache.org/log4j/2.x/

[215] Tinylog, TinyLog 2 (2023).
URL https://tinylog.org/v2/

[216] Logback, Logback Project (2023).
URL https://logback.qos.ch/

[217] MITRE, CVE-2021-44228 (2021).
URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2021-44228

[218] Apache Group, Apache Log4j™ 2 v. 2.14.0 (2020).
URL https://logging.apache.org/log4j/2.x/release-notes/2.14.0.html

[219] FasterXML, Jackson project (2023).
URL https://github.com/FasterXML/jackson

[220] Jenkov.com, GSON (2023).
URL https://jenkov.com/tutorials/java-json/gson.html

[221] P. Koloveas, T. Chantzios, S. Alevizopoulou, S. Skiadopoulos, C. Tryfonopoulos, inTIME: A Machine Learning-Based Framework for Gathering and Leveraging Web Data to Cyber-Threat Intelligence, Electronics 10 (7) (2021) 818. doi:10.3390/electronics10070818.
URL https://doi.org/10.3390/electronics10070818

[222] R. Moeller, Fast Serialization (2023).
URL https://github.com/RuedigerMoeller/fast-serialization

[223] C. Bauer, G. King, G. Gregory, Java Persistence with Hibernate, 2nd Edition, Manning Publications Co., USA, 2015.

[224] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secururе Computing 1 (1) (2004) 11–33. doi:10.1109/TDSC.2004.2.
URL http://dx.doi.org/10.1109/TDSC.2004.2

[225] S. Mukherjee, Architecture Design for Soft Errors, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[226] J. Gray, Why Do Computers Stop and What Can Be Done About It?, Technical Report TR 85.7, Tandem Computers, [Online; accessed 13-July-2019] (1985).
URL https://doi.org/10.6084/m9.figshare.27043420

[227] M. Grottke, K. S. Trivedi, Fighting bugs: remove, retry, replicate, and rejuvenate, Computer 40 (2) (2007) 107–109. doi:10.1109/MC.2007.55.

[228] G. Carrozza, D. Cotroneo, R. Natella, R. Pietrantuono, S. Russo, Analysis and Prediction of Mandelbugs in an Industrial Software System, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 262–271. doi:10.1109/ICST.2013.21.

[229] M. Winslett, Bruce Lindsay Speaks out: On System R, Benchmarking, Life As an IBM Fellow, the Power of DBAs in the Old Days, Why Performance Still Matters, Heisenbugs, Why He Still Writes Code, Singing Pigs, and More, SIGMOD Rec. 34 (2) (2005) 71–79. doi:10.1145/1083784.1083803.
URL http://doi.acm.org/10.1145/1083784.1083803

[230] M. Grottke, A. P. Nikora, K. S. Trivedi, An empirical investigation of fault types in space mission system software, in: 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN), 2010, pp. 447–456. doi:10.1109/DSN.2010.5544284.

[231] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, K. S. Trivedi, Fault triggers in open-source software: An experience report, in: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), 2013, pp. 178–187. doi:10.1109/ISSRE.2013.6698917.

[232] R. Chillarege, Comparing four case studies on Bohr-Mandel characteristics using ODC, in: 2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2013, pp. 285–289. doi:10.1109/ISSREW.2013.6688908.

[233] J. S. Bradbury, K. Jalbert, Defining a Catalog of Programming Anti-Patterns for Concurrent Java, in: Proceedings of the 3rd International Workshop on Software Patterns and Quality, SPAQu'09, 2009, pp. 6–11.

[234] J. Duffy, Solving 11 Likely Problems In Your Multithreaded Code, MSDN Magazine (October 2008).

[235] G. Gopalakrishnan, J. Sawaya, Achieving Formal Parallel Program Debugging by Incentivizing CS/HPC Collaborative Tool Development, in: Proceedings of the 1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models, SCREAM '15, ACM, New York, NY, USA, 2015, pp. 11–18. doi:10.1145/2753524.2753531.
URL http://doi.acm.org/10.1145/2753524.2753531

[236] P. C. Mehlitz, W. Visser, J. Penix, The JPF Runtime Verification System, http://www.doc.gold.ac.uk/%7Emas01sd/classes/jpf_release/doc/JPF.pdf (2005).
URL http://www.doc.gold.ac.uk/%7Emas01sd/classes/jpf_release/doc/JPF.pdf

[237] A. Mahmood, E. J. McCluskey, Concurrent error detection using watchdog processors-a survey, IEEE Transactions on Computers 37 (2) (1988) 160–174. doi:10.1109/12.2145.

[238] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante, Soft-error detection using control flow assertions, in: Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems, 2003, pp. 581–588. doi:10.1109/DFTVS.2003.1250158.

[239] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, A watchdog processor to detect data and control flow errors, in: 9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003., 2003, pp. 144–148. doi:10.1109/OLT.2003.1214381.

[240] A. Li, B. Hong, Software implemented transient fault detection in space computer, Aerospace Science and Technology 11 (2) (2007) 245–252. doi:https://doi.org/10.1016/j.ast.2006.06.006.
URL http://www.sciencedirect.com/science/article/pii/S1270963806000800

[241] C. Flanagan, P. Godefroid, Dynamic Partial-Order Reduction for Model Checking Software, ACM SIGPLAN Notices - Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages 1 (40) (2005) 110–121.

[242] T. Sharma, D. Spinellis, A survey on software smells, Journal of Systems and Software 138 (2018) 158–173. doi:https://doi.org/10.1016/j.jss.2017.12.034.
URL http://www.sciencedirect.com/science/article/pii/S0164121217303114

[243] M. S. Haque, J. Carver, T. Atkison, Causes, Impacts, and Detection Approaches of Code Smell: A Survey, in: Proceedings of the ACMSE 2018 Conference, ACMSE '18, ACM, New York, NY, USA, 2018, pp. 25:1–25:8. doi:10.1145/3190645.3190697.
URL http://doi.acm.org/10.1145/3190645.3190697

[244] S. S. Rathore, S. Kumar, A study on software fault prediction techniques, Artificial Intelligence Review 51 (2) (2019) 255–327. doi:10.1007/s10462-017-9563-5.
URL https://doi.org/10.1007/s10462-017-9563-5

[245] V. V. Kuliamin, A. A. Petukhov, A survey of methods for constructing covering arrays, Programming and Computer Software 37 (3) (2011) 121. doi:10.1134/S0361768811030029.
URL https://doi.org/10.1134/S0361768811030029

[246] E. P. Enoiu, A. Čaušević, T. J. Ostrand, E. J. Weyuker, D. Sundmark, P. Pettersson, Automated test generation using model checking: an industrial evaluation, International Journal on Software Tools for Technology Transfer 18 (3) (2016) 335–353. doi:10.1007/s10009-014-0355-9.
URL https://doi.org/10.1007/s10009-014-0355-9

[247] A. Salahirad, H. Almulla, G. Gay, Choosing the fitness function for the job: Automated generation of test suites that detect real faults, Software Testing, Verification and Reliability 29 (4-5) (2019) e1701, e1701 stvr.1701. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1701, doi:10.1002/stvr.1701.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1701

[248] A. Schwartz, D. Puckett, Y. Meng, G. Gay, Investigating faults missed by test suites achieving high code coverage, Journal of Systems and Software 144 (2018) 106–120. doi:https://doi.org/10.1016/j.jss.2018.06.024.
URL http://www.sciencedirect.com/science/article/pii/S0164121218301201

[249] B. S. Ahmed, Test case minimization approach using fault detection and combinatorial optimization techniques for configuration-aware structural testing, Engineering Science and Technology, an International Journal 19 (2) (2016) 737–753. doi:https://doi.org/10.1016/j.jestch.2015.11.006.
URL http://www.sciencedirect.com/science/article/pii/S2215098615001706

[250] S. M. Ghaffarian, H. R. Shahriari, Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey, ACM Comput. Surv. 50 (4) (2017) 56:1–56:36. doi:10.1145/3092566.
URL http://doi.acm.org/10.1145/3092566

[251] N. Bazan, Static and Dynamic Verification Tools, https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-and-dynamic-verification-tools/, [Online; accessed 24-March-2019] (2017).

[252] M. Felleisen, R. Cartwright, Safety as a metric, in: Proceedings 12th Conference on Software Engineering Education and Training (Cat. No.PR00131), 1999, pp. 129–131. doi:10.1109/CSEE.1999.755192.
URL https://doi.org/10.1109/CSEE.1999.755192

[253] Wikipedia, Satisfiability modulo theories, https://en.wikipedia.org/wiki/Satisfiability_modulo_theories, [Online; accessed 24-March-2019] (2019).
URL https://en.wikipedia.org/wiki/Satisfiability_modulo_theories

[254] N. Machado, B. Lucia, L. Rodrigues, Production-guided Concurrency Debugging, ACM SIGPLAN Notices - PPoPP '16 51 (8) (August 2016).

[255] N. Machado, B. Lucia, L. Rodrigues, Concurrency Debugging with Differential Schedule Projections, ACM SIGPLAN Notices - PLDI 15 50 (6) (2015) 586–595.

[256] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, I. Neamtiu, Finding and Reproducing Heisenbugs in Concurrent Programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 267–280.
URL http://dl.acm.org/citation.cfm?id=1855741.1855760

[257] F. Koca, H. Sözer, R. Abreu, Spectrum-Based Fault Localization for Diagnosing Concurrency Faults, in: H. Yenigün, C. Yilmaz, A. Ulrich (Eds.), Testing Software and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 239–254.

[258] R. Abreu, P. Zoeteweij, A. J. C. v. Gemund, Spectrum-Based Multiple Fault Localization, in: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE 09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 88–99. doi:10.1109/ASE.2009.25.
URL https://doi.org/10.1109/ASE.2009.25

[259] S. Park, S. Lu, Y. Zhou, CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places, ACM SIGPLAN Notices - ASPLOS 2009 44 (3) (2009) 25–36. doi:10.1145/1508284.1508249.
URL http://doi.acm.org/10.1145/1508284.1508249

[260] C. S. Păsăreanu, W. Visser, Symbolic Execution and Model Checking for Testing, in: K. Yorav (Ed.), Hardware and Software: Verification and Testing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 17–18.

[261] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided Abstraction Refinement for Symbolic Model Checking, J. ACM 50 (5) (2003) 752–794. doi:10.1145/876638.876643.
URL http://doi.acm.org/10.1145/876638.876643

[262] C. B. Bergersen, Java Path Finder, https://www.uio.no/studier/emner/matnat/ifi/INF5140/v15/slides/jpf.pdf, jPF (2015).

[263] K. Y. Rozier, Survey: Linear Temporal Logic Symbolic Model Checking, Comput. Sci. Rev. 5 (2) (2011) 163–203. doi:10.1016/j.cosrev.2010.06.002.
URL http://dx.doi.org/10.1016/j.cosrev.2010.06.002

[264] NASA, Java PathFinder, https://github.com/javapathfinder/jpf-core, [Online; accessed 13-July-2019] (2017).
URL https://github.com/javapathfinder/jpf-core

[265] A. Malkis, A. Podelski, A. Rybalchenko, Precise thread-modular verification, SAS'07 Proceedings of the 14th international conference on Static Analysis (2007) 218–232.

[266] NASA, On-the-fly Partial Order Reduction, http://javapathfinder.sourceforge.net/On-the-fly_Partial_Order_Reduction.html, [Online; accessed 24-March-2019] (2009).
URL https://web.archive.org/web/20240418024307/http://javapathfinder.sourceforge.net/On-the-fly_Partial_Order_Reduction.html

[267] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, S. K. Rajamani, Partial-order reduction in symbolic state space exploration, in: O. Grumberg (Ed.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 340–351.

[268] R. Büschkes, M. Borning, D. Kesdogan, Transaction-based anomaly detection, ID'99 Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring 1 (April 1999).

[269] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, B. Saha, Enforcing isolation and ordering in STM, ACM SIGPLAN Notices - Proceedings of the 2007 PLDI conference 42 (6) (2007) 78–88.

[270] M. Mansouri-Samani, P. Mehlitz, C. Pasareanu, J. Penix, G. Brat, L. Markosian, O. O'Malley, T. Pressburger, W. Visser, Program Model Checking: A Practitioner's Guide, https://ti.arc.nasa.gov/m/pub-archive/1439h/1439%20(Mansouri-Samani).pdf, [Online; accessed 11-July-2019] (2012).

[271] NASA, Java PathFinder: A Model Checker for Java Programs, https://ti.arc.nasa.gov/tech/rse/vandv/jpf/, [Online; accessed 11-July-2019] (2012).

[272] H. R. Lewis, C. H. Papadimitriou, Elements of the Theory of Computation, 2nd Edition, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[273] I. U. Plale, Thread Design Patterns, https://www.cs.indiana.edu/classes/b534-plal/ClassNotes/thread-design-patterns4.pdf, [Online; accessed 26-March-2019] (2001).
URL https://www.cs.indiana.edu/classes/b534-plal/ClassNotes/thread-design-patterns4.pdf

[274] A. S. Tanenbaum, M. van Steen, Distributed systems - principles and paradigms, 2nd Edition, Pearson Education, 2007.

[275] P. Parizek, T. Kalibera, Verifying Nested Lock Priority Inheritance in RTEMS with Java Pathfinder, in: International Conference on Formal Engineering Methods, ICFEM 2016, Springer, Cham, 2016, pp. 417—432. doi:https://doi.org/10.1007/978-3-319-47846-3_26.
URL https://link.springer.com/chapter/10.1007/978-3-319-47846-3_26

[276] "djessup" GitHub user, Java Web Server, https://github.com/djessup/java-webserver, [Online; accessed 29-July-2019] (2016).
URL https://github.com/djessup/java-webserver

[277] P. Sotiropoulos, Java Web Server simulation without injected faults, https://github.com/pansot2/java-webserver/tree/simulation, [Online; accessed 29-August-2019] (2019).
URL https://github.com/pansot2/java-webserver/tree/simulation

[278] P. Sotiropoulos, Java Web Server with injected faults, https://github.com/pansot2/java-webserver/tree/jpf-simulation, [Online; accessed 29-August-2019] (2019).
URL https://github.com/pansot2/java-webserver/tree/jpf-simulation

[279] S. Dandamudi, Addressing Modes, in: T. editor (Ed.), Introduction to Assembly Language Programming, Undergraduate Texts in Computer Science, Springer, New York, NY, 1998, Ch. 5, pp. 173–206.

[280] A. Holzer, D. Schwartz-Narbonne, M. Tabaei Befrouei, G. Weissenbacher, T. Wies, Error Invariants for Concurrent Traces, in: J. Fitzgerald, C. Heitmeyer, S. Gnesi, A. Philippou (Eds.), FM 2016: Formal Methods, Springer International Publishing, Cham, 2016, pp. 370–387.

[281] S. K. Pandey, R. B. Mishra, A. K. Tripathi, Machine learning based methods for software fault prediction: A survey, Expert Systems with Applications 172 (2021) 114595. doi:10.1016/j.eswa.2021.114595.
URL http://dx.doi.org/10.1016/j.eswa.2021.114595

[282] N. Wild, H. Lichter, Unit Test Based Component Integration Testing, in: 2023 30th Asia-Pacific Software Engineering Conference (APSEC), 2023, pp. 1–10. doi:10.1109/APSEC60848.2023.00010.

[283] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, E. Wang, Automated Unit Test Improvement using Large Language Models at Meta, in: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Association for Computing Machinery, New York, NY, USA, 2024, p. 185–196. doi:10.1145/3663529.3663839.
URL https://doi.org/10.1145/3663529.3663839

[284] M. Barbareschi, S. Barone, V. Casola, S. Della Torca, D. Lombardi, Automatic Test Generation to Improve Scrum for Safety Agile Methodology, in: Proceedings of the 18th International Conference on Availability, Reliability and Security, ARES '23, Association for Computing Machinery, New York, NY, USA, 2023. doi:10.1145/3600160.3605061.
URL https://doi.org/10.1145/3600160.3605061

[285] S. Lukasczyk, F. Kroiß, G. Fraser, An empirical study of automated unit test generation for Python, Empirical Software Engineering 28 (2) (Jan. 2023). doi:10.1007/s10664-022-10248-w.
URL http://dx.doi.org/10.1007/s10664-022-10248-w

[286] X. Cheng, H. Wang, J. Hua, G. Xu, Y. Sui, DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network, ACM Trans. Softw. Eng. Methodol. 30 (3) (apr 2021). doi:10.1145/3436877.
URL https://doi.org/10.1145/3436877

[287] D. Hin, A. Kan, H. Chen, M. A. Babar, LineVD: statement-level vulnerability detection using graph neural networks, in: Proceedings of the 19th International Conference on Mining Software Repositories, MSR '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 596–607. doi:10.1145/3524842.3527949.
URL https://doi.org/10.1145/3524842.3527949

[288] Q. I. Sarhan, A. Beszedes, A Survey of Challenges in Spectrum-Based Software Fault Localization, IEEE Access 10 (2022) 10618–10639. doi:10.1109/ACCESS.2022.3144079.

[289] W. E. Wong, H. Agrawal, X. Zhang, Slicing-Based Techniques for Software Fault Localization (Apr. 2023). doi:10.1002/9781119880929.ch3.
URL http://dx.doi.org/10.1002/9781119880929.ch3

[290] Y. Li, S. Wang, T. Nguyen, Fault Localization with Code Coverage Representation Learning, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 661–673. doi:10.1109/ICSE43902.2021.00067.

[291] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, L. Zhang, Boosting coverage-based fault localization via graph-based representation learning, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 664–676. doi:10.1145/3468264.3468580.
URL https://doi.org/10.1145/3468264.3468580

[292] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, S. Li, Just-In-Time Defect Identification and Localization: A Two-Phase Framework, IEEE Transactions on Software Engineering 48 (1) (2022) 82–101. doi:10.1109/TSE.2020.2978819.

[293] R. Widyasari, J. W. Ang, T. G. Nguyen, N. Sharma, D. Lo, Demystifying Faulty Code: Step-by-Step Reasoning for Explainable Fault Localization, in: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2024, pp. 568–579. doi:10.1109/SANER60148.2024.00064.

[294] S. Kang, G. An, S. Yoo, A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization, Proc. ACM Softw. Eng. 1 (FSE) (jul 2024). doi:10.1145/3660771.
URL https://doi.org/10.1145/3660771

[295] D. Chkliaev, J. Hooman, P. van der Stok, Serializability Preserving Extensions of Concurrency Control Protocols, in: Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2000, pp. 180–193. doi:10.1007/3-540-46562-6_15.
URL https://doi.org/10.1007%2F3-540-46562-6_15

[296] P. Sotiropoulos, JPF 2nd listener with db storage, [Online; accessed 17-September-2024] (2024).
URL https://doi.org/10.6084/m9.figshare.27043441

[297] P. Sotiropoulos, JPF 2nd listener with variable-field relations, [Online; accessed 17-September-2024] (2024).
URL https://doi.org/10.6084/m9.figshare.27043660

[298] P. Sotiropoulos, Replication package for JPF-based intermittent fault detection method, [Online; accessed 17-September-2024] (2024).
URL https://zenodo.org/TOBECOMPLETED

[299] N. Mohamed, R. F. R. Sulaiman, W. R. W. Endut, The Use of Cyclomatic Complexity Metrics in Programming Performance's Assessment, Procedia - Social and Behavioral Sciences 90 (2013) 497–503. doi:10.1016/j.sbspro.2013.07.119.
URL http://dx.doi.org/10.1016/j.sbspro.2013.07.119

[300] S. Liu, J. Bender, J. Palsberg, Compiling Volatile Correctly in Java, in: Proceedings of the 36th European Conference on Object-Oriented Programming (ECOOP 2022), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ECOOP.2022.6.
URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2022.6

[301] P. Sotiropoulos, JPF experimental listener with db storage, [Online; accessed 17-September-2024] (2024).
URL https://doi.org/10.6084/m9.figshare.27043711

[302] WIPO, WIPO - World Intellectual Property Organization, https://www.wipo.int/portal/en/index.html, [Accessed 06-05-2025] (2023).

[303] T. U. S. Patent, T. Office, 2105-Patent Eligible Subject Matter - Living Subject Matter, https://www.uspto.gov/web/offices/pac/mpep/s2105.html, [Accessed 06-05-2025] (2015).

[304] T. U. S. Patent, T. Office, Patent Subject Matter Eligibility, https://www.uspto.gov/web/offices/pac/mpep/s2106.html, [Accessed 06-05-2025] (2016).

[305] EPC, European Patent Convention, https://www.epo.org/law-practice/legal-texts/epc.html, [Accessed 06-05-2025] (2024).

[306] EPCnat, National law relating to the EPC, https://www.epo.org/law-practice/legal-texts/national-law.html, [Accessed 06-05-2025] (2024).

[307] O. S. Initiative, OSI Approved Licenses, https://opensource.org/licenses, [Accessed 10-05-2025] (2024).

[308] O. S. Initiative, OSI Approved Licenses, https://opensource.org/licenses, [Accessed 10-05-2025] (2024).

[309] L. Insider, Open Source License Definition, https://www.lawinsider.com/dictionary/open-source-license, [Accessed 10-05-2025] (2024).

[310] C. a License, Licenses, https://choosealicense.com/licenses/, [Accessed 10-05-2025] (2025).

[311] C. a License, Non-software licenses, https://choosealicense.com/non-software/, [Accessed 10-05-2025] (2025).

[312] L. Insider, Product License, https://www.lawinsider.com/clause/product-license, [Accessed 10-05-2025] (2024).

[313] C. Alliance, Differences Between Copyright, Trademarks, Patents, and Trade Secrets?, https://copyrightalliance.org/faqs/difference-copyright-patent-trademark/, [Accessed 10-05-2025] (2024).

[314] Himanshu Gupta, Suresh Kumar, Saroj Kumar Roy, Ram S. Gaud, Patent protection strategies, Journal of Pharmacy And Bioallied Sciences 2 (1) (2010) 2–7. doi:10.4103/0975-7406.62694.

[315] Bruno van Pottelsberghe, Didier François, The Cost Factor in Patent Systems, Journal of Industry Competition and Trade 7 (4) (2009) 325–355.

[316] Erwin F. Berrier, Jr, GLOBAL PATENT COSTS MUST BE REDUCED, The Journal of Law and Technology (1996) 473.

[317] Stephen J. Sundfold, The need for pre-patent protection: an analysis of Lear, inc. v. Adkins, in: Sun Diego Law Review, Vol. 7, 1964, pp. 325–331.

[318] FRANÇOIS LÉVÊQUE, YANN MÉNIÈRE, COPYRIGHT VERSUS PATENTS: THE OPEN SOURCE SOFTWARE LEGAL BATTLE, Review of Economic Research on Copyright Issues 4 (1) (2007) 27–46.

[319] Christopher Vendome, Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Daniel German, Denys Poshyvanyk, When and Why Developers Adopt and Change Software Licenses, IEEE International Conference on Software Maintenance and Evolution (ICSME) (September 2015). doi:doi:10.1109/ICSM.2015.7332449.

[320] Miriam Ballhausen, Free and Open Source Software Licenses Explained, COMPUTER PUBLISHED BY THE IEEE COMPUTER SOCIETY (2019) 82–86.

[321] Amandeep Singh, R.K Bansal, Ph.D Neetu Jha, Open Source Software vs Proprietary Software, International Journal of Computer Applications 114 (18) (2015) 82–86.

[322] YI-HSUAN LIN , TUNG-MEIKO , TYNG-RUEY CHUANG AND KWEI-JAY LIN, Open Source Licenses and the Creative Commons Framework: License Selection and Comparison, JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 22 (2006) 1–17.

[323] Thomas A. Alspaugh, Hazeline U. Asuncion, and Walt Scacchi, The Role of Software Licenses in Open Architecture Ecosystems, in: Proceedings of the first International Workshop on Software Ecosystems, Vol. 22, 2009, pp. 1–17.

[324] B. van Pottelsberghe de la Potterie, D. François, The cost factor in patent systems, Journal of Industry, Competition and Trade 9 (4) (2008) 329–355. doi:10.1007/s10842-008-0033-2.
URL http://dx.doi.org/10.1007/s10842-008-0033-2

[325] T. C. for Open Science, Licensing - OSF Support — help.osf.io, https://help.osf.io/article/148-licensing, [Accessed 25-05-2025] (2024).

[326] Wikipedia, Software relicensing — Wikipedia, the free encyclopedia, http://en.wikipedia.org/w/index.php?title=Software%20relicensing&oldid=1288873583, [Online; accessed 25-May-2025] (2025).

[327] N. Mavrogiannopoulos, The perils of LGPLv3 — nikmav.blogspot.com, https://nikmav.blogspot.com/2013/03/the-perils-of-lgplv3.html, [Accessed 25-05-2025] (2013).

[328] MariaDB, Business Source License 1.1 | MariaDB (change license: Version 2 or later of the gnu general public license as published by the free software foundation), https://mariadb.com/bsl11/, [Accessed 25-05-2025] (2016).

[329] S. Xu, Y. Gao, L. Fan, Z. Liu, Y. Liu, H. Ji, Lidetector: License incompatibility detection for open source software, ACM Trans. Softw. Eng. Methodol. 32 (1) (Feb. 2023). doi:10.1145/3518994.
URL https://doi.org/10.1145/3518994