Technische
Universität
Braunschweig

Delta-oriented Test Set Optimization using Genetic Algorithms

Masterthesis

Sabrina Lischke

November 15, 2016

Institute of Software Engineering and Automotive Informatics
at the
Technische Universität Carolo-Wilhelmina in Braunschweig

Remo Lachmann, M.Sc.

Ich versichere, dass ich die beiliegende Masterarbeit ohne Hilfe Dritter und ohne Benutzung anderer, als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher Form noch keiner Prüfungsbehörde vorgelegen.
Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum Unterschrift

Zusammenfassung

Software Produkt Linien (SPL) bieten die Möglichkeit Software an eine große Anzahl von Kunden individuell anzupassen. Die exponentiell ansteigende Anzahl der Produktvarianten mit Gemeinsamkeiten und Unterschieden führen zu neuen Herausforderungen for das Testen von Software. Besonders in sicherheitskritischen Systemen sollte jede Produktvariante gründlich getestet werden. Zusätzlich würden durch die Gemeinsamkeiten der Produktvarianten das Testen jeder einzelnen zu hohen Redundanzen beim Testen führen. Deshalb werden neue Konzepte und Methoden, die die Gemeinsamkeiten ausnutzen, zum effizienten Testen von SPL benötigt. Diese Masterarbeit stellt eine search-based Methode for den Integrationstest von delta-orientierten SPL vor. Die search-based Methode benutzt meta-heuristische Optimierungen, wie z.B. Genetische Algorithmen (GA), um Testabläufe zu automatisieren. In dieser Masterarbeit wird ein GA benutzt um Testfälle für den Integrationstest in Hinblick auf mehrere Ziele zu priorisieren. Probleme aus der realen Welt benötigen oft eine Optimierung in Hinblick auf mehrere oft gegensätzliche Ziele. Ein GA versucht eine Menge von optimalen Lösungen für diese Probleme in einer akzeptablen Zeit zu finden. In dieser Masterarbeit haben wir mathematisch mehrere Ziele definiert um das Priorisierungsproblem zu lösen. Dabei soll die Variabilität zwischen den einzelnen Produktvarianten, z.B. Änderungen in der Architektur, fokussiert werden und Testfälle so geordnet werden, dass Variabilität zwischen Produktvarianten früh getestet wird. Wir stellen ein allgemeines Konzept für die Ziele vor und benutzen einen Prototyp des Konzepts um die Ergebnisse zu evaluieren.

Für die Evaluation in dieser Masterarbeit haben wir die *Average Percentage of Faults Detected* (APFD) Metrik vorgestellt von Rothermel et al. [18] genutzt um die Qualität der priorisierten Testsets zu messen. Wir vergleichen zwei verschiedene GA mit einem zufälligen Ansatz und zwei anderen Priorisierungsansätzen.

Schlüsselwörter: Delta-orientiert, Search-based, Software Produkt Linien, Integrationstest, Model-basiert

Abstract

Software product lines (SPL) are used to provide mass customization of software. The exponentially increasing number of possible product variants with commonalities and variabilities lead to new challenges in software testing. Especially in safety-critical systems every product variant should be thoroughly tested. Additional, testing every product variant individually will lead to high redundancy in the testing process due to the commonalities between product variants. Therefore, new testing concepts and techniques are required to provide efficient SPL testing exploiting the commonalities between product variants. This thesis presents a search-based testing approach to prioritize test sets for integration testing in delta-oriented SPLs. The search-based testing approach uses meta-heuristic optimization search techniques, such as a genetic algorithm (GA), to automate testing tasks. In this thesis, a GA is used to prioritize test cases for integration testing with respect to multiple objectives. Real world problem often needs an optimization of a problem with regards to multiple often competing objectives. However, for these problems not one, single solutions exists, but a set of possible optimal solutions. A GA aims to find the set of possible optimal solutions in an acceptable time. In this thesis we mathematically specify objectives to solve the prioritization problem of integration test set with focus on the variability between product variants, e.g. changes in the architecture, and aim to order test cases, such that variability between product variants is tested earlier. We introduce the general concept of the objectives and use a prototype of this concept for the evaluation.

For the evaluation of this thesis we used the *Average Percentage of Faults Detected* (APFD) metric introduced by Rothermel et al. [18] to measure the quality of the resulting test sets. We compare two different GAs with a random approach and two other prioritization approaches to evaluate how effective the concept in this thesis is. The results of the evaluation shows that a prioritization of test sets with the introduced concept increases the fault detection rate compared to other prioritization approaches.

Keywords: Delta-oriented, Search-based, Software Product Lines, Integration Testing, Model-based

Contents

List of Figures

List of Tables

Listings

1 Introduction

The increasing use of software systems in more and more domains makes it essential to ensure that these systems meet their requirements. Therefore, the validation of the correct and reliable behavior of a software system is an important task [40]. To ensure the reliability, the software has to be tested thoroughly. Studies have shown that at least 50% of the overall resources for a software project are spent on testing a software system [46].

Due to the need of customized software and personalized adaption of features, the development of single software systems was replaced with the concept of *software product lines* (SPLs) [34] in many domains, e.g., the automotive industry [67]. A SPL is a set of software systems which share commonalities and variabilities [51]. To handle the variability in the development process *software product line engineering* (SPLE) was introduced [34]. SPLE aims to reduce time to market and to increase product quality by reusing software artifacts [6]. Variability in SPLs is represented by variability models, e.g., *feature models* [36]. Every *feature* describes increments of product functionality which are visible to the customer. A product variant is identified by a valid *feature combination*, i.e., a valid combination of features from the feature model. Since the number of products increases exponentially based on the number of optional features, testing SPLs leads to new challenges [34]. Testing all product variants of a SPL thoroughly is infeasible [57]. Moreover, testing each product variant in isolation will lead to a vast amount of redundant test cases due to the commonalities between product variants. However, safety-critical software, e.g, in the automotive domain, requires comprehensive testing of all possible product variants to fulfill certain standards, e.g., the ISO26262 standard [30] for the automotive domain. To cope with these challenges new techniques have to be developed. The commonalities between product variants have to be used to reduce the testing effort and increase the test efficiency. Additionally, test artifacts, e.g., test cases, have to be reused [57].

The reuse of test artifacts can be achieved by using model-based testing approaches [40, 68]. For model-based testing an abstract model of the system under test is designed which is used to generate test data, e.g, test cases. Due to the commonalities of product variants the test models for SPL testing can be partially reused [44]. One approach to cope with the variability of SPLs is the delta-oriented approach [6, 62]. In delta-oriented modeling (DOM), a core model is defined which represents a valid product variant for the SPL. Additionally, deltas are defined which specify modifications to the core model to generate new product variants.

Additionally, regression testing techniques can be used to reduce the testing effort and reuse test artifacts. *Regression testing* techniques focus on the validation of modified software and ensures that no new failures are introduced into previously tested code [46]. These regression testing techniques, such as test case selection and test case prioritization [71], show promising results to reduce the testing effort in single software systems [57]. For SPL testing, regression testing techniques were adapted to exploit the commonalities between product variants to reduce redundancy in testing [57, 59, 69].

Currently, regression testing techniques for SPLs focus on one objective to select test cases or prioritize them. However, for many selection or prioritization problems it is necessary to focus on multiple objectives, e.g., cost and profit, at the same time. However, this problems can have a huge search space and no algorithm with low polynomial time is available. To cope with this challenge *search-based software testing* which uses meta heuristic search techniques for the automatic generation of test data has gained more interest [48]. In general, test data selection is a manual process which is costly and difficult due to the fact that software systems are complex. Therefore, the application of meta heuristic search techniques, e.g., Genetic Algorithms (GAs) [47], for test data generation is a promising solution to this problem. The idea of GAs is based on the genetic process of biological organisms. They mimic the process that over many generations natural populations evolve according to the principles of natural selection and *survival of the fittest*.

1.1. Contribution of this Thesis

The purpose of this work is to design multiple objectives for GAs for test case prioritization in SPL testing. The complexity of the testing task shall be reduced. This means important test cases are at the beginning of the test set and if the test execution stops at any given point, e.g., due to resource constraints, the most important test cases have already been executed. Therefore, objectives have to be defined to support the test case prioritization. The contribution of this thesis are three novel objectives for SPL testing.

One objective is the maximization of the *Average Percentage of Changes Covered* (APCC) metric introduced by Lachmann et al. [57]. This metric measures at which position a change in the architecture is covered by test cases. Another objective is the maximization of the dissimilarity between test cases. Two objectives introduced in this thesis focus feature combinations covered by test cases. To prioritize test sets with these four objectives, it is necessary to develop novel fitness functions to measure the quality of a test set with regards to the objectives.

Furthermore, the different operators of the GA shall be properly defined, so that these operators support the test case prioritization of the test set with respect to the aforementioned goals. Afterwards, the concept is prototyped to select different GAs and executes them. For the implementation of GAs the framework jMetal [13] is used. The prototype tool should be integrated into existing landscape, e.g., a test case description [41], an architecture description [37] and a delta-oriented prioritization [57].

To complete this thesis a proper evaluation shall summarize and validate the results of this work. The *Body Comfort System* study [60] serves as the foundation for the evaluation providing the architecture models, state chart diagrams and the test cases for this work. To validate the results the *Average Percentage of Faults Detected* (APFD) metric introduced by Rothermel et al. [18] of the prioritized test sets should be measured. With this metric it is possible to compare the different objectives and the objective combinations.

1.2. Structure of the Thesis

The remainder of this thesis of structured as follows:

The second chapter will introduce necessary background to understand the topics of this thesis. At first, SPLs, delta-oriented SPLs and delta-oriented modeling is explained in more detail. Afterwards, software testing and especially SPL testing are described. Finally, search-based software testing and GAs are explained. The third chapter discusses related work. Afterwards, the fourth chapter presents the concept of the developed objectives. Each objective is explained and a metric for each objective is defined. In chapter five the implementation of the prototype tool is elucidated. At first, the requirements of the tool are discussed and afterwards the architecture of the implementation is shown and described. The usage of the prototype is explained. Chapter six describes the evaluation of the concept. At first, two research questions are defined which are answered in the evaluation using the results of the implementation. The result is presented and discussed and the threats to the validity of the evaluation are explained. The seventh chapter concludes the thesis and presents future work of this thesis.

2 Background

This chapter introduce necessary background to understand the topics of this thesis. At first, SPL and delta-oriented SPLs and delta-oriented modeling is explained. Afterwards, software testing and SPL testing with all challenges is described. Finally, search-based software testing and GAs are explained in more detail.

2.1. Software Product Lines

The need of customized software and personalized adaption of functionality has shown that single software system can not handle the challenges for this need. Developing customized software with single software development approaches is costly and time intense. To cope with these challenges the concept of SPLs was introduced [51].

A software product line (SPL) is a set of software systems, with commonalities and variabilities, taking into account the customers' requirements [34]. SPLs are described by product characteristics or product functionalities encapsulated by *Features*. Kang et al. [36] introduced *Feature Models (FM)* to represent features and their relationship in a graphical way. They are a hierarchical tree structure, which illustrate the commonalities and variabilities of the SPL. Constraints in the feature model describe if some features can not be selected at the same time or must be selected together. Therefore, the relations between features and product line are easy to comprehend. Figure 2.1 shows an example of a feature model for a home automation system.

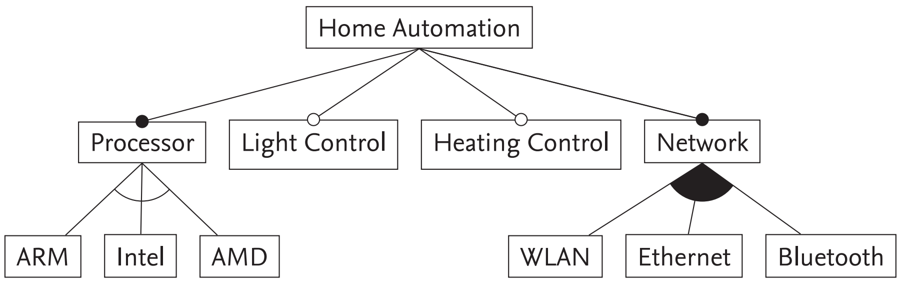


Figure 2.1.: Example of a feature model

A filled circle on a feature implies that a feature is mandatory and is present in every product variant. For example, the feature *Processor* in Figure 2.1 got a filled circle, and thus, every product variant of this home automation system requires a processor. The feature *Network* is present in every product variant, as well. In addition, a FM describes a parent-child relationship between features. This means a drawn line between a child feature and a parent feature indicates that a child feature requires its parent feature to be present [36]. The feature *Processor* is a parent feature and has three child features, *ARM, Intel* and *AMD.* The empty arc on the edges from the parent feature to the child features indicates an *Alternative* between the child features. Only one of the child features can be selected in this case. For example, the features *ARM, Intel* and *AMD* can only be selected if the feature *Processor* is present and only one of them can be present in the product variant. Likewise, the feature *Network* is a parent feature. The filled arc between *Network* and it's child features *WLAN, Ethernet* and *Bluetooth* indicates an *or-relationship*, where at least one of the children has to be selected. Not every feature is mandatory in a SPL. These features are *optional*, because they are not needed in an executable product variant, but the customer can chose to select them. These features are illustrated with an empty circle in the FM. In Figure 2.1 the features *Light Control* and *Heating Control* are optional and not required for an executable product variant.

Several challenges arise for SPL development. First, the number of possible products increase exponentially based on the number of features [34]. Second, the *Optional Feature Problem* occurs when optional features are independent in the feature model, but are not independent in their implementation [33]. Therefore, a development process is needed, to avoid these problems. The next subsection introduce *Software Product Line Engineering*, which describes an efficient process to develop SPL.

2.1.1. Software Product Line Engineering

It is not feasible to implement and test each variant of a SPL. This would be expensive and the time to market for each product variant would be long. Therefore, software product line engineering (SPLE) was introduced. This paradigm reduces the development costs by reusing artifacts in several different kinds of product variants [34]. In addition, each artifact will be reviewed and tested in many product variants and this implies a significantly higher chance of detecting faults [34]. Therefore, the quality of product variants in a SPL increase.

Pohl et al. [34] defined the term SPLE as follows:

Definition 2.1.1. *„Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization.“*

An overview over the framework for SPLE from Pohl et al. [34] is given in Figure 2.2. The development process is separated into the two processes Domain Engineering and Application Engineering. They will be explained in the following. The domain engineering is defined as follows:

Definition 2.1.2. *„Domain engineering is the process of software product line engineering in which the commonality and the variability of the product line are defined and realized.“ [34]*

In this process the domain artifacts will be created, which are reusable development artifacts. They can be requirements, architectures, components and tests. The domain engineering has five sub-processes which will be described now.

- **Product Management:** In this sub-process the economic aspects and in particular the market strategy of the SPL are handled. The input consists of the company goals and the output is a product roadmap that determines the major common and variable features. Also a schedule and release dates for the future product variants are defined.

- **Domain Requirements Engineering:** The second sub-process summarizes all activities for eliciting and documenting the common and variable requirements of the SPL. The roadmap of the product management is the input for this process. The outputs are reusable, textual and model-based requirements and, in particular, the variability model of the SPL.

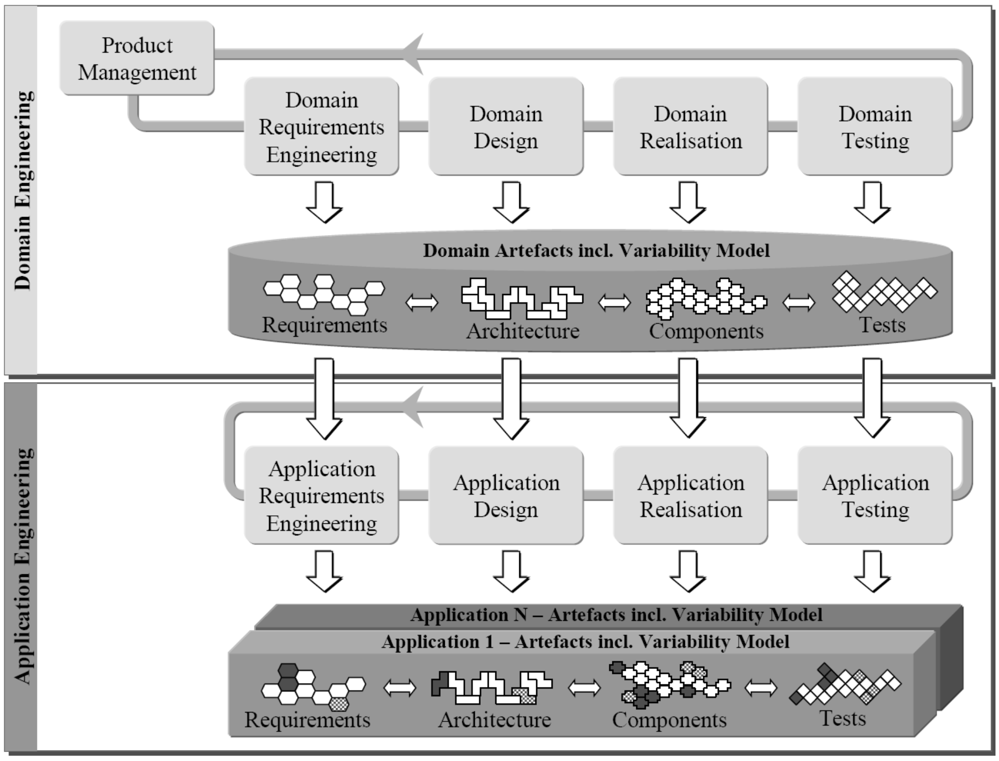


Figure 2.2.: Software Product Line Engineering [34]

- **Domain Design:** This sub-process defines the reference architecture of the SPL. The inputs are the domain requirements and the variability model. The reference architecture and a refined variability model are the output of this sub-process.
- **Domain Realization:** With aid of the reference architecture from the previous sub-process, the domain realization deals with the detailed design and implementation of reusable software components, which are the outputs of this sub-process.
- **Domain Testing:** The last sub-process of the domain engineering is responsible for the validation and verification of reusable components from the domain realization. The components are tested against their specification, i.e. requirements, architecture and design artifacts. The test results and reusable test artifacts are the output of this sub-process.

After domain engineering the process of application engineering will performed. It needs the domain artifacts as input and is defined as follows:

Definition 2.1.3. *„Application engineering is the process of software product line engineering in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability.“ [34]*

This process derives product line applications from the domain artifacts. Application engineering has the following four sub-processes.

- **Application Requirements Engineering:** This sub-process is responsible for developing the application requirements specification. The input are the domain requirements and the product roadmap and the output is the requirements specification for a particular product.

- **Application Design:** This sub-process produces the application architecture. The reference architecture is modified and incorporated with application-specific adaptations. The result is the application architecture.
- **Application Realization:** The third sub-process creates the considered application. It selects and configures the reusable software components and realises application-specific assets. Input are the application architecture and the reusable realisation artifact from the domain engineering. Output is a running application and also detailed design artifacts.
- **Application Testing:** This sub-process validate and verify the application against its specification. It needs the application requirements specification and the application architecture to be used as a test reference. Also it needs the reusable test artifacts from the domain testing. A test report with the results of the tests is the output of this sub-process.

This framework for SPLE is a general description of all activities in the engineering process of a SPL. The next subsection will describe a concrete approach for modeling and programming system variability.

2.1.2. Delta-Oriented Product Lines

It is not feasible to develop each product of a SPL with single software development methods. Therefore, new approaches which use reusable software artifacts are needed. Schaefer et al. [64, 61] introduced the delta-oriented approach for model-based SPL development to realize the variability. Delta-oriented approaches represent the product line with a core and a set of deltas $\Delta = \{\delta_{p1}, \ldots, \delta_{pn}\}$[62]. The core is assumed as a valid feature configuration. The deltas specify modifications to the core, i.e. *add*, *remove* or *modify* objects, to generate all possible product variants of the product line. Each delta contains a boolean expression, the *application condition*. It determines for which feature configuration the specified modifications of the delta are to be carried out. An application condition can be a complex constraint over the product features. Likewise, it is possible to describe the execution order of deltas with an *ordering function*.

Delta-oriented Programming [62] is a programming language approach designed for implementing SPLs. An extension of this approach is *Pure Delta-oriented Programming* which dropped the requirement to chose one product as core product [63], i.e. only deltas are used for product generation. To facilitate automated product derivation for SPL *Delta-oriented Modeling* [6, 64, 61] was introduced. Therefore, no manual intervention during application engineering is needed. This thesis focuses on the delta modeling approach for architecture models [60]. The following Figure 2.4 shows an example of the delta modeling process.

While deltas specify modification from the core to different product variants *Regression Deltas* contain modifications from one product variant to another product variant [45].

Since this thesis focuses on integration testing we use the delta-oriented architecture modeling approach to specify the architecture of a product variant. The next subsection will introduce software architectures and the delta-oriented architecture modeling approach.

2.1.3. Delta-Oriented Architecture Modeling

Software systems get bigger and bigger because of the high number of functionalities. Therefore, it is important to divide the software system into small manageable parts which provide an overview of the system structure, subcomponent dependencies and communication paths [23]. The software architecture of a software system describes the structure of the system as a model. The main elements of an architecture are *components*. These are elements which provide a specific functionality. Two components communicate using a *connector* which exchange *signals*. The most common way to describe an architecture is an UML 2 [52] component diagram. Figure 2.3 shows an example of a component diagram.

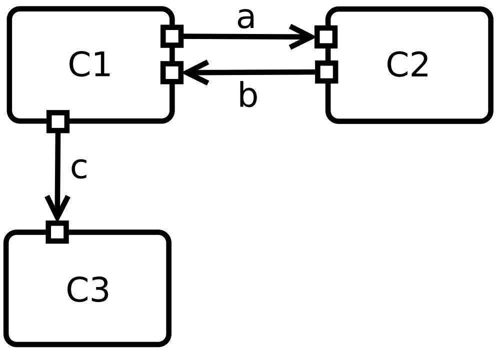


Figure 2.3.: Sample Component Diagram

Components in the diagram are represented by rectangles with round corners. For example in Figure 2.3 there are three components C1, C2 and C3. Component C1 communicates with component C2 over the connector with signal *a*. Component C2 can communicate with C1 over the connector with signal *b*. Connector *c* provides the communication from component C1 to component C3.

Another way to describe an architecture are textual *architecture description languages* (ADL). In this thesis we use the delta-oriented ADL Deltarx which was introduced by Lochau et al. [43]. A delta-oriented architecture test model *arc* consists of a finite set of components $\mathcal{C} = \{c_1, \dots, c_n\}$ representing computational units within the system. A finite set of connectors $\mathcal{CON} = \{con_1, \dots, con_m\}$ transmit a finite set of signals $\Pi = \{\pi_1, \dots, \pi_l\}$ between two components. Each product variant $p_i \in \mathcal{P}$ of the set of all product variants has an own architecture model arc_{p_i}. The core model $arc_{p_{core}}$ represents the architecture model for a designated product variant $p_{core} \in \mathcal{P}$, e.g. the smallest product variant. A set of *deltas* $\Delta = \{\delta_{p_1}, ..., \delta_{p_i}\}$ describes the differences between the core product variant p_{core} and the other product variants $p_i \in \mathcal{P}$ by means of *delta operations* $\delta_{p_i} = \{\delta_i, ..., \delta_m\}$. As described earlier these operations are add, remove and modify components, connectors and signals in order to transform the core model into a variant-specific model. The process of delta-oriented architecture modeling is shown in Figure 2.4.

At first the core is defined as a valid feature configuration from the feature model. In the example, the core consists of three components C1, C2 and C3. Two deltas are applied on the core. The first delta adds a new component C4 and a new connector to the core. The two new objects are marked with a „+“. After adding two new objects the second delta removes the component C3 and their connectors, which is marked with a „−“.

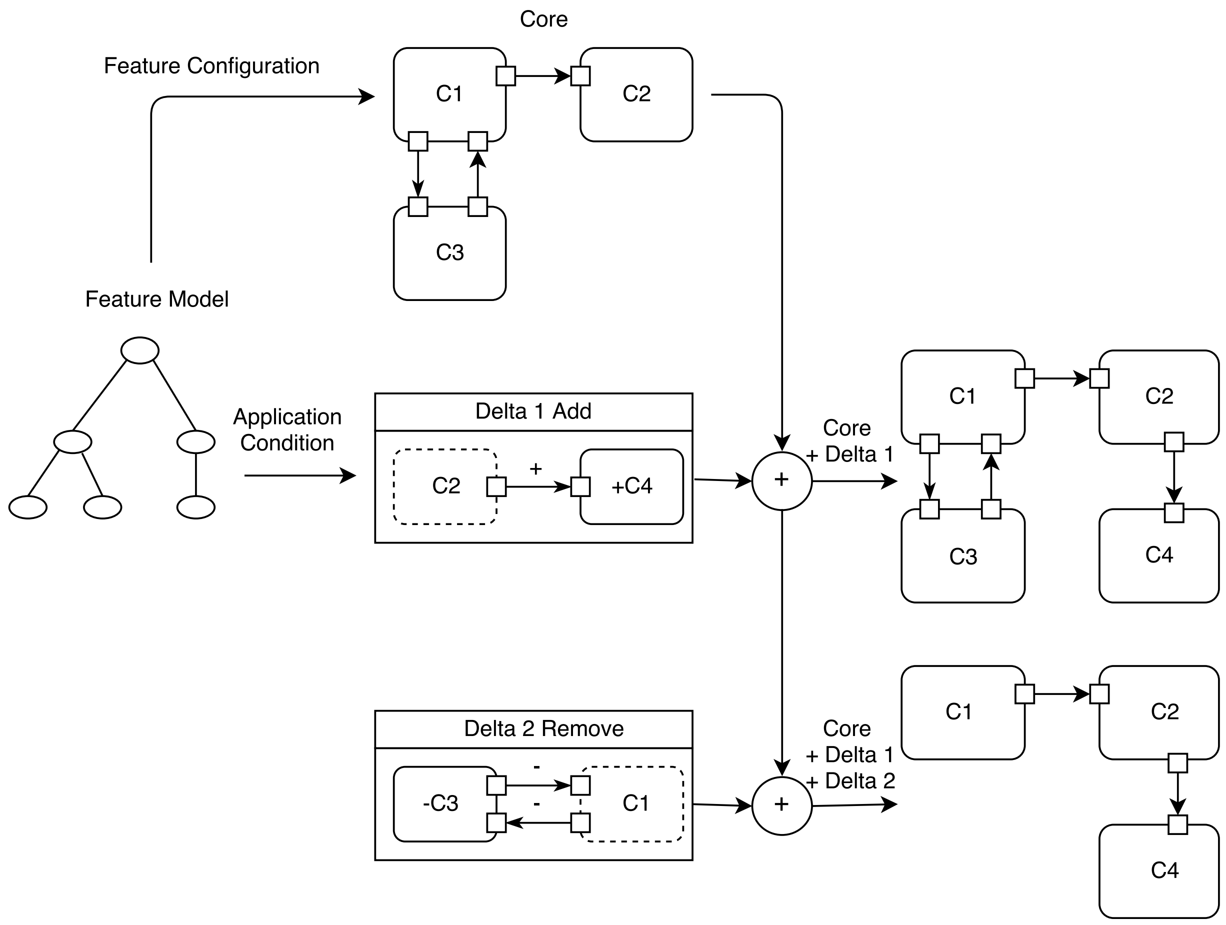


Figure 2.4.: Delta-oriented Modeling for architectures (cf. [37])

After specifying and modeling a SPL the quality of each product variant has to be tested to ensure the quality. The next section will introduce software testing in general and the challenges for SPL testing. Likewise, the next section will introduce approaches which can be used for SPL testing to reduce the testing effort.

2.2. Software Testing

Today software runs on more and more devices. For example, televisions have applications and the coffee machine is controllable with smartphones. Many processes in the industry are controlled by software, as well. The production process is monitored by microprocessors and many corporations work with software tools to manage their staff or orders. Therefore, quality and reliability is very important in software systems. One way to achieve a better quality is to systematically verify and test software [34].

Two terms must be distinguished in software testing. The *failure* is the observable behavior of the system in case of an error, e.g. a value is incorrect or the system crashes [2]. The source of this failure is the *fault*, e.g. a false programmed or missing instruction. Furthermore, the two terms *Validation* and *Verification* have to be introduced [34]. Validation means the tester checks if the test object accomplish his specification. Whereas the verification proves that a particular process is correct and complete.

Software testing aims to detect failures in the system. The developer must then locate and correct the fault in the program code. Figure 2.5 shows the development method *V-Model* for software, which treat the development process and the testing process as equal and corresponding processes.

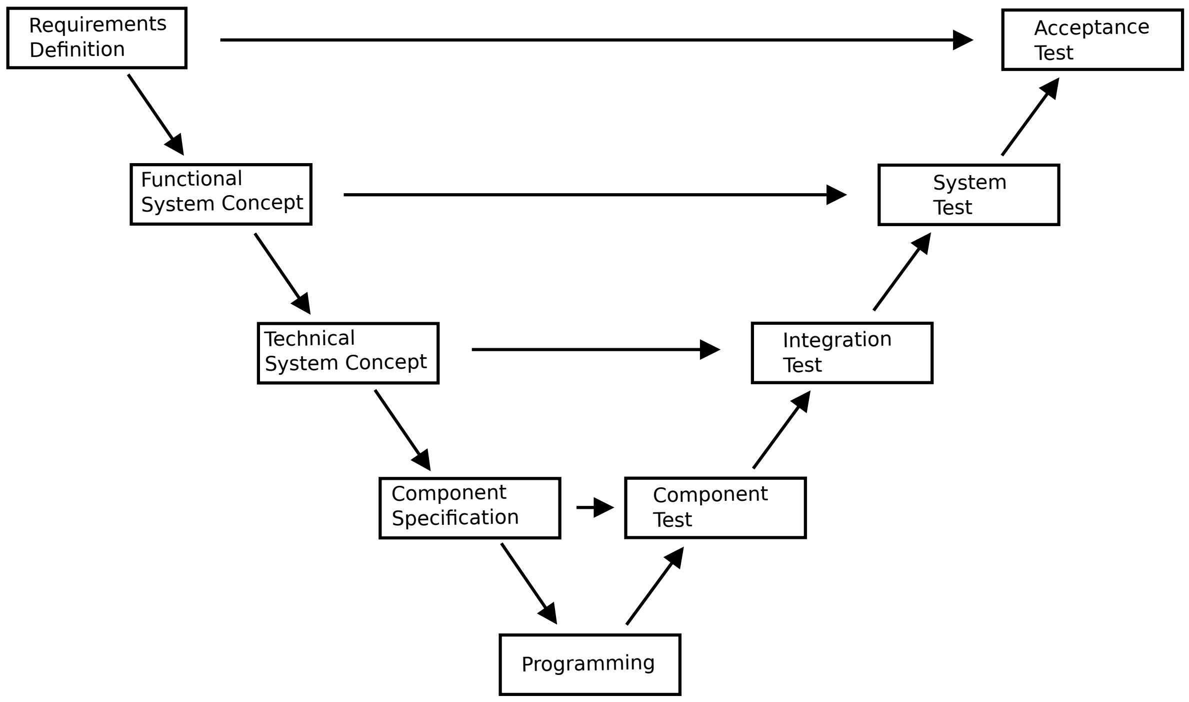


Figure 2.5.: Development Method V-Model (cf. [2])

All development processes are on the left branch. This development processes design and at least program the software system. They are described as:

- **Requirements Definition:** Requirements and demands from the customer are collected and specified. Likewise, the specification is created.
- **Functional System Concept:** The requirements from the previous process are mapped to functions and sequences in the new software systems.
- **Technical System Concept:** In this process the technical realization of the system is created. These include, e.g. definition of the interfaces to the environment and the decomposition of the software system in small components. These components should be independent from each other so that they can be developed independently.
- **Component Specification:** Purpose, behavior, internal structure and interfaces for every component is defined.
- **Programming:** Every specified component is implemented in a particular programming language.

Since it is easier to detect failures on the same abstraction level on which they where developed the right branch assign a testing process to every specification and construction process of the left branch. Therefore, the right branch of the V-Model contains all testing activities. During the testing process the particular components of the system will be integrated into the whole system.

The particular testing processes are:

- **Component Test:** All components from the previous programming process are now tested. It validates if every requirement of the component specification is complied.
- **Integration Test:** Now, all particular components are integrated into subsystems. Precondition of this test is that all components are tested and faults are fixed. Integration testing aims to find faults in the interfaces of components and the interaction of components. The software architecture of the system is important for this process. It describes which components are in the system and dependencies between components. This thesis focus the process of integration testing.
- **System Test:** All components are integrated to the entire software system. While the previous process tested if the integrated components accomplish the technical specification this process aims to validate if the requirements of the customer are accomplished.
- **Acceptance Test:** All test process before where executed by the developer of the software system. The final test process is executed by the customer. It aims to test if the system accomplish the requirements of the customer.

Each test process has *test cases* which describe how the tester should test the software. They describe instructions and expected results. A *test set* is defined as a sequence of test cases $TC = \{tc_1, \ldots, tc_n\} \subseteq \mathcal{TC}$, where $\mathcal{TC}$ refers to the set of all test cases for the system. The test set specifies the order in which the test cases are executed. Generally the test set is created by an experienced tester, but this could lead to a not optimal test set. Therefore, an optimization of the testing order for certain problems, e.g., early fault detection, is necessary. GAs can be used to optimize the test set.

The V-Model is a development method, which aims to improve the quality of software. But it does not describe how the particular testing processes especially for SPL or modified software has to be performed. The next subsection describe regression testing to validate modified software.

2.2.1. Regression Testing

Software evolves over time, due to market or technical changes. Updates are necessary to fix faults or to introduce new features. Likewise, every time a new product variant of a SPL is required the software of the SPL evolves. Therefore, the software must be validated again.

Regression testing aims to validate modified software and ensure that no new faults are introduced into previously tested code [46]. The *retest-all* strategy reruns all applicable test cases for the software. However, as software evolves the test set tends to grow, which means that it can be expensive to execute the entire test set [71]. Many approaches aim to reduce the effort in regression testing [57, 18, 22]. The major topics in regression testing are *test set minimization, test case selection* and *test case prioritization* [71].

Test set minimization identifies and then eliminates test cases from the test set, which are redundant or dispensable to reduce testing overhead.

Test case selection reduces the quantity of the test cases as well. However, it focuses on the identification of modified parts in the software and select only relevant test cases to this changes.

Finally, test case prioritization orders the test cases and seeks to find the optimal permutation of test cases for desirable properties, e.g., the rate of fault detection. This approach does not include test case selection, but testing can be terminated if some condition is fulfilled. For example, all test cases are executed with a prioritization over a certain threshold.

Leung and White [39] introduce three categories for test cases in regression testing. These categories can be used to construct a test set based on the categories of the test cases. For new versions of software the test cases have to be categorized anew.

- **Reusable:** Test cases in this category can be retested again. Their inputs and outputs exists in the modified software. This test cases are defined in TC_{Reuse} But some of this test cases are not necessary to retest, because the parts they will test are not modified.
- **Retestable:** This is a subset of the reusable test cases $TC_{Retest} \subseteq TC_{Reuse} \subseteq TC$. It includes all test cases that retest the modified parts of the software.
- **Invalid:** Due to the modifications in the software some test cases can not executed again. For example their inputs or outputs are no longer in the modified software.

Furthermore, it is possible to design *new* test cases for the modified software TC_{New}. They test all parts that are new in the software and are not tested by old test cases.

The next subsection introduce the principles of SPL testing.

2.2.2. Software Product Line Testing

SPL testing is a challenge due to the combinatorial explosion of the number of products. To test the entire SPL every product variant need to be tested. As mentioned earlier the number of possible product variants grows exponentially with the number of optional features in the SPL. Therefore, approaches were introduced to reduce redundant testing and to minimize testing effort by reusing testing artifacts, e.g. test cases. Similarly the variability in a SPL is a challenge to the testing process [15]. It has to be tested if only the correct features are available in a product variant. Features not supposed to be in the product variant should not included. However, many strategies testing a SPL were developed. Following testing strategies are described by Tevanlinna et al. [66] and Oster et al. [53]:

Product By Product is a traditional strategy, where every product is tested separately without reusing the test assets [31]. Every product is generated and tested individually, each using a standard testing technique.

Incremental Model-Based Testing exploits the commonalities of products [57]. Only the first product is tested individually, while the following products are partially retested with regression techniques, using previously executed test cases. The design of test cases can be really time-consuming because of the high number of possible test cases. Model-based testing is a technique that automates the design of black-box test cases [68]. It uses a test model which describes the specification and the expected behavior of the system under test (SUT) to automatically generate test cases. Therefore, it is easy to derive new test cases if the SUT is modified. Instead of manually writing the test cases the test designer models the specification of the SUT as test model. Afterwards, a model-based testing tool generates a set of test cases from that test model. Test models can be structure diagrams, e.g., component diagram or class diagram, which show the structure of the system and the interaction of components. Likewise, behavior diagrams, e.g., state machine diagrams or sequence diagrams, can be used as test models to describe the behavior of the system. There are usually an infinite number of possible test cases for the test model. Therefore, the test designer has to choose some *test coverage criteria* that defines which test cases the tool should generate. The test case generation tool can focus on a particular part of the test model or generate test cases for a particular model coverage criterion. Coverage criteria can be *all-states*, where each state is visited at least once or *all-transitions*, where every transition must be traversed at least once [68]. The automated test case generation leads to a reduction of test design time and it is possible to generate a variety of test sets from the same model by using different test selection criteria [68].

Model-based testing is also used in SPL testing, where each product variant is a separate SUT with commonalities and variabilities to the other product variants. Therefore, it is better to automatically design a new test model for each product variant under test and generate test cases instead of writing test cases manually. This thesis is based on a model-based testing approach for integration testing of SPLs from Lachmann [57]. The test models are component diagrams which describe the architecture of the SUT. These test models are generated automatically with the delta-oriented modeling approach. Based on the changes of the architecture the test cases are selected and prioritized.

Sampling was introduced by Oster et al. [53]. Instead of testing every product variant combinatorial testing is used to identify a minimal set of product variants. Therefore, the dependencies between features or feature groups must be identified. Afterwards, a minimal set of product variants which test every possible combination of features has to be determined. *Product Prioritization* is proposed in many approaches [24, 3, 27] and uses sampling algorithms to minimize the set of product variants. Product variants in the minimized set are prioritized based on different criteria. For example a similarity-based approach [24] prioritize the product variants based on the dissimilarity. This should lead to an earlier interaction coverage of the SPL.

The next subsection introduce search-based software testing, which automates testing tasks using optimization techniques.

2.3. Genetic Algorithm and Search-Based Software Testing

Some testing tasks, such as test data selection, are difficult and costly. Therefore, the use of techniques which automate or partially automate some of these testing tasks are important [48]. Search-based software testing is one technique to automate testing tasks. It uses meta-heuristic optimization search techniques, such as a GA, Hill Climbing or Simulated Annealing. The problem-specific fitness function is the key for the optimization process [47]. It guides the search to good solutions from a potentially infinite search space within a practical time limit.

This thesis uses GAs to optimize test sets for SPL testing. Therefore, the optimization process of GAs will be described in more detail.

Genetic Algorithms

Many computational problems require searching through a large solution space [49]. For example searching among the huge number of possible amino acid sequences for a protein with specified properties. Those search problems can benefit from an effective use of parallelism in which many different possibilities are explored simultaneously. Rather than evaluate one amino acid sequence at a time it would be much faster to evaluate many simultaneously. However, this requires computational parallelism and an intelligent strategy for choosing the next set of sequences to evaluate.

Hill Climbing is a local search algorithm [47], which can be applied to these type of problems. It chooses an initial solution randomly from the search space as starting point. Then, the neighborhood of this solution is investigated and if a better solution is found the current solution is replaced. Now, the neighborhood of the new solution is investigated, and so on, until no improved neighbors can be found for the current solution. While this is simple and gives fast results this can lead to a suboptimal solution which is locally optimal but not globally [47].

In the 1970s, Holland [29] developed the concept of genetic algorithms (GA). They are methods which are used to solve search and optimization problems [9] inspired by the genetic process of biological organisms. Natural populations evolve according to the principles of natural selection and *survival of the fittest.* These algorithms use the concept of mutation which alter the solutions to prevent it to convergence to a suboptimal solution [28]. Figure 2.6 shows the general process of GAs.

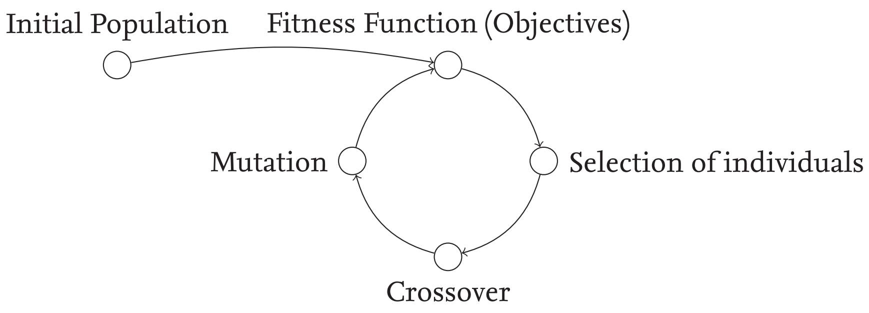


Figure 2.6.: Process of genetic algorithms

At first, the algorithm generate randomly the *initial population*, which is a set of *n individuals*. Each individual represents a possible solution for the problem to be solved and are a set of *genes*, which describes one or more features of the corresponding individual [35].

1. **Fitness Evaluation**

   In the first step, each individual is assigned a *fitness score* according to how good a solution to the problem it is [9]. The evaluation of the individuals is an important task in GAs because only the fittest individuals should survive. The fitness evaluation uses one or more optimization objectives to evaluate the quality of individuals. If the GA uses more than one objective to optimize the problem it solves a multi-objective optimization problem. A sufficient result to a multi-objective function is a set of solutions which satisfy the given objectives at an acceptable level without being dominated by any other solution [35]. These solutions are said to be *pareto optimal*, i.e., they can not be improved with respect to any objective without worsening at least one other objective [35]. The *pareto optimal set* is the set of all non-dominated solutions and is called *pareto front.* The objectives for multi-objective GAs are often competing and the result is a trade-off between the objectives. The fitness evaluation is a difficult task because every problem and domain needs new objectives. However, only good objectives will find good results for the problem. Poor objectives will find a result but it is not optimal for the problem.

2. **Selection**

   After rating the individuals only a few are selected to produce the next generation. The assumption of this operator is that better individuals are more likely to produce better offspring [4]. The selection operator intends to give individuals with higher quality a higher probability to be copied into the next generation to improve the average quality of the population. Blickle and Thiele [4] describe several selection operators:

   - **Tournament Selection:** Randomly t individuals are chosen from the population and the best individual from this group is copied into the intermediate population. This is repeated n times.
   - **Truncation Selection:** With a threshold T only the T best individuals of the population are selected.
   - **Linear Ranking Selection:** The individuals are sorted according to their fitness values and the rank n is assigned to the best individual and the rank 1 to the worst individual. All individuals get a different rank, i.e. a different selection probability, even if the fitness value is the same. Selection probability is linearly assigned to the individuals according to their ranks.
   - **Exponential Ranking Selection:** Similarly to Linear Ranking Selection the individuals get a rank. However, the probabilities of the ranked individuals are exponentially weighted.

3. **Crossover**

   The third step generates new individuals applying crossover techniques. This is an important operator in genetic algorithms [35]. In the previous step the population was reduced by a selection operator. In crossover individuals of the reduced population get the opportunity to reproduce and generate new individuals, called *offspring*, to fill up the population to its previous size. Generally two individuals, the *parents*, are combined together to form the offspring. Larrañaga et al. [38] describe several operators as crossover for the *Traveling Salesman Problem (TSP)*. A set of cities should be visited with the shortest route and every city precisely once. At the end, the route should lead to the starting point. These operators for TSP are interesting for the problem of test set optimization, because every gene exists only once in the individual. In a test set every test case should only be once, because a repetition will not lead to another result.

   - **Partially-Mapped Crossover (PMX Crossover):** This operator was suggested by Goldberg and Lingle [19]. At first the operator selects randomly two cut points along the individual. The genes between these cut points are called *mapping sections*. The offspring is created by copying the mapping sections from the first parent into the second offspring and the mapping section of the second parent into the first offspring. Afterwards, the remaining genes of the first offspring are filled with the genes from the first parent and the second offspring gets the genes of the second parent. Is a gene already present it is replaced according to the mappings.

Consider the following example with the two parents, where the numbers are IDs of the genes:

$$\text{parent 1: } (1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8)$$
$$\text{parent 2: } (5 \quad 2 \quad 7 \quad 1 \quad 3 \quad 8 \quad 6 \quad 4)$$

The cut points for the parents are for example as following:

$$\text{parent 1: } (1 \quad 2 \quad 3 \quad | \quad 4 \quad 5 \quad 6 \quad | \quad 7 \quad 8)$$
$$\text{parent 2: } (5 \quad 2 \quad 7 \quad | \quad 1 \quad 3 \quad 8 \quad | \quad 6 \quad 4)$$

The mapping sections are between the two cut points. For the first parents 4 5 6 and for the second parent 1 3 8. Now the two mapping sections are copied into the offspring and the other genes are at this moment unknown. The first offspring gets the mapping section of the second parent and the second offspring from the first parent. In this example the mappings are $4 \leftrightarrow 1$, $5 \leftrightarrow 3$ and $6 \leftrightarrow 8$

$$\text{offspring 1: } (x \quad x \quad x \quad 1 \quad 3 \quad 8 \quad x \quad x)$$
$$\text{offspring 2: } (x \quad x \quad x \quad 4 \quad 5 \quad 6 \quad x \quad x)$$

In the following step the first offspring is filled with genes of the first parent and the second offspring with genes from the second parent. The first gene for offspring 1 is the first gene of the parent 1, in this case the 1. However, there is already a 1 present in the offspring and the 1 that should fill the offspring must be replaced. Because of the mapping $4 \leftrightarrow 1$, the operator choose the first gene of offspring 1 to be a 4. The second and seventh gene of offspring 1 can be taken from the first parent. The third gene would be the 3, but this is already in the offspring. According to the mapping $5 \leftrightarrow 3$ the third gene of offspring 1 will be the 5. Finally the last gene of offspring 1 would be the 8, which is already present. Hence, the operator will choose according to the mapping $6 \leftrightarrow 8$ the 6 as the last gene.

$$\text{offspring 1: } (4 \quad 2 \quad 5 \quad 1 \quad 3 \quad 8 \quad 7 \quad 6)$$

Similarly the operator will create the second offspring as following:

$$\text{offspring 2: } (3 \quad 2 \quad 7 \quad 4 \quad 5 \quad 6 \quad 8 \quad 1)$$

- **Order Crossover:** This operator proposed by Davis [10] exploits that the order of the genes are important. Therefore, it select randomly two cut points along the parents. The sequence of genes between these two cut points are copied into one offspring at the same position. Now, the rest of the genes are copied in the order in which they appear in the other parent. Consider the following two parents:

$$\text{parent 1: } (1 \quad 2 \quad | \quad 3 \quad 4 \quad 5 \quad | \quad 6 \quad 7 \quad 8)$$
$$\text{parent 2: } (5 \quad 2 \quad | \quad 7 \quad 1 \quad 3 \quad | \quad 8 \quad 6 \quad 4)$$

At first, the segments between the cut points are copied into the offspring and the rest is unknown. The first offspring gets the segment of the first parent and the second offspring from the second parent.

$$\begin{array}{ll} \text{offspring 1:} & (x \quad x \quad 3 \quad 4 \quad 5 \quad x \quad x \quad x) \\ \text{offspring 2:} & (x \quad x \quad 7 \quad 1 \quad 3 \quad x \quad x \quad x) \end{array}$$

Now the offsprings are filled up with the genes from the other parent. The first offspring is filled with genes from the second parent and the second offspring with genes from the first parent. Starting from the second cut point the genes are copied into the offspring according the order in which they appear in the other parent, likewise starting from the second cut point. Genes are omitted that already exists in the offspring.

For the example the offspring will be:

$$\begin{array}{ll} \text{offspring 1:} & (7 \quad 1 \quad 3 \quad 4 \quad 5 \quad 8 \quad 6 \quad 2) \\ \text{offspring 2:} & (4 \quad 5 \quad 7 \quad 1 \quad 3 \quad 6 \quad 8 \quad 2) \end{array}$$

The first offspring got the segment 3 4 5 from the first parent. Starting at the second cut point, here behind the five, the offspring is filled up with genes from the second parent, likewise starting at the second cut point. Therefore, the 8 is the first gene in the second parent at the second cut point. The 8 is not present in the offspring and can be copied. Likewise the gene 6. The next candidate from the second parent is the 4 which is already present in the offspring. Therefore, it is omitted and also the next gene 5. Now, the next three genes 2 7 1 are copied into the offspring. The offspring is now filled up and the crossover is finished.

- **Position Based Crossover:** This operator was introduced by Syswerda [65] and starts by selecting a random set of positions in the parents. It exchanges the gene at the position of the selected parent on the corresponding position of the other parent. Afterwards the offspring $i (i = 1, 2)$ is filled with genes of the i-th parent. Already existing genes are omitted.

  Assume, following parents:

$$\begin{array}{ll} \text{parent 1:} & (1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8) \\ \text{parent 2:} & (5 \quad 2 \quad 7 \quad 1 \quad 3 \quad 8 \quad 6 \quad 4) \end{array}$$

  Further, the third, sixth and seventh positions are selected and the offspring will be:

$$\begin{array}{ll} \text{offspring 1:} & (x \quad x \quad 7 \quad x \quad x \quad 8 \quad 6 \quad x) \\ \text{offspring 2:} & (x \quad x \quad 3 \quad x \quad x \quad 6 \quad 7 \quad x) \end{array}$$

  Now, the first offspring is filled with genes from the first parent and the second offspring with genes from the second parent. In this example the first offspring is created straight forward, because no genes have to be omitted.

$$\begin{array}{ll} \text{parent 1:} & (1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8) \\ \text{offspring 1:} & (1 \quad 2 \quad 7 \quad 3 \quad 4 \quad 8 \quad 6 \quad 5) \end{array}$$

However, this leads to conflicts in the second offspring. The first and second gene can be copied from the parent. The third gene was exchanged with the other offspring. However, at the fourth position the first conflict occurs. Following the second parent the 7 must be inserted, but it already exists in the offspring. Therefore, this gene is omitted and the next one, in this case the 1, is copied to the fourth position of the offspring. Likewise, another conflict happened on the fifth position. According to the parent the 3 should copied, but it already exists in the offspring. Hence, this gene is omitted and the 8 is copied on this position. Another conflict happened on the last position. The 6 should be copied, but is already on the sixth position. Now, the 4 is inserted on the last position in the offspring. Finally, this leads to the second offspring.

$$\begin{aligned} \text{parent 2:} \quad & (5 \quad 2 \quad 7 \quad 1 \quad 3 \quad 8 \quad 6 \quad 4) \\ \text{offspring 2:} \quad & (5 \quad 2 \quad 3 \quad 1 \quad 8 \quad 6 \quad 7 \quad 4) \end{aligned}$$

4. **Mutation**

   After crossover the *mutation* introduces random changes into features of the individuals. Generally mutation is applied at the gene level [35]. It reintroduces genetic diversity back into the population, while crossover leads the population to converge by making the individuals alike [35]. Typically the mutation rate, which is the probability of changing the properties of a gene, is very small and the mutation acts only on one individual [25]. One or more genes in this individual are modified. There are many types of mutation [25], which will explained in the following:

   - **Reciprocal Mutation:** Two genes from the individual are selected randomly and exchanged.
   - **Inversion Mutation:** Between two randomly chosen cuts in the individual the genes are reversed in order.
   - **Insertion Mutation:** A new gene and a place where it should inserted is selected randomly.
   - **Boundary Mutation:** One gene in the individual is selected randomly and its value is replaced either by the user-defined upper bound or lower bound.
   - **Uniform Random Mutation:** The user defines at the beginning an interval and one gene is selected randomly and its value is replaced by a random number of that interval.
   - **Flip Bit Mutation:** This mutation can only be executed on binary representation of genes in an individual. One gene is randomly selected and its value is inverted. The 0 will be inverted to a 1 and vice versa.
   - **Displacement Mutation:** A randomly chosen block of genes is selected and moved to another location in the individual.

Termination Criteria

The process ends when a previous defined criteria is fulfilled. A simple approach is to terminate after a fix number of iterations [58]. However, there is no guarantee that this will lead to an optimal solution. Likewise, it is possible to define a bound for an objective value to terminate the process if this bound is reached. This technique requires the knowledge about the possible optimum of an objective value which is not given every time.

This thesis introduces suitable objectives, selection operators, mutation operators and crossover operators for test set prioritization for delta-oriented integration testing.

3 Related Work

Due to the high number of possible product variants SPL testing is a difficult task. To overcome this problem several testing approaches were proposed to reduce the number of product variants to test or select retestable test cases from the set of all test cases. Combinatorial interaction testing [7] was proposed to reduce the number of product variants to test and, hence, reduce the testing effort. This approach is based on the observation that most of the faults are expected to be caused by the interaction of a few features. Some approaches use *t-wise* testing to achieve combinatorial interaction between features [8, 55, 32]. In contrast, this thesis focus on prioritization of test cases and not on product variants. Other approaches exploit the commonalities between product variants to reduce redundancy in testing. Several approaches [57, 16, 59] use regression testing techniques, such as test set minimization, test case selection and test case prioritization [71]. Lachmann et al. [57] introduce an approach for delta-oriented integration testing to prioritize test cases based on changes in the architecture model. This thesis uses this approach to evaluate individuals. However, this thesis considers more than the changes. It provides objectives for dissimilarity between test cases and the coverage of pairwise feature combinations. Recently, some approaches use search-based techniques to prioritize or generate product variants of test cases. This approaches will be described in the following.

Al-Hajjaji et al. [24] introduce a similarity-based prioritization for product variants of a SPL. The order in which the product variants are tested is up-to the tester. In their approach Al-Hajjaji et al. want to increase the interaction coverage of the SPL under test as fast as possible over time. To achieve this goal they use the hamming distance [26] to evaluate the degree of similarity between configurations. The first product variant to test is the product with maximum number of features. The next product variant is selected that has the minimum similarity with all previously tested product variants. This means the second product variant is selected that has the minimum similarity to the first product variant. Afterwards the third product variant is selected that has the minimum similarity to the previous two product variants, and so on. The underlying idea for this approach is that similar product variants are likely to contain similar defects. They have shown that their similarity-based prioritization detects faults earlier than the default order of the algorithms they have used. However, Al-Hajjaji et al. prioritize product variants and just with a similarity-based approach.

Henard et al. [27] proposed an approach to generate product variants with regards to multiple objectives. The approach avoids invalid product variants and is capable of dealing with multiple objectives. The objectives are maximization of pairwise feature coverage, minimizing the number of product variants selected and minimizing the overall cost of the test set. The test set is the set of product variants to test. They introduce a GA which solves the multi-objective optimization problem. They have shown that the objectives are better fulfilled with their approach than with a random set of generated product variants. Henard et al. use a multi-objective GA to generate product variants.

The approach from Wang et al. [70] prioritizes test cases with respect to their costs and effectiveness. They consider two different parts as costs, the execution time and the cost for setting up resources. The effectiveness measures consists of three elements in their approach. The first element *prioritized extent* measures to what extent test cases can be prioritized for execution before all available test resources are allocated. The element *feature pairwise coverage* measures the pairwise feature coverage of test cases. Finally, the third element *fault detection capability* measures the fault detection capability achieved by the test cases. The approach from Wand et al. [70] prioritize test cases with respect to multiple objectives.

Devroey et al. [12] introduce an approach to test the behavior of a SPL. They use a *Feature Transition System* (FTS) to represent the behavior and generate test cases from this model. Their approach tries to maximize the product and the behavior coverage of a set of test cases. For this they use a dissimilarity distance between test cases based on the products able to execute each test case and the actions used in those test cases. To compute the dissimilarity they use several approaches. The product dissimilarity is computed with the Jaccard distance. For the actions dissimilarity they used several dissimilarity distances, e.g., Hamming, Jaccard, Levenshtein, and evaluate the best measurement for their purpose. They have shown that test cases generated with respect to their dissimilarity are better than random ordering or with respect to all-actions coverage, i.e., every action in the FTS must at least once covered, generated test cases. Between different types of dissimilarity distances they have shown that Hamming and Jaccard-based distances are the most efficient. Devroey et al. generate test cases for behavioral testing of a SPL with respect to dissimilarity.

Baller et al. [3] present an approach to optimize test suites for SPLs with respect to cost and profit objectives. A test suite for their approach is a subset of all possible test cases for the SPL. They do not only consider test cases, but also test requirements, products and their interrelations. Additionally, their approach produces an optimized testing order on the set of product variants. The optimization of the test suite is a minimization problem with respect to cost and profit requirements. The approach tries to find a test suite which satisfies each test requirement but with a minimum of overall costs. Likewise, another optimization goal is to find a minimal test suite which satisfies a profit goal. Every test requirement has a profit value. The overall profit of the test suite is the sum of profits of all test requirements satisfied by the test suite. Baller et al. introduce an approach which minimize a test suite with respect to cost and profit objectives. They define this minimization for single products and for product families.

In contrast to the approaches from Al-Hajjaji et al. [24] and Henard et al. [27], this thesis prioritizes test cases with respect to several objectives. This thesis assumes that the product variants to test are already generated and ordered in a efficient way. The idea of the evaluation of similarity not just with the previous product variant but with all previously tested product variants from Al-Hajjaji et al. [24] is adapted to evaluate the dissimilarity of test cases. The approach from Wang et al. [70] does not use the delta-oriented approach and considers other objectives than this thesis. Devroey et al. [12] use a single-objective approach to generate test cases. In contrast, this thesis prioritize test cases with respect to several different objectives where one of these objectives is the dissimilarity between test cases. Likewise, this thesis does not consider the behavior on component level because it focus the integration test. In contrast to the approach from Baller et al. [3] this thesis does not minimize a test set because this process was accomplished before with another approach. Likewise, this thesis has different objectives to optimize and is performed for the delta-oriented approach.

4 Concept

This chapter introduce the general concept of objectives of this thesis. To prioritize a test set a GA needs a fitness function to evaluate how good the test set is. The fitness function is an important part in the process of a GA and can lead to bad solutions if the function is not well defined. We introduce four different concepts and their fitness functions to prioritize test sets. The first two objectives are alternatives to each other and focus the coverage of pairwise feature combinations. The third objective focus the coverage of changes in the architecture of a product variant and the fourth objective aims to cope with clustering of test cases in the test set.

In the thesis we assume that a GA prioritize only test sets with retestable test cases. If we refer to a test set we mean the set of retestable test cases for a product variant.

4.1. Maximization of Feature Combinations Covered

In SPL testing most of the faults occur due to the interaction between a small number of features [7]. This observation is exploited by sampling methods to reduce the testing effort. Some sampling methods use t-wise testing where combinations of t features are used to generate product variants. Each t-wise feature combination occurs at least once in the final set of product variants to test. Pairwise or 2-wise testing was introduced by Cohen et al. [7] and is prevalent in sampling methods. Four combination capabilities are possible for feature combinations in pairwise testing. The combination capabilities are that none of the two features are in a product variant, only one of the two features and both features together. This combinations are shown in Table 4.1.

Table 4.1.: Possible combination capabilities of features

Feature A	Feature B
X	
	X
X	X

This thesis adapts the underlying idea of sampling and evaluate the given test set from the GA with respect to pairwise feature combinations. The earlier test cases in a test set cover *retestable feature combinations* the better is a test set. A feature combination is retestable, if it was covered with a new test case in a previous product variant and not by a retestable test case. Currently the mapping between test cases and features is not given. However, deltas are assigned with the information for which feature or feature combination they are applied to the core. This is specified in the application condition of a delta. Likewise the knowledge about deltas for a product variant and test cases for each product variant are given. The relationship of this informations is shown in Figure 4.1.

The feature model $fm \in FM$ is given to specify which features are possible and describes the constraints and relationships between features. All product variants to test are given which where

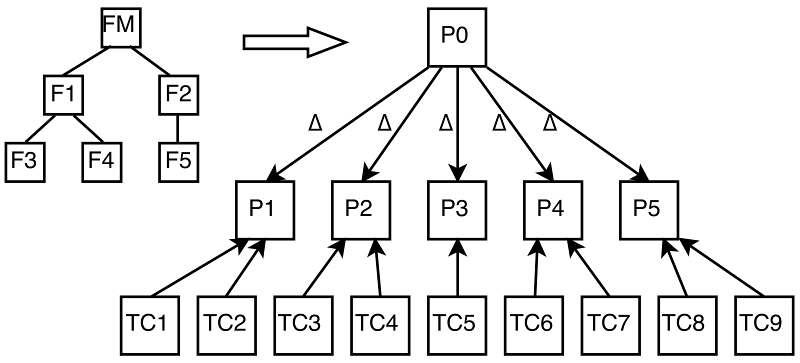


Figure 4.1.: Coherences between Deltas and Test Cases

selected by sampling. In the example in Figure 4.1 the product variants are $\{P1, ..., P5\}$. Likewise, the order in which the product variants are tested is given. Product *P0* is assumed as the core of the SPL. The information about the deltas which have to be applied to the core to generate a product variant is available. These deltas specify which components, signals and connectors have to be added or removed from the core. As said earlier, deltas are matched by the application condition to a feature or a feature combination. Likewise, techniques exists which generate test cases to cover new interactions of components for product variants [37]. Therefore, the test cases $\{TC1, ..., TC9\}$ are automatically mapped to a product variant. For the mapped product variant the assigned test cases are new and can be reused for next product variants.

Figure 4.2 shows the different steps for the evaluation of a test set with respect to pairwise feature combinations. The first step is the mapping between test cases and features. A test case can cover connectors from multiple deltas. In this case a test case which covers multiple deltas will be mapped to multiple application conditions. An application condition can be comprised as a combination of different features. Since this thesis focus on pairwise feature combinations the mapped features have to be separated into pairwise feature combinations. In the end the quality of a test set is measured. A test set got a higher rank if retestable feature combinations are tested earlier. This three steps are described in the following subsections.

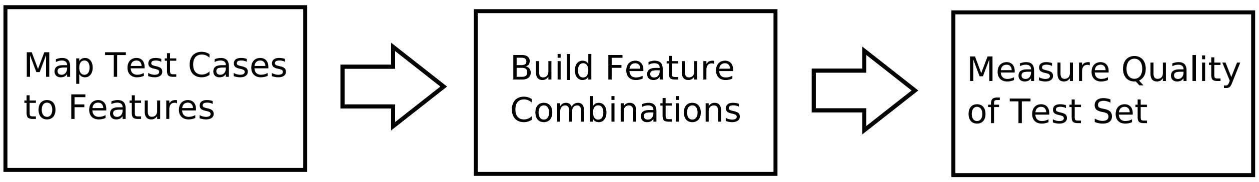

Map Test Cases to Features
Build Feature Combinations
Measure Quality of Test Set

Figure 4.2.: Concept for Evaluation of Feature Combinations

4.1.1. Mapping between Test Cases and Features

Currently a mapping between features and test cases did not exist. This mapping is introduced in this thesis. Nevertheless, a delta has an application condition which specifies for which features the delta is applied on the core and, hence, a mapping to features. To map test cases to features it is possible to exploit the mapping between deltas and features by comparing the described connectors in a test case with connectors in a delta. If a test case covers a connector which is also in a delta this test case covers connectors of the feature combination given in the application condition. In this thesis we consider positive integration testing and, hence, only elements which are present in the current architecture are tested. Thus, only delta operations are considered which add connectors to the system, as test cases can only cover present connectors not absent ones. It is possible that a signal of a connector is used in multiple parts of the architecture. Therefore, a connector from a test case and from a delta is only mapped if the source components, the signals and the destination components are exactly the same.

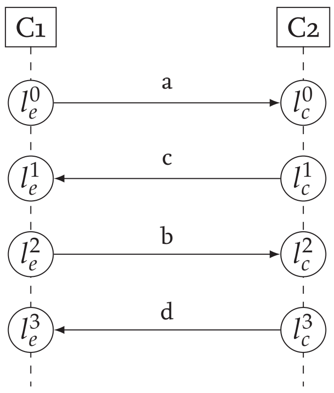


Figure 4.3.: Sample Test Case

Listing 4.1.: Sample Deltas

```
DELTA1 when 'Feature A' {
addconnector {
C1,a,C2
C2,c,C1
}
}

DELTA2 when 'Feature C' {
addconnector {
C3,e,C1
C4,f,C3
}
}
```

Example 4.1.1: Example for Mapping of Test Cases and Features

To illustrate the content of deltas and test cases assume the given test case in Figure 4.3 and the deltas in Listing 4.1.
The test case in Figure 4.3 describes an interaction between component *C*1 and component *C*2 and covers four connectors with four signals $\{a, b, c, d\}$. Deltas *DELTA*1 and *DELTA*2 are given in Listing 4.1. After an identifier for the delta the keyword *when* indicates the beginning of the application condition. The first delta is applied for *Feature A* and the second delta for *Feature B*. The specification of all operations for this delta are limited by two „braces“. Keyword *addconnectors* indicates that the connectors between the next two braces are added to the architecture. As said earlier the first delta *DELTA*1 in Listing 4.1 is applied if the feature *Feature A* is selected for the current product variant and adds two connectors to the architecture. The second delta *DELTA*2 in Listing 4.1 is applied for feature *Feature C*. DELTA2 contains two connectors which are added to the architecture.
Comparing the connectors from the test case in Figure 4.3 with the added connectors of the first delta we observe that there is a match between two connectors of the test case. Connector *C*1,*a*,*C*2 and connector *C*2,*c*,*C*1 are given in the test case and in the delta. This indicates that the test case can be mapped to the features in the application condition. In this case, *Feature A* is mapped to this test case. Comparing with the second delta there is no match between any connectors. This means this test case covers nothing of the delta and, hence, nothing of the feature *Feature C*.

A special case for this mapping are the connectors which are defined in the core product variant. If a covered connector from a test case is defined in the core architecture no mapping between the features of the core product and the test case is possible. This is due to the fact that there is no explicit mapping between connectors in the core architecture and features in the core product like it is given in deltas.

4.1.2. Building Feature Combinations

The information of the previous step where features were mapped to test cases is used to build pairwise feature combinations. At first we will describe how a mapping for multiple deltas and its application condition is handled and afterwards how pairwise feature combinations can be build from the resulting application condition.

It is possible that test cases are mapped to more than one delta. This means these test cases cover every feature combination given by the application conditions of the deltas. The combination of multiple application conditions is accomplished by conjunct all conditions with the boolean *and*-operator. This means if a test case is mapped to an application condition with *Feature A* $\wedge$ *Feature B* from one delta and the application condition with *Feature C* $\wedge$ $\neg$*Feature D* from another delta these two conditions are connected to *Feature A* $\wedge$ *Feature B* $\wedge$ *Feature C* $\wedge$ $\neg$*Feature D*.

Additionally, to the mapped application conditions the feature configuration is used to receive the features which are not selected for the current product variant. With the information of absent features more feature combinations can be covered. This is possible because we assume that features from an application condition can be considered in isolation.This means if a test case is mapped to delta which is applied for an application condition *Feature A* $\wedge$ *Feature B* it is possible to map this test case to every feature combination with *Feature A* $\vee$ *Feature B*. These absent features are also conjuncted with the *and*-operator to the application condition.

After the conjunction of all application conditions to one boolean expression we can build pairwise feature combinations. However, it is possible to have a mixture of *and*- and *or*-operators in application conditions which is a challenge for building feature combinations. To cope this challenge it is possible to replace the *or*-operator with the *and*-operator by considering the feature configuration of the product variant and the truth table of the *or*-operator. This makes it easier to build feature pairs.

For the *or*-operator three possible combinations lead to the value *TRUE* which are shown in Table 4.2:

Table 4.2.: Truth table of Or-Operator

Feature A	Feature B	Result
False	False	False
True	False	True
False	True	True
True	True	True

Since we want to build pairs of features we have to define which combination of the truth table is covered by a test case. If a test case covers every combination which is true in the truth table this would lead to the following boolean expression: *(Feature A* $\wedge$ $\neg$*Feature B)* $\wedge$ *(*$\neg$*Feature A* $\wedge$ *Feature B)* $\wedge$ *Feature A* $\wedge$ *Feature B*. But this lead to a contradiction. In the first expression Feature A is true this means it is in the current product variant and Feature B is false and, hence, not in the current product variant. In the second expression it is the opposite of the first expression. Feature A is not in the current product variant and Feature B is. But this is a contradiction to the first expression. Therefore, it is not possible that a test case covers every combination which is true in the truth table.

If a test case covers a delta with an application condition that contains the boolean operator *or* the feature configuration of the product variant is important. The feature configuration contains the features that were selected for this product variant. With the information of the feature configuration it is possible to check which boolean expression of the *or*-operator is true. This is an expression which contains the *and*-operator. If Feature A is in the current product variant and Feature B not expression *Feature A* $\vee$ *Feature B* in the application condition is replaced by expression *Feature A* $\wedge$ $\neg$*Feature B*.

Example 4.1.2: Example for Building Pairs of Features

Assuming that the current product variant has the following feature configuration: *Feature A, Feature B, Feature C*. A test case for this product variant is mapped to a delta with the application condition *Feature A* $\wedge$ *Feature C*. Three combinations are possible

- *Feature A* $\wedge$ *Feature C*
- *Feature A* $\wedge \neg$*Feature C*
- $\neg$*Feature A* $\wedge$ *Feature C*

Considering the feature configuration of this product variant only the expression *Feature A* $\wedge$ *Feature C* is true. Because *Feature A* and *Feature C* are selected for the considered product variant. Therefore, the expression *Feature A* $\vee$ *Feature C* is replaced by this expression *Feature A* $\wedge$ *Feature C*. Now it is easier to build pairs of features because the resulting expression only contains the *and*-operator.

After replacing the *or*-operator and the conjunction of all mapped application conditions pairwise feature combinations can be built. In Example 4.1.2 we have shown sample application conditions for deltas. However, application conditions can be complex boolean expressions, e.g., *Feature A* $\wedge$ *Feature B* $\wedge\neg$ *Feature C*. Therefore, we have to define how we cope with complex boolean expressions. Pairwise feature combinations of complex boolean expressions are built by building every possible pair of combination which is possible with the features of the expression. For the expression *Feature A* $\wedge$ *Feature B* $\wedge\neg$ *Feature C* this will lead to the pairwise feature combinations: *Feature A* $\wedge$ *Feature B*, *Feature A* $\wedge\neg$ *Feature C*, *Feature B* $\wedge\neg$ *Feature C*. For this approach the negation of a feature belongs to a feature and can not be separated from it.

For feature combinations the commutativity of boolean expressions is exploited. This means for the combination of two features it is not important in which order they appear. The commutativity of two elements a, b is defined as:

$$a \wedge b = b \wedge a$$
$$a \vee b = b \vee a$$

According to Table 4.1 four feature combinations are possible. One possible combination for a test case is that two features are negated which means both features are not in the product variant. For the evaluation of a test set this scenario is only applicable if this combination is given by application conditions. Deltas describe for which feature or feature combination they are applied on the core product and there is not every time a coherence between not selected features for this product variant. To map test cases with features we use the deltas. If no delta is defined for an absent feature it is not possible to map a test case to an absent feature. The application condition can describe that a delta is applied if features are not in the product variant. But this is not given for all combinations of two absent features. Therefore, this combination is only applicable if application conditions of deltas describes explicit that they are applied for features not in the product variant.

Other possible combinations for a test case are that one of the two features is negated and the other not. This means one feature is in the current product variant and the negated feature not.

The last possible combination is that both features are not negated which means they are selected for the current product variant.

Example 4.1.3: Example for Mapping of Test Cases and to absent Features

Considering the condition $Feature\ A \land Feature\ B \land \neg Feature\ D \land \neg Feature\ E$ we can build following pairwise feature combinations:

- $Feature\ A \land Feature\ B$
- $Feature\ A \land \neg Feature\ D$
- $Feature\ A \land \neg Feature\ E$
- $Feature\ B \land \neg Feature\ D$
- $Feature\ B \land \neg Feature\ E$
- $\neg Feature\ D \land \neg Feature\ E$

As we can see a test case mapped to the application condition $Feature\ A \land Feature\ B \land \neg Feature\ D \land \neg Feature\ E$ can cover multiple pairwise feature combinations. This test case can cover that two features are selected for a product variant, that one feature is selected and the other not and it can cover that both features are absent in the current product variant.

After connecting all application conditions for a test case and then building pairwise feature combinations the quality of the test set can be measured. The next subsection will introduce a metric to measure the quality.

4.1.3. Measuring the Quality of Test Set

After mapping test cases with features and separating application conditions to pairwise feature combinations the GA needs a metric to evaluate a test set. As said earlier, a test set got a higher rank if test cases which test new feature combinations are tested early. The fitness function $fit_{FC} : \mathcal{P}(\mathcal{TC}) \rightarrow [0, 1]$ measures at which position T_{FC_i} in the ordered test set connectors of new feature combinations are tested. It is possible that a test case did not cover every connector of a feature combination, hence, the position $T_{featureComb_i}$ is multiplied with a factor which describe how much connectors of all connectors of this feature combination are covered by this test case. This is the quantity of connectors $|DCON_{TC_j}|$ the test case covers divided by all added connectors $|DCON|$ in the delta of the correspondent application condition. If a test case covers k deltas the factor $\frac{|DCON_{TC_j}|}{|DCON_j|}$ is calculated for these k deltas and summarized. All positions are summarized and normalized over n test cases times m connectors in all covered deltas. This metric is based on the *Average Percentage of Faults Detected (APFD)* metric introduced by Rothermel et al. [18]. The APFD measures the quality of a prioritized test set with respect to the fault detection rate. Prioritized test sets get a higher rank if they cover faults early.

$$fit_{FC} = 1 - \frac{\sum_{i=1}^{n} T_{FC_i} \cdot \sum_{j=1}^{k} \frac{|DCON_{TC_j}|}{|DCON_j|}}{n \cdot m} + \frac{1}{2 \cdot n} \tag{4.1}$$

Example 4.1.4: Example for Feature Combinations Coverage

Assumed that we have two retestable test cases $\{TC_1, TC_2\}$ and one delta with five connectors. Test case TC_1 covers three of the five connectors from the delta and TC_2 covers the least two connectors. If the prioritized test set is $TS = \{TC_1, TC_2\}$ this means test case TC_1 is tested before TC_2 fit_{FC} is computed as follows:

$$fit_{FC} = 1 - \frac{1 \cdot \frac{3}{5} + 2 \cdot \frac{2}{5}}{2 \cdot 5} + \frac{1}{2 \cdot 2} = 0,86$$

If we change the order of the test cases to $TS = \{TC_2, TC_1\}$ where only two of five connectors are tested in the first test case fit_{FC} is computed as follows:

$$fit_{FC} = 1 - \frac{1 \cdot \frac{2}{5} + 2 \cdot \frac{3}{5}}{2 \cdot 5} + \frac{1}{2 \cdot 2} = 0,84$$

As we can see a test set where more connectors are tested earlier got a higher value for fit_{FC} than a test set which test less connectors early.

The next Subsection will introduce another technique for feature combinations with a history-based approach.

4.1.4. Minimization of History-based Feature Combinations Covered

The previous objective maximization of covered feature combinations considers a feature combination only in the first product variant where it is retested. This prioritization can lead to a test set which detects faults late because it is possible that a previous considered feature combination lead to a failure with a current considered feature combination. To cope with this problem a history-based approach which prioritizes test sets with aid of informations from previous tested product variants is used to prioritize test sets. The minimization of history-based feature combinations coverage objective is an alternative to the maximization of covered feature combinations coverage objective.

History-based testing is a regression testing technique to prioritize test cases [14]. The history-based technique uses information from previous testing processes to prioritize test cases. These information might be the fault detection rate from a test case for previous executions, the execution time or how often a test case was tested in the previous iterations.

For this thesis we rank test cases higher which covers pairwise feature combinations which were not often tested in previous product variants. The assumption of this idea is, that a feature combination which was not often covered in previous product variants has a higher probability to lead to faults with other feature combinations. If a feature combination was covered often it was executed with multiple other feature combinations and the probability is inferior that it leads to faults with other feature combinations.

The first two steps where test cases and features are mapped and feature combinations are build are the same, but measuring the quality will now consider how often a feature combination was retested compared to the maximization of covered feature combinations. Additionally, this objective will be minimized and not maximized. The previous object for feature combinations aims to cover early as much feature combinations as possible. In contrast, the history-based feature combinations approach wants to cover often covered feature combinations late. This means if a feature combination was only covered once for a product variant it receives a higher rank than a feature combination which was covered in three previously tested product variants. The history-based feature combination approach will retest every feature combination covered by test cases of the product variant but with different priorities which indicates the times the feature combination was covered in previous product variants.

The fitness function $fit_{HFC} : \mathcal{P}(\mathcal{TC}) \rightarrow [-1, 1]$ measures at which position T_{FC_o} a feature combination is retested and multiplies it with a weight $w_{FC_o} = Q_{FC_o}$ where Q_{FC_o} indicates how often a feature combination was tested in the set of previously tested product variants P_{tested}. Since only retestable test cases are executed in our approach, Q_{FC} is only incremented for feature combinations covered by retestable test cases. Test cases which are only reusable but not retestable will not affect Q_{FC} because they will not be tested and, hence, can not reveal faults with other feature combinations.

It is possible that a test case covers more than one feature combination for that all weights w_{FC_o} of the l covered feature combinations are summarized. The function fit_{HFC} summarizes the position of test cases multiplied with the covered delta connectors and the weights for the retested feature combinations and normalizes over n test cases times the highest summarized weight $w_{FC_{max}} = max(\sum_{o=1}^{l} w_{FC_o})$ for a test case in the test set.

$$fit_{HFC} = 1 - \frac{\left(\sum_{i=1}^{n} T_{FC_i} \cdot \left(\sum_{j=1}^{k} \frac{|DCON_{TC_j}|}{|DCON_j|}\right) \cdot \left(\sum_{o=1}^{l} w_{FC_o}\right)\right)}{n \cdot w_{FC_{max}}} + \frac{1}{2 \cdot n} \tag{4.2}$$

Example 4.1.5: Example for History-based Feature Combinations Coverage

Assumed that a test set with two retestable test cases TC_1, TC_2 is evaluated. TC_1 covers two connectors of a delta with 3 connectors and covers the feature combination $A \land B$ which is covered the second time. TC_2 covers two connectors of another delta with 3 connectors and covers the feature combination $A \land C$ which is covered for the third time. The evaluated test set is $TS_1 = \{TC_1, TC_2\}$. fit_{HFC} is computed as follows:

$$fit_{HFC} = 1 - \frac{1 \cdot \frac{2}{3} \cdot 2 + 2 \cdot \frac{2}{3} \cdot 3}{2 \cdot 3} + \frac{1}{2 \cdot 2} = 0.36$$

If the order of the two test cases is changed to $TS_2 = \{TC_2, TC_1\}$ where the test case TC_2 is executed before TC_1 fit_{HFC} is computed as follows:

$$fit_{HFC} = 1 - \frac{1 \cdot \frac{2}{3} \cdot 3 + 2 \cdot \frac{2}{3} \cdot 2}{2 \cdot 3} + \frac{1}{2 \cdot 2} = 0,47$$

The test set TS_1 where test cases which covers not often tested feature combinations earlier got a lower value for fit_{HFC} than the test set TS_2 where test cases which covers later not often tested feature combinations.

The next section introduces the third objective for this thesis.

4.2. Maximization of Changes Covered

Commonalities and variabilities of the product variants can be used to reduce the testing effort. Lachmann et al. [57] introduced an approach to use delta information to select and prioritize test cases. These information can be used to identify changed elements in the architecture and test only changed elements because unchanged elements were tested before. We assume that detected faults in previous product variants are fixed before a new product variant is tested. Changes in the architecture which were not tested before are more likely to introduce failures to the already tested system parts. Therefore, a test set should cover changed elements as early as possible to detect failures earlier. Changed elements are defined as the set of connectors $CCON_{p_j} \subseteq CON_{p_j}$ which are connected to components with changed interfaces after applying a delta [57]. As said earlier in subsection 2.1.3 components are connected by connectors. The interface of a component is the set of incoming and outgoing connectors. Changes on the architecture for the current product variant are only considered if the current interface of the component has not been tested in any previously tested product variant. For this purpose regression deltas are computed which specify the differences between product variants. For the integration test regression deltas specify the differences between architectural models. Additionally, connectors added for the first time are not considered because new connectors are only covered by new test cases and will be tested anyway.

Figure 4.4 shows an example architecture and a changed architecture after a delta is applied on the first one.

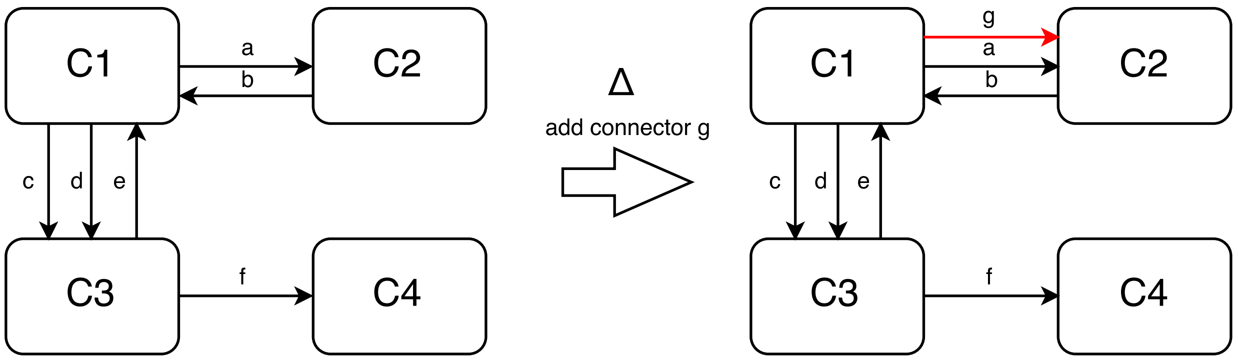


Figure 4.4.: Changed architecture after applying a delta

The example architecture contains four components $\{C1, C2, C3, C4\}$ and six connectors $\{a, ..., f\}$. In the example, the delta adds the connector g between component $C1$ and $C2$. Changed elements in an architecture are the set of all connectors which are connected to components with changed interfaces. In this example component $C1$ and $C2$ have changed interfaces. Thus, all connectors $\{a, b, c, d, e\}$ will be retested. Connector f is not in the set of changed parts, because component $C3$ and $C4$ have no changed interfaces. Additionally, connector g is a new connector and is tested by a new test case and is not covered here.

To measure the quality of their approach Lachmann et al. [57] introduced a new metric the *Average Percentage of Changes Covered (APCC)*. They assume that changes of a product variant compared to all previous tested product variants are more likely to introduce failures to already tested system parts. The metric $fit_{APCC} : \mathcal{P}(\mathcal{TC}) \rightarrow [0, 1]$ measures at which position T_{change_i} in an ordered test set a changed connector i $change_i \in Change(p_j)$ from product variant j is covered by the respective test case at position T. The positions are summarized and normalized over n test cases times the number m of overall changes to cover.

$$fit_{APCC} = 1 - \frac{\sum_{i=1}^{m-1} T_{change_i}}{n \cdot m} + \frac{1}{2 \cdot n} \tag{4.3}$$

Example 4.2.1: Example for Changes Covered

Assuming that we have two test cases TC_1 and TC_2. The first test cases TC_1 covers three changed connectors and the second test cases TC_2 one changed connector. The first test set $TS_1 = \{TC_1, TC_2\}$ has the following value for fit_{APCC}:

$$fit_{APCC} = 1 - \frac{1+1+1+2+2}{2 \cdot 5} + \frac{1}{2 \cdot 2} = 1 - \frac{7}{10} + \frac{1}{4} = 0.3$$

For test set $TS_2 = \{TC_2, TC_1$ fit_{APCC} is computed to:

$$fit_{APCC} = 1 - \frac{1+1+2+2+2}{2 \cdot 5} + \frac{1}{2 \cdot 2} = 1 - \frac{8}{10} + \frac{1}{4} = 0.2$$

The test set TC_1 where more changes are covered with the first test cases got a higher value for fit_{APCC} than test set TC_2 where the second test case covers more changes.

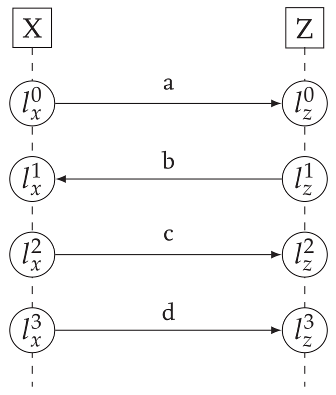
This metric is used to prioritize the test sets given by the GA with respect to changes in the architecture. The next section introduces an objective which aims to order test cases in a way that the resulting test set is more diverse.

4.3. Maximize Test Case Dissimilarity

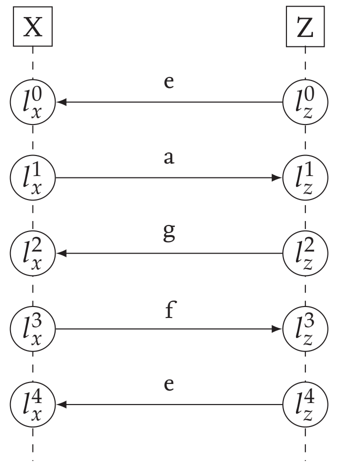
For software testing the order of test cases in the test set is important. Similar test cases have the similar capability of revealing faults [5]. As a consequence it is important to order the test cases in a way that the resulting test set is diverse. In the process of integration testing the interaction between components is tested. Therefore, it is possible that test cases tests the same connectors but in a different order. Likewise, it is possible that an interaction of components needs a pre- or postcondition in form of certain connectors. This can lead to redundancy between test cases. A solely weight-based prioritization of test cases in integration testing can lead to clusters of similar test cases because they have similar weights. This might decrease the fault detection rate as testing focus on the same parts of the system. Therefore, in this thesis one objective for the GA is the dissimilarity of the test cases. For this the principle of similarity-based testing is adapted.

Similarity-based test case selection is a strategy for model-based testing to select test cases in a way that the resulting test set is more diverse. The underlying idea of this strategy is that the fault revealing potential is higher if test cases in the test set are more diverse [5]. To measure the similarity or dissimilarity between a set of test cases we need a (dis)similarity function.

The two main dissimilarity functions are the *Hamming Distance* [17] and the *Jaccard Distance* [56]. The hamming distance is one of the most used distance functions. It is a simple edit-distance [26], which is defined between two sequences as the minimum number of edit operations, e.g. insertions, deletions, to transform the first sequence into the second sequence. The hamming distance is only applicable on sequences with identical length which is difficult in most applications because the compared sequences have different lengths. However, it is possible to force the sequences to have the identical length. Therefor, a binary vector with the length of every possible element of the sequences indicates which element exists in the input.



(a) Test Case A



(b) Test Case B

Figure 4.5.: Two compared test cases A and B

For the two sample test cases in Figure 4.5a and Figure 4.5b the vector for the hamming function would be:

Example 4.3.1: Example for Hamming Distance

Vector:(a, b, c, d, e, f, g)

The vector is a joined set of all connectors which are covered by the test cases. Since, this test cases covers an interaction between same components X and Z only the signal is considered.

Test Case A:$(1, 1, 1, 1, 0, 0, 0)$
Test Case B:$(1, 0, 0, 0, 1, 1, 1)$

As we can see in Figure 4.5 test case A covers the connectors $\{a, b, c, d\}$. Therefore, the vector for test case A indicates that these four connectors are in the set of covered connectors. Test case B covers the signals $\{a, e, f, g\}$ which is indicated in the vector for test case B. Using this trick we can use the Hamming distance even if the number of signals differs between the test cases.

To transform the first sequence of test case A to the sequence of test case B six operations are necessary. The signals *b,c,d* must be deleted and the signals *e,f,g* must be inserted into the sequence. This means the hamming distance for this two test cases is 6.

However, Hemmati et al. [26] show in an experiment that the jaccard distance has a better computation time than those of other metrics, e.g. Hamming distance, Levenshtein [21]. Likewise it does not need inputs with identical length and, thus, no vector which represents all elements of the inputs.

The jaccard distance $distance_j : A \times A \to [0,1]$ computes the dissimilarity between two inputs. It is defined as follows:

$$distance_j(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|} \tag{4.4}$$

A jaccard value of 0 indicates that the two compared inputs are exactly the same. For this thesis test cases are compared which are described as sequence of connectors. Due to the fact that it is possible that signals are used in different parts of the architecture two connectors are only identical if the same signal is exchanged between the same components in both compared test cases. Likewise, it is important how often a connector occurs in a test case. The process of integration testing tests the interaction between components. Therefore, it is possible that a test case contains connectors multiple times to cover a certain interaction between components. It is possible that a test case covers another interaction and uses the same connectors as well. For this case the quantity of connectors is important. The following test cases in Figure 4.6 show a different interaction between components with the exact same connectors.

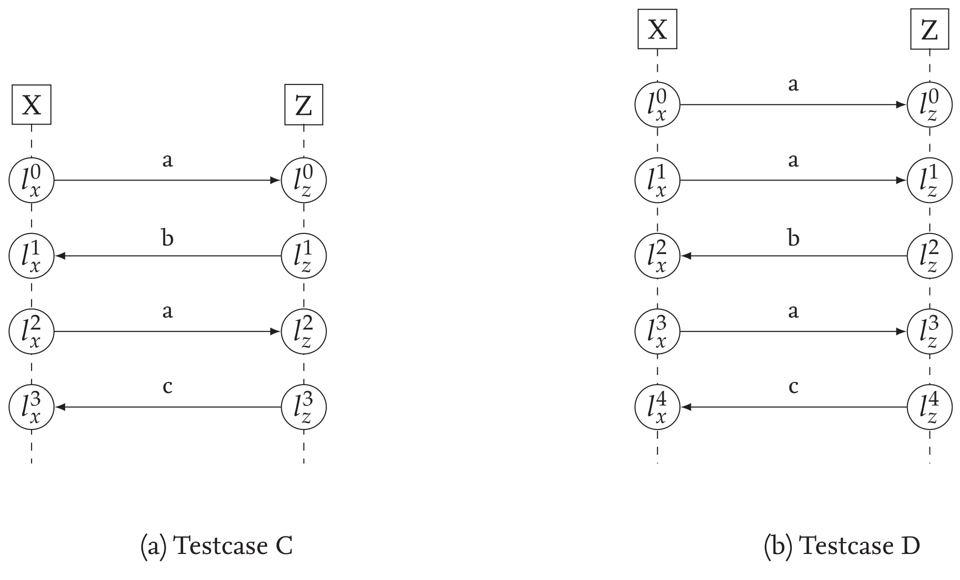


Figure 4.6.: Two compared Test Cases C and D

The jaccard distance measures the dissimilarity between finite sets. Which considers only the existence of an element and not the quantity of occurrences of an element. For this test cases the jaccard distance will calculate that they are exactly the same.

Example 4.3.2: Example for Jaccard distance

The two sample test cases in Figure 4.6 exchange the identical connectors, but the message sequence chart in Figure 4.6b covers a new interaction between the components. Therefor the sample test case D exchanges the connector with signal *a* multiple times. The calculation of the jaccard distance following the definition of sets indicates that these two test cases are identical. The two test cases are defined as sets

$$C = \{a, b, a, c\}$$
$$D = \{a, a, b, a, c\}$$

These two sets describe the exchanged signals of the test cases. For this example the components are ignored because they are the same, in practice we have to compare the whole connector with source component, exchanged signal and target component. The intersection for this two sets is

$$C \cap D = \{a, b, c\} \text{ with } |C \cap D| = 3$$

The intersection contains the signals $\{a, b, c\}$ because these three signals occur in both test cases.
The union contains every signal that occurs in the test cases. In this case both test cases exchange the same signal and, hence, the union and the intersection are identical.

$$C \cup D = \{a, b, c\} \text{ with } |C \cup D| = 3$$

With the information of the cardinalities of the intersection and the union the jaccard distance is calculated as

$$distance_j(C, D) = 1 - \frac{|C \cap D|}{|C \cup D|} = 1 - \frac{3}{3} = 0$$

In fact, these two test cases are not identical and the we have to modify the jaccard distance for this thesis. Thus, we use multi sets to calculate the jaccard distance for test cases. A multi set is a set where elements can occur multiple times and can be described as a pair (M, r_S) where M is the set of elements and $r_S : M \to \mathbb{N}$ indicates how often an element from M occurs in the multi set [1]. The intersection in multi sets is defined as follows [1]:

$$A \cap B := (S, r_{A \cap B}) \text{ with } r_{A \cap B} := \min(r_A, r_B)$$

This means that the intersection of two multi sets is defined as a multi set where $r_{A \cap B}$ is the minimum of the cardinalities from an element in set A and B. And the union is defined as [1]:

$$A \cup B := (S, r_{A \cup B}) \text{ with } r_{A \cup B} := \max(r_A, r_B)$$

The union is the maximum of the cardinalities from an element in set A and B.

Example 4.3.3: Example for Multi Sets

Assuming the two multi sets

$$A = \{a, b, a, c\}$$
$$B = \{a, c, d\}$$

Following the definition of the intersection we get the set $\{a, c\}$. Looking at element a we see that this element is twice in set A and once in set B. The minimum of this two cardinalities is one. Therefore, the element is once in the intersection. Element b is only in set A and not in set B and, hence, not in the intersection. Likewise, element d is only in set B and not in set A. At last, element c is once in set A and in set B.
The union is the set $\{a, a, b, c, d\}$. For the union the maximum of the two cardinalities is used. As said before element a is twice in set A and once in set B. Therefore, element a is twice in the union. The other elements $\{b, c, d\}$ are once in each set and, hence, only once in the union of the two sets.

The calculation of the jaccard distance is used to compare two test cases in a test set. The objective for the GA did not only compare two test cases. The fitness is calculated for the entire test set. Therefore, the computation of dissimilarity has to be extended compared to the definition of jaccard. Each test case in the test set is compared with all test cases which are previously in the test set. Figure 4.7 shows the procedure of computation.

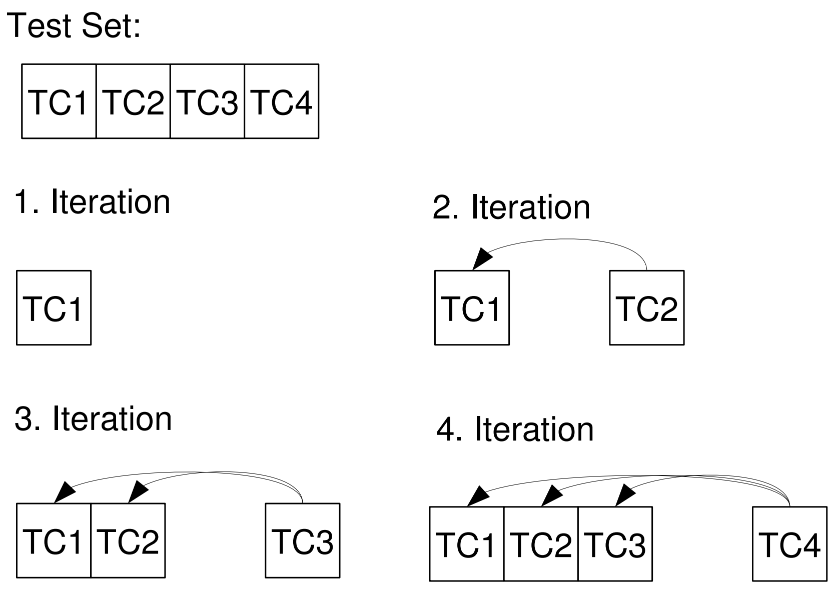


Figure 4.7.: Computation of the Dissimilarity of Test Sets

The test set $TS = \{TC1, TC2, TC3, TC4\}$ in Figure 4.7 consists of four test cases. In the first iteration the first test case $TC1$ in the order is evaluated with respect to his dissimilarity. In this case, no evaluation is needed because no previous test cases exists. In the second iteration the second test case is compared with all previous test cases. Only $TC1$ is in the set of previous test cases and, hence, $TC2$ is only compared with $TC1$. The third iteration shows, that $TC3$ is compared with the previous test cases $TC1$ and $TC2$. It is not important in which order $TC3$ is compared with previous test cases, because the mathematical sum is associative. In the fourth iteration $TC4$ is compared with the three previous test cases $TC1, TC2$ and $TC3$.

The function dissimilarity: $\mathcal{TC} \times \mathcal{P}(\mathcal{TC}) \rightarrow [0,1]$ computes the average of pairwise dissimilarity between one test case TC_i and a set of test cases TC_{P_i} for a product variant as follows:

$$dissimilarity(tc_i, TC) := \frac{\sum_{n=0}^{|TC|} distance_j(tc_i, TC_n)}{|TC_p|} \tag{4.5}$$

$$distance_j(tc_i, tc_n) = 1 - J(tc_i, tc_n) \tag{4.6}$$

For example the test set $TS = \{TC1, TC2, TC3\}$ has to be evaluated. As said earlier the first test case $TC1$ does not need any evaluation because it would only be compared with itself. The dissimilarity for test case $TC2$ is computed only by comparing with $TC1$ divided by 1. For test case TC_3 the dissimilarity is the sum of the dissimilarities between $TC1 \leftrightarrow TC3$ and $TC2 \leftrightarrow TC3$ divided by 2.

5 Implementation

After the specification of the objectives for a GA this concept is implemented as an Eclipse plug-in. This chapter describes the implementation. At first the requirements of the tool are described. Afterwards, the concrete implementation and the used frameworks are described. Finally, the usage of the tool is described.

5.1. Requirements

At first, we describe the requirements of the tool that should be developed. We use UML Use Case diagrams to visualize these requirements. In Figure 5.1 the use cases for the main part of the wizard are shown.

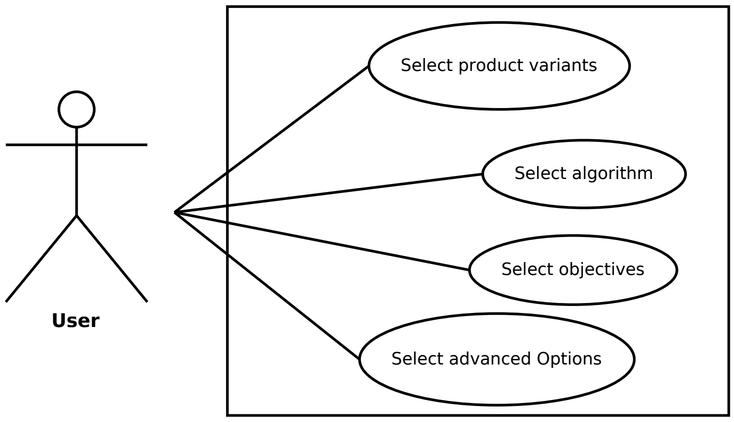


Figure 5.1.: Use Case for Wizard

It should be possible for the user to select for which product variants the test set should be prioritized. The user can select just a few product variants or all product variants. Likewise, the user shall be able select which algorithm is used for the optimization and which objectives described in the concept should be used. A second use case in Figure 5.2 shows the requirements of advanced options to modulate the GA.

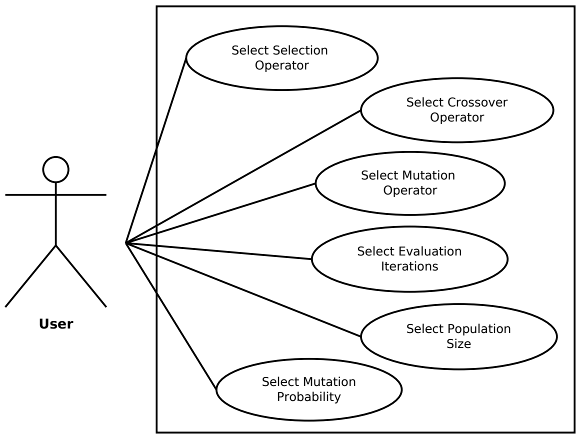

Select Selection Operator
Select Crossover Operator
Select Mutation Operator
Select Evaluation Iterations
Select Population Size
Select Mutation Probability
User

Figure 5.2.: Use Case for Advanced Options

The advanced options for this thesis can be used by users which want to refine the process of the GA. If the user is not advanced the default values should be used. These are values which are the only selectable options due to the implementation or values which guarantee good results. The default values are shown in Table 5.1.

Table 5.1.: Default Values of Advanced Options

Option	Default Value
Selection Operator	Tournament Selection
Crossover Operator	PMX Crossover
Mutation Operator	Swap Mutation
Max. Evaluations	5000
Population Size	100
Mutation Probability	0.10

The default operator for selection is selected because it is a widespread operator and the operators for the mutation and crossover is selected because these are the only supported operators in jMetal. For the max count of evaluations a high number is selected because a higher count of evaluations has a higher probability to find a good solution. The default value of the population size with 100 is selected because it is a good quantity for tournament selection and the crossover afterwards. The mutation probability value of 0.1 is selected because it will introduce a lot diversity to the population.

It should be possible to select the operators for the phases of a GA. This means the selection operator, crossover operator and mutation operator should be selectable. Likewise, the evaluation quantity, the population size and the mutation probability should be selectable.

Additionally, the tool should be extensible. Possible extensions are additional objectives for the fitness evaluation, other operators for the phases of the GA and more algorithms.

The intended workflow of the tool is shown in Figure 5.3.

At first the product variants for which the test set should be prioritized must be selected. Afterwards an algorithm and the objectives must be selected. In step four the user can make refinements for the process of the GA. This step is optional and default values are appointed for all advanced

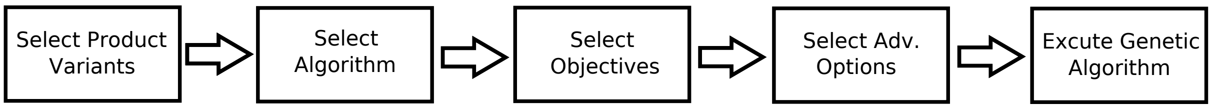


Figure 5.3.: Workflow for Prototype

options. Afterwards, the GA optimize the test sets for the product variant. These steps should be implemented by the tool. The next section will describe the implementation and the preconditions of this tool.

5.2. Realization and Implementation

The tool is realized as a plug-in for Eclipse-IDE. Eclipse supports the plug-in development with an own framework since Eclipse is a composition of multiple frameworks, too. Our prototype, the DoMoRe testing plug-in provides a wizard to start the process of a GA. The wizard providess the possibility to select which product variants and its test sets have to be prioritized, which algorithm should be used and which objectives are used to evaluate the individuals. Likewise, it is possible to select advanced options for GAs. The prototype is implemented using Xtext version 2.10.0 and Eclipse version Neon.

The concept of this thesis is implemented using the java-based framework *Metaheuristic Algorithms in Java* called jMetal [13]. It provides a rich set of classes which can be used as the building blocks of multi-objective techniques. The framework includes a number of classic and modern state-of-the-art optimizers, a wide set of benchmark problems and a set of well-known quality indicators to asses the quality of the algorithms. A set of base classes is provided to the user which can be used to design new algorithms, problems and operators for a GA. Additionally, jMetal provides multiple implementations of GAs, operators for selection, crossover and mutation. We use this framework to implement our objectives which are described in the concept in Chapter 4. Therefore, a new problem was specified in jMetal using the abstract class *problem* which is provided by the framework. For this thesis our individuals contains the retestable test cases for a product variant. Individuals in jMetal are only permutations of numbers which represent the position of the genes and are called solutions. This means if a product variant has three retestable test cases $\{TC_1, TC_2, TC_3\}$ the individual in jMetal is encoded to $\{0, 1, 2\}$ where 0 stands for the first position. With this encoded solutions the framework will proceed during the entire process. This means selection, crossover and mutation is applied on this encoded individuals. If a solution in jMetal is $\{2, 0, 1\}$ this is the encoded test set $\{TC_3, TC_1, TC_2\}$. The whole population, i.e., the entire set of individuals, are called solution set in the framework. It is possible to extend the framework with more problems, operators and algorithms. Therefore, jMetal provides a set of abstract classes which have to be extended with the new implementation.

Additionally, the implementation of the prototype use the Eclipse plug-in from Lachmann et al. [57]. This plug-in provides the architecture description language (ADL) DeltaRx to model delta-oriented architectures. With DeltaRx it is possible to model a core architecture and deltas based on a feature model. Likewise, they implemented the message sequence chart description language MSCDL to model test cases for integration testing. MSCDL is directly embedded in DeltaRx via grammar mixin. This means to validate a test case MSCDL use the architecture descriptions generated by Deltarx. These two description languages are used to compare connectors in test cases with connectors in deltas.

Figure 5.4 shows the class diagram for the SPL DoMoRe Testing Tool. For reasons of clarity the variables and operations of the classes are disregarded. More detailed class diagrams can be found in the appendix.

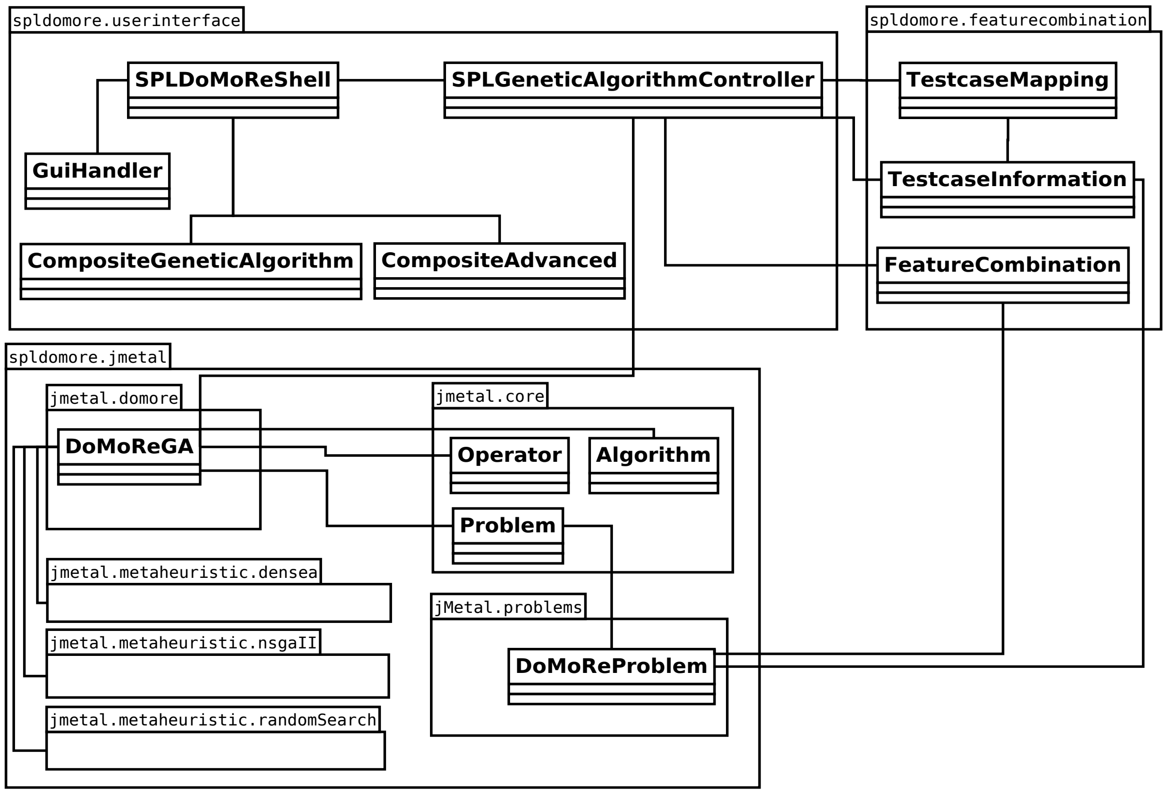


Figure 5.4.: Classdiagram of SPL DoMoRe Tool

In the following we describe each class of the class diagram in Figure 5.4 more detailed.

- **CompositeGeneticAlgorithm:** This class implements the graphical user interface (GUI) of the main tab. It describes which product variants are able to evaluate and which GAs can be used. Likewise, the list of objectives is defined.
- **CompositeAdvanced:** The *CompositeAdvanced* class implements the GUI for the advanced options. The possible selection operators, crossover operators and mutation operators are specified. Likewise, the maximum of evaluations, the population size and the mutation probability is described.
- **SPLDoMoReShell:** The class *SPLDoMoReShell* put the two tab implementations *CompositeGeneticAlgorithm* and *CompositeAdvanced* into one shell and specifies the size and the ordering of the tabs. Additionally, this class calls the method in class *SPLGeneticAlgorithmController* which load all relevant files.
- **SPLGeneticAlgorithmController:** This class made several previous computation before the evaluation with the GA is executed. It loads every information from files like all features given

in the feature model, features and test cases for every product variant. This was implemented this way to avoid loading multiple times the same data. Additionally, this class intercept some corner cases. The first one is that a GA in jMetal will not optimize one single objective. This means if a product variant just got one single test case to retest the process of the GA will not start and the user get a massage that a product variant has just one retestable test case. Likewise, it is not possible to select no objective. This will lead into an exception and the process is not started.

- **TestcaseMapping:** The class *TestcaseMapping* is called from *SPLGeneticAlgorithmController* before the process of the evaluation is started. It will map automatically each test case for the whole SPL to features according to the concept in Chapter 4.
- **TestcaseInformation:** This class provides informations about each test case. It stores in an object which features are mapped to a test case and how many connectors a test case covers for a delta.
- **FeatureCombination:** The class *FeatureCombination* provides a method to store a feature combination. Likewise, it provides a method to check if two feature combinations are equal this is needed because two feature combinations are equal although the features are not in the same order. Additionally, this class contains methods to make a string out of a feature combination and a method to check if a feature combination contains a feature.
- **DoMoReGA:** The *DoMoReGA* class is responsible to define every parameter the GA need to be executed. The class defines which algorithm should be used for the evaluation and which operators should be used for the GA. Likewise, it specifies which problem the GA should use to optimize the individuals, how much evaluations have to be performed, the population size and the mutation probability. After defining every parameter this class calls the method to run the GA.
- **DoMoReProblem:** This class is the main part of this implementation and contains all methods to evaluate individuals. The class implements the concept described in Chapter 4 and is executed from the GA in jMetal.
- **Operator, Algorithm, Problem:** The classes *Operator, Algorithm* and *Problem* are provided by jMetal and are not implemented by us. They provide all needed methods for the GA.

The next subsection will describe how the SPL DoMoRe Testing Tool can be used within Eclipse.

5.3. Workflow

The SPL delta-oriented multi-objective regression testing (DoMoRe) wizard provides an easy way to solve a delta-oriented prioritization problem. Before the prioritization of retestable test cases we can commence to execute the SPL regression testing tool from Lachmann et al. [57] which is also an Eclipse plug-in. The SPL regression testing tool performs the test case categorization based on their approach and will load every necessary file for the prioritization, e.g., a description of the feature model and the feature configurations of each product variant.

In the following, we will describe how all these plug-ins can be used. First, all projects for the SPL regression testing tool from Lachmann et al. [57] and all projects from the SPL DoMoRe testing tool must be imported into Eclipse. Afterwards, the user have to run all these projects in a new instance of Eclipse and execute following steps. The workflow for the SPL regression testing tool and the SPL DoMoRe testing tool is shown in Figure 5.5.

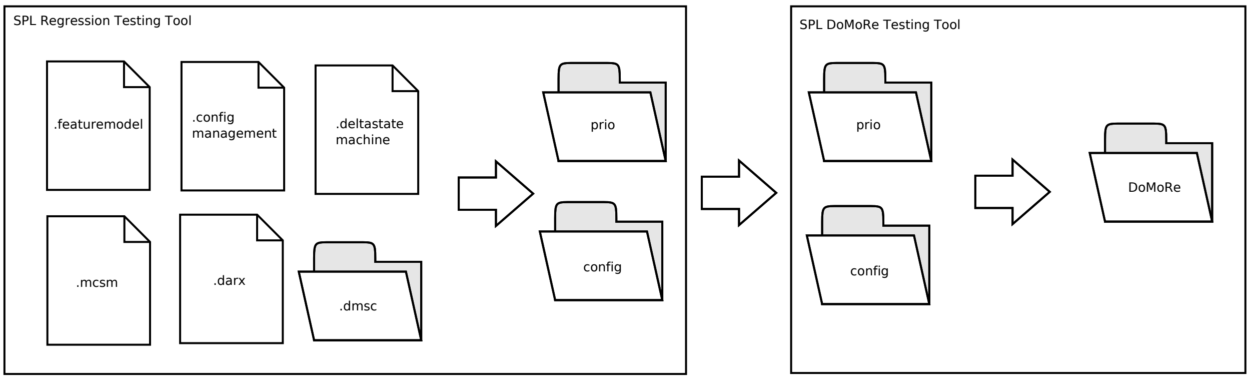


Figure 5.5.: Workflow of SPL Regression Testing Tool and SPL DoMoRe Testing Tool

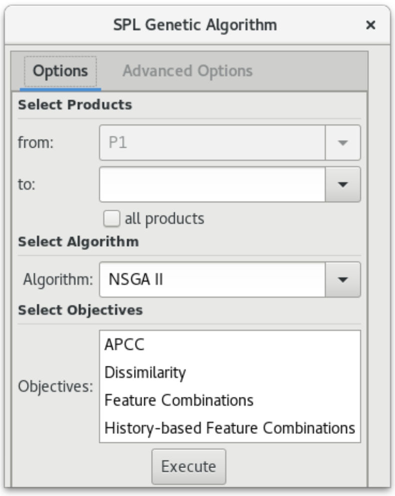
At first the workflow for the SPL regression tool from Lachmann et al. [57] is described in detail.

1. At first a new test case prioritization project via „new - SPL Regression Testing Wizard - SPL Regression Testing Project“ must be created
2. Within the wizard the user has to enter a name for the project and the paths of all files described in Figure 5.5. The tool needs a file .featuremodel which describes the feature model of the SPL in a XML scheme. Additionally, it needs a file .configmanagement which describes the feature configurations of the possible product variants. The file .deltastatemachine describes state machines for each delta. File .mcsm describes the mapping between components and features and .darx describes the core architecture and deltas.
3. It is not necessary to use HP Quality Center and, hence, the user has to check the „local“ check box and enter the path of the folder for the test cases described by .dmsc-files.
4. Once all fields are filled the user can click „Finish“. Now there is a new project in the workspace which contains each file that was entered and some other files, e.g., descriptions of regression deltas between product variants, which are needed for the process.
5. To prioritize test cases the user has to perform a right click on the project and select „SPL Regression Testing“. Afterwards, a new window will appear where the weights for the approach from Lachmann et el. [57] can be modified.
6. In the upper left screen the user has to select the product variants for which the test cases should be prioritized. We suggest to select all product variants to prevent further faults. The product variants are prioritized in the order in which they are described in the .configmanagement file. The prototype of this thesis needs .prio files which are generated for each selected product variant. If a .prio file for a product variant was not generated it is not possible to prioritize the test set for this product variant.
7. After selecting the product variants which shall be tested three different prioritization techniques are available introduced by Lachmann et al. [57]. For our approach the user should

use the „regression delta concept“ technique shown in the upper right corner. Our approach needs the categorization of test cases which is generated by this technique. It is all right to select no wighting factors and just click on „Create Testsuite“.

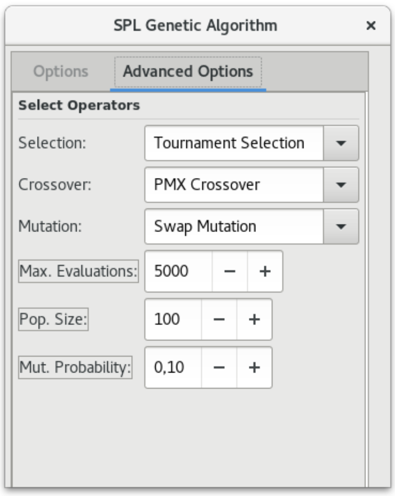
8. Now, the project contains some new files, e.g., the .prio-files for each product variant. The important files for our approach can be found in folder „prio“. For every product variant which was selected for prioritization a .prio-file must be generated.

These steps described the execution of the SPL regression tool from Lachmann et al. [57]. Now the execution of the SPL DoMoRe testing tool is described which can be seen in Figure 5.5.



from: P1
to:
Algorithm: NSGA II
Objectives: APCC, Dissimilarity, Feature Combinations, History-based Feature Combinations

(a) Graphical User Interface of the DoMoRe Wizard



Selection: Tournament Selection
Crossover: PMX Crossover
Mutation: Swap Mutation
Max. Evaluations: 5000
Pop. Size: 100
Mut. Probability: 0,10

(b) Advanced Options in GUI of the DoMoRe Wizard

With another right click on the project the user can select „SPL DoMoRe“ to open the wizard for the tool. In Figure 5.6a we can see the main part of the GUI. Afterwards, the user must select the product variants for which the prioritization should be performed, the used GA,e.g., *NSGAII* and the objectives.

In Figure 5.6b the advanced options tab of the GUI is shown. In this tab the user can select advanced options,e.g., selection operator, population size. The values shown as a preview in the advanced option tab are the default values for a GA. The user can select which operators should be used for the GA. Additionally, the user can select the maximum quantity of evaluation iterations. This is the stop criteria for the GA. The population size and the mutation probability can be selected as well. After selecting all wanted advanced options the user must return to the main tab and click on „Execute“.

Afterwards a click on „Execute“ will start the process of the GA. The GA will use the files from the folders „prio“ and „config“ to prioritize the test sets. In the project a new folder „DoMoRe“ and new files will be generated which show which options were selected and which optimized test sets were evaluated as the best solutions for the problem. Likewise, the values of the selected objectives are shown below the test sets. An example of the contents of this file is shown in Listing 5.1.

Listing 5.1.: Example File of Prioritized Test Sets

```
Product: P2, Algorithm: NSGA II
```

```
2 Evaluations: 3000, Population Size: 100, Execution Time: 2912 ms
3 Mutation: Swap Mutation, Selection: Tournament Selection, Crossover: PMX Crossover
4 APCC Feature Combinations
5 P1MSC19 P1MSC21 P1MSC11 P1MSC35 P1MSC7 P1MSC16 P1MSC25 P1MSC26 P1MSC10 P1MSC5 P1MSC28
6 -0.734632683658171 -0.9730392156862745
7 P1MSC19 P1MSC11 P1MSC35 P1MSC7 P1MSC16 P1MSC21 P1MSC25 P1MSC26 P1MSC10 P1MSC5 P1MSC3
8 -0.7376311844077962 -0.9728831851042805
9
```

The first rows describes the options for the prioritized test sets. It is shown for which product variant this test set was prioritized, which algorithm was used, how much evaluation and so on. Afterwards, the prioritized test set is shown in Listing 5.1 the first prioritized test set is given in line 5. Below the test set are the computed objective values for this test set. In the following we describe the selectable algorithms of the prototype.

5.3.1. Selectable Algorithms

In the DoMoRe testing tool it is possible to select three different algorithm. These three algorithms were selected because the framework jMetal provided them. The first algorithm is the *Random Search* algorithm which generate a random population for the specified problem. This algorithm is not really a genetic algorithm but can be used as a base line for evaluating other algorithms. The steps selection, crossover and mutation are not executed by this algorithm and in each generation a random population is generated.

The second algorithm is the *Non-dominated Sorting Genetic Algorithm* (NSGAII) which was introduced by Deb et al. [11]. The NSGAII algorithm is a well tested and efficient algorithm [35]. NSGAII uses elitism to increase the performance in the evaluation of the solution. Elitism means that non-dominated solutions are stored and survive to the next generation [35]. If a solution of a new population is better than a stored solution the stored solution will be replaced by the dominating solution. Elitism leads to a better performance since a new population is only compared with the previous population and the solutions in the non-dominated set of solutions.

The third algorithm is the *Duplicate Elimination Non- dominated Sorting Evolutionary Algorithm* (DENSEA) introduced by Greiner et al. [20]. The DENSEA algorithm emphasizes the creation and maintenance of population diversity. Additionally, DENSEA was developed to cope with the problem that the number of pareto optimal solutions is smaller than the population size.

5.3.2. Selectable Operators

Optimizing a problem does not only need a good metric to evaluate the solution for a problem but also good operators to support the optimization. In the following we will discuss which described operators in Chapter 2.3 are suitable for the test set prioritization problem and which operators are implemented in the prototype tool.

Selection: Possible selection operators are explained in Subsection 2.3. All these operators are applicable for the problem in this thesis because they are independent of the type of individuals. Likewise, selection operators will not manipulate a individual and the genes. However, we use jMetal as a framework for GA. Not all described selection operators are implemented. Therefore, only the tournament, best solution selection and random selection are selectable for the GA.

Crossover: The crossover operators described in Subsection2.3 were developed for the TSP. This problem considers that genes in the individual only occurs once. Therefore, every operator described in the chapter is a good way to generate offspring. Similar to the selection operators not all described crossover operators are implemented in jMetal. Therefore, only „PMX Crossover“ is selectable as crossover operator for the GA. In PMX Crossover randomly two cut points are selected and the genes between these cut points are copied to the offspring. Afterwards, the other genes are filled with genes from the other parent. This crossover operator consider that no gene occurs multiple times in the individual.

Mutation: For the mutation of a test set not every mutation operator described in Subsection 2.3 is possible. Reciprocal, inversion, swap and displacement mutation are applicable for the problem of this thesis. These operators did not need any special representation of a gene or manipulate the value of a gene. The genes in this thesis are test cases and they did not got a special value. Therefore, boundary and uniform random mutation are not applicable for the test set optimization problem. The *Flip Bit Mutation* is only applicable on a binary representation of genes. This mutation operator is not applicable for this thesis, because our genes have no binary representation.s Likewise, the insertion mutation is not applicable. This operator selects a random gene and replace it with another gene. In this thesis, all retestable test cases are in the individual and there are no other test cases to replace one test case. Additionally, the test cases were previously selected and we do not want to replace any of them. Similar to the other two operators, only *Swap Mutation* is implemented in jMetal.

However, it is possible to extend jMetal with the mentioned but not implemented operators.

6 Evaluation

This chapter will discuss the evaluation of the introduced concept of this thesis. At first, we define two research questions which should be answered in the evaluation and describe the methodology of the evaluation. Afterwards, we discuss the results of the evaluation with regards to the two research questions. Finally, we will discuss threats to the validity of this evaluation.

6.1. Research Questions

For the evaluation of the concept we focus on the following research questions:

- **RQ 1:** *Did effective test cases got a higher prioritization?* This thesis aims to reduce the testing effort for SPL testing. For this purpose, test cases are prioritized to detect faults as early as possible. If then the testing process is terminated, e.g., due to resource constraints, the most important test cases up to this point have already been executed [18]. Therefore, we have to evaluate if the specified objectives support this goal and effective test cases, i.e. test cases which are more likely to find faults, are prioritized higher than less effective test cases.

- **RQ 2:** *How efficient are the used genetic algorithms?* We have to evaluate if the introduced approach is an efficient replacement to other techniques, e.g., manual ordering or random test sets. Therefore, we have to evaluate how efficient the used GAs prioritize test sets. This means the prioritization of test sets a reduction of the testing effort with regards to the execution time.

The next section will introduce the case study which was used to evaluate the concept.

6.2. Case Study

The case study used in this thesis is a *body comfort system* (BCS) SPL for a vehicle [60]. The BCS case study was originally developed by Müller et al. [50] in cooperation with industrial partners from the automotive sector and has the following functionalities:

- *Human Machine Interface* as control unit and with *Status LEDs*
- Electric *Power Window* with a *Finger Protection*
- Electric *Exterior Mirror* which is *heatable*
- *Central Locking System* with *Automatic Locking* for doors
- *Remote Key Control* with *Safety Function*, *Alarm System* and *Power Window Control*

Oster et al. [54] enhanced the BCS to a SPL by making some of the functionalities in the original case study into variable parts. A feature model shown in Figure 6.1 was defined to specify the variability of the BCS case study. The feature model comprises 27 features and each feature represents a system functionality. The feature model has five *requires* constraints, e.g. the selection of feature *LED Alarm System* requires the selection of feature *Alarm System* and one *exclude* constraint, e.g. features *Manual Power Window* and *Control Automatic Power Window* can not be selected for the same product variant. Additionally, the SPL has two alternatives of the *Power Window*. *Manual Power Window* moves up or down when the button for the window movement is pressed and hold and the *Automatic Power Window* moves up or down when the button is pressed once. With all 27 possible features the BCS SPL comprises 11, 616 valid product variants. Zink [72] has evaluated the BCS SPL study for combinatorial pairwise testing and has identified a representative subset of 17 product variants. Afterwards, Lity et al. [42] added a new valid product variant P0 which comprises the most commonalities among the various product variants. Product variant P0 is used as the core product for the delta-oriented approach. Additionally, Lity et al. [60] added architecture models, deltas and state machines to the product variants.

To evaluate our approach we use the deltas, feature configurations and the 92 test cases of this case study.

6.3. Methodology

To investigate the research questions from Chapter 6.1, the concept of this thesis was evaluated by means of the BCS SPL study. Lachmann et al. [57] introduce a test case prioritization technique for the BCS SPL study. This technique includes the categorization of the test cases and the computation of regression deltas to identify the changes between product variants. The categorization of test cases for each product variant is used to evaluate our concept.

In this thesis we introduced four different objectives in Chapter 4 where the objective for feature combinations and for history-based feature combinations are an alternative to each other. Since this thesis focuses multi-objective prioritization of test sets we want to evaluate how good the different objective combinations are. Possible objective combinations for this thesis are: For clarity the objective names are abbreviated. *Maximization of Changes Covered* is abbreviated to *APCC*, *Maximization of Dissimilarity* to *Diss*, *Maximization of Feature Combinations* to *FC* and *Maximization of History-based Feature Combinations* to *HFC*.

Table 6.1.: Objective Combinations

Objective 1	Objective 2	Objective 3
APCC	Diss	
APCC	FC	
APCC	HFC	
APCC	Diss	FC
APCC	Diss	HFC
Diss	FC	
Diss	HFC	

For each objective combination we get the pareto optimal front of the prioritized test sets. This means we get the best test sets for the objective combination. However, we can not compare the objective combinations because we get the prioritized test sets and the values for each objective.

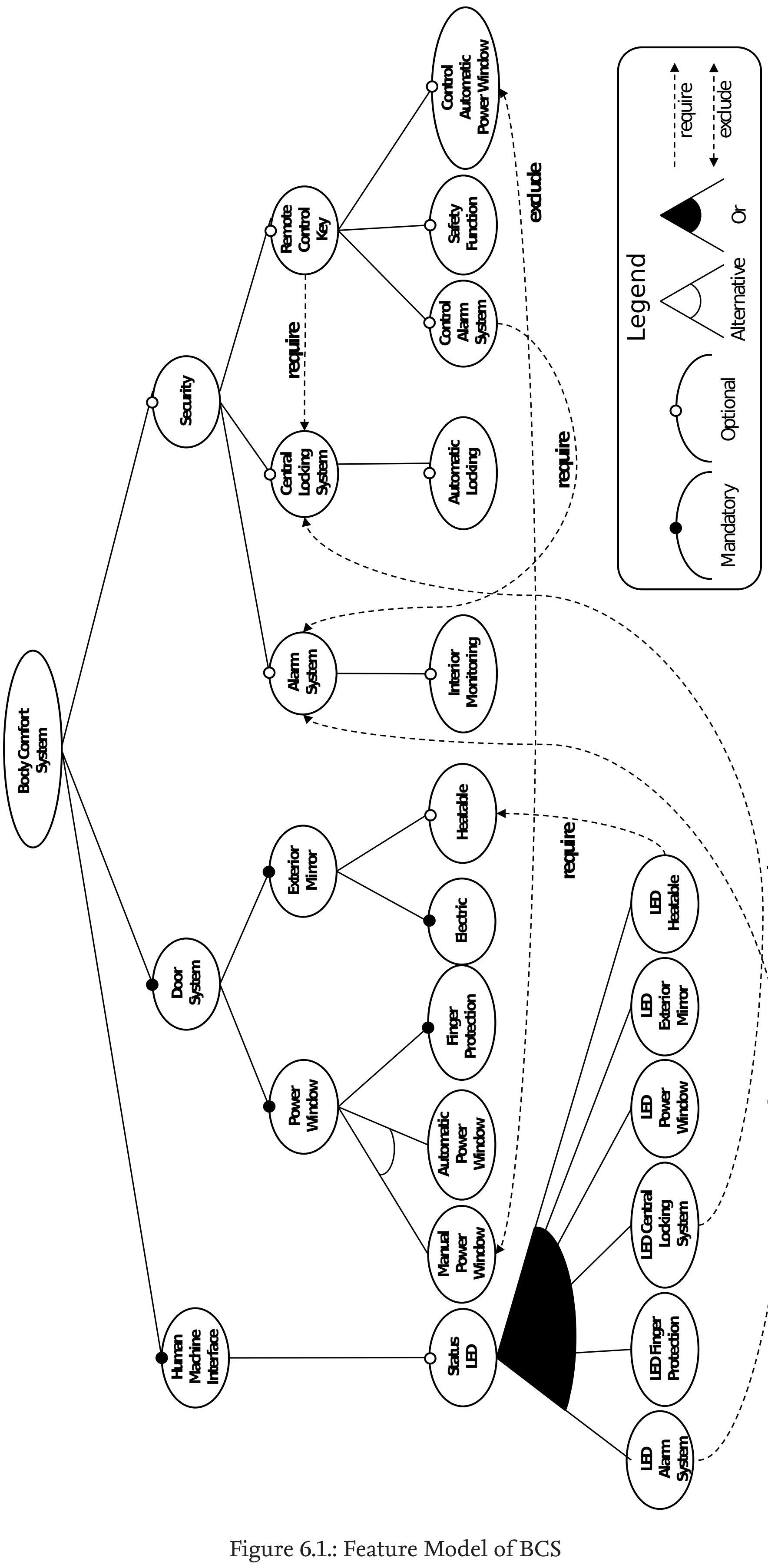


Figure 6.1.: Feature Model of BCS

Therefore, we measure the *Average Percentage of Failures Detected* (APFD) of the prioritized test sets. The APFD was introduced by Rothermel et al. [18] to measure the quality of a prioritized test set. It ranks test sets higher which have an early fault detection rate. This means it measures on which position of the test set a failure is covered first. The metric returns a value between 0 and 1 where 1 is the best result. The usage of this metric allows the comparison between the different objective combination in this thesis.

The BCS SPL study does not provide failures for the product variants. To use the APFD to measure the quality of test sets we use the assumption that failures are more likely to be found in changed parts of the architecture. Therefore, we have introduced exactly one random failure to changed parts of the architecture of each product variant. The changed parts are defined as connectors which are connected to components with changed interfaces after applying a delta [57]. The interface of a component is the set of incoming and outgoing connectors of a component. Changed parts are only considered if the current interface was not tested before. From this set of changed connectors we randomly chose one connector which theoretically generates a failure. A failure is covered if a test case covers it. The APFD measures at which position in the test set the introduced failure is covered by a test case. For each run of the evaluation a new failure to a product variant is introduced.

The evaluation of our concept was executed using three different GAs which where introduced in Chapter 5.3.1. As base-line we use the *Random Search* algorithm and compare it with the two algorithm *NSGAII* [11] and *DENSEA* [20] which are supported by jMetal. Additionally, we compared the results of the *NSGAII* and the *DENSEA* with the prioritization approaches from Lachmann et a. [57]. They introduced a component-based and signal-based prioritization. To cope with statistically outliers we prioritized test sets for a product variant and objective combination in 50 runs. Since it is possible that a pareto front of prioritized test sets contains more than one test set we measured the average APFD value for each run. In conclusion we combine the different objectives and use jMetal to get optimal test sets for the BCS case study. For every objective combination we generate 50 files with pareto optimal solutions for every product variant and for a single algorithm. This will lead to $16\ products \cdot 50\ runs \cdot 11\ objective\ combinations = 8800$ files per algorithm. We prioritized only 16 product variants because product variant P0 is the core product and has to be fully tested. These optimal test sets are evaluated measuring the APFD that is stored for each pareto optimal solution.

We executed the evaluation on an AMD Phenom II X6 1090 T processor with 3.21 GHz and 8 GB RAM.

6.4. Results

This section discusses the results of the two research questions defined in Section 6.1.

6.4.1. Results of Research Question 1

RQ 1: Did effective test cases got a higher prioritization?

This thesis aims to reduce the testing effort for SPL testing. Test case prioritization is used to order a test set in a way that effective test cases are tested early. Therefore, we have to discuss if the approach from this thesis prioritize effective test cases.

For the evaluation of the concept we measured the average APFD value for 50 runs to cope with statistical outliers. Additionally, the concept is evaluated using two different GAs *NSGA II* and

DENSEA. To evaluate how effective the introduced objective combinations from this thesis are we compare them with three other approaches. The first approach is a random algorithm, the second the component-based prioritization and the third the signal-based prioritization both introduced by Lachmann et al. [57].

The first objective combination which is compared with the three other approaches is *APCC-Diss-FC*. For the final APFD value for the evaluation the average APFD per run is calculated and stored to a file. Afterwards, the average of these 50 values is taken.

The resulting APFD values for product variants P2-P17 are shown in Figure 6.2. Product variant P1 is not shown because this product variant has only one retestable test case and a GA is not capable to optimize an individual with only one gene. Therefore, we have no values for product variant P1.

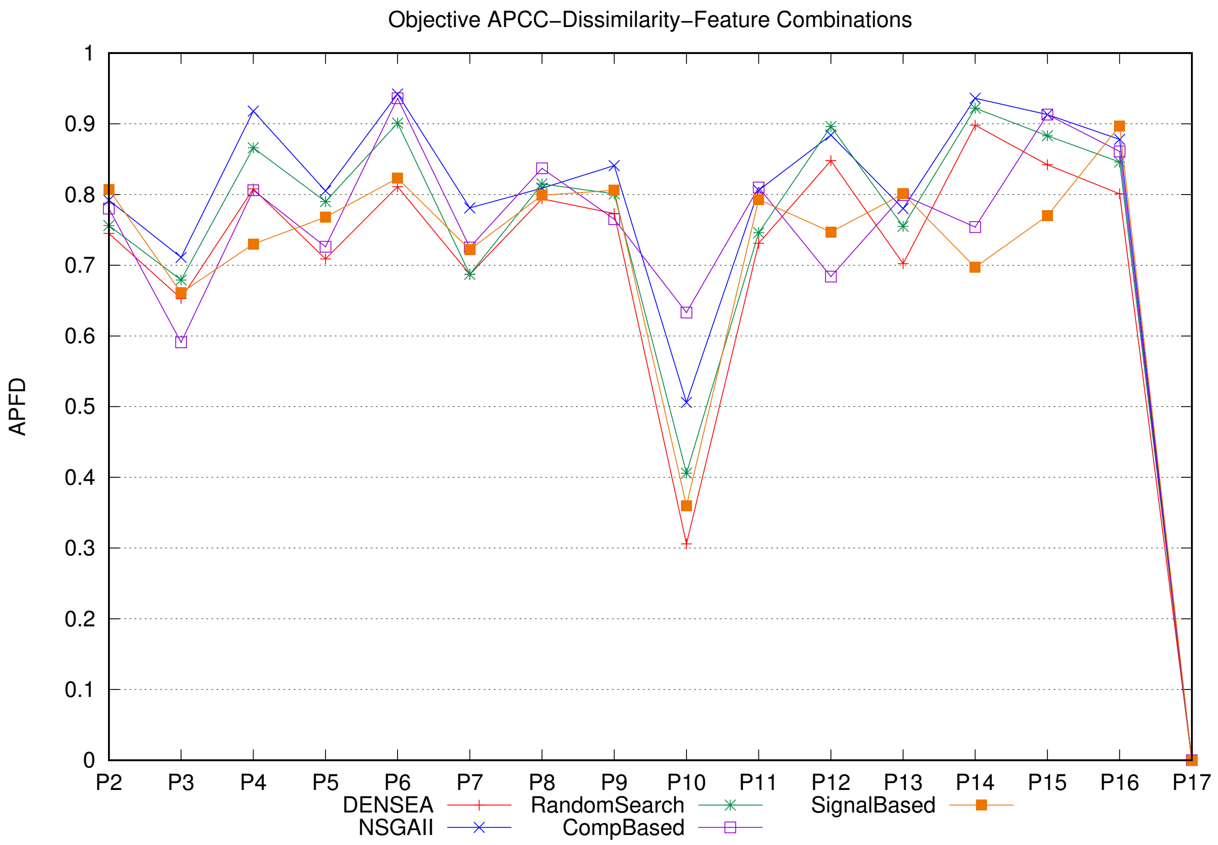


Figure 6.2.: Comparison for Objective Combination APCC, Diss and FC with different Approaches

First, the results in Figure 6.2 indicate that for each approach the APFD value of product variant P17 is 0. Product variant P17 does neither contain any new signals nor comprise untested component interface configurations. Therefore, no failure is introduced for product variant P17 and an APFD value of 0 is expected. Additionally, for product variant P10 all approaches have a drop in the APFD value. If we compare the changes in the architecture of P10 with other architectures of product variants the relation between components in the architecture and changed components is the highest for P10. Product variant P10 has nine components and six have changed interfaces. For other product variants this relation between components in the architecture and changed components is lower. This explains the drop of the APFD value for product variant P10.

After the previous general observations, we will discuss the results for the *NSGA II* algorithm and the *DENSEA* algorithm. Overall the *NSGA II* algorithm prioritizes the test sets for 9 of 16 product variants for the objective combination *APCC-Diss-FC* better with regards to the APFD than the other approaches. The results indicates that the APFD values for *NSGA II* algorithm are always better than APFD values from the *DENSEA* algorithm and the random approach. The *DENSEA* algorithm has only for product variants P12 and P14 better APFD values than the component-based and signal-based approaches. For all other product variants the values from the APFD for *DENSEA* are worse or equal than the values of component-based or signal-based.

In the following we will discuss the results for the second objective combination. The resulting APFD values for objective combination *APCC-Diss-HFC* are shown in Figure 6.3.

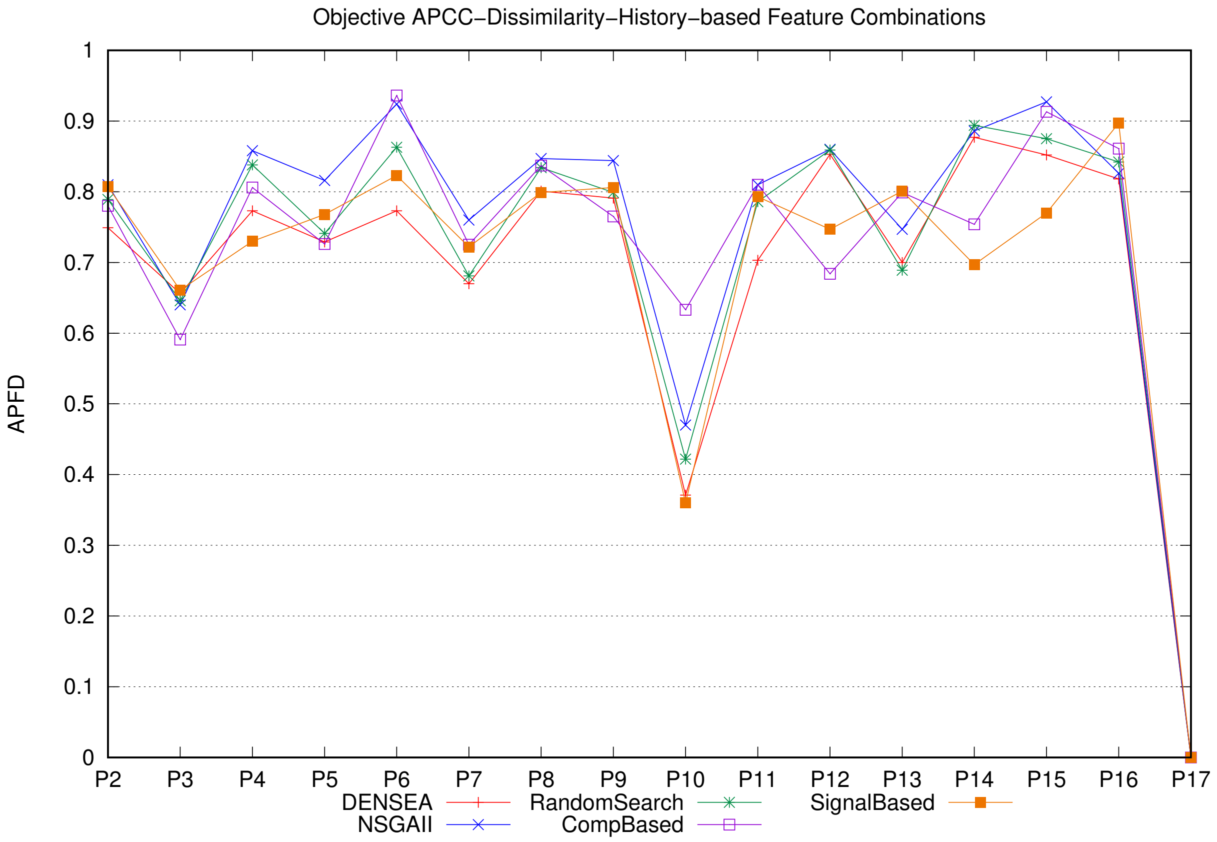


Figure 6.3.: Comparison for Objective Combination APCC, Diss and HFC with different Approaches

The results for objective combination *APCC-Diss-HFC* in Figure 6.3 indicate the same observations for P17 and P10 as for objective combination *APCC-Diss-FC*. Compared to objective combination *APCC-Diss-FC* the results show that the *NSGA II* algorithm has the best APFD values for 9 out of 16 product variants as well. Overall the APFD values for *NSGA II* are smaller and also the differences between the APFD values of the other approaches. Similar to objective combination *APCC-Diss-FC* the *DENSEA* algorithm has worse APFD values and the APFD value for the component-based approach from Lachmann et al. [57] is in 10 of 16 product variants better.

Comparing the two objective combinations *APCC-Diss-HFC* and *APCC-Diss-FC* the results indicate that the combination with FC is better with regards to the APFD value. Additionally, the evaluation has shown that both objective combinations with the *NSGA II* algorithm has mostly better APFD values than a random, a component-based and a signal-based approach, hence, the test set has a better prioritization.

The previous results were related to objective combinations with three objectives. As the following results will show, these are not the ideal objective combinations. To evaluate the best objective combination for this thesis we measured the average values of the APFD for every objective combination and product variant. The table for the *NSGA II* algorithm can be found in Table 6.2 and for the *DENSEA* algorithm in Table 6.3.

Table 6.2.: Comparison of Average Values of the APFD for NSGAII and Objectives

Objectives	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17
APCC-Diss-FC	.792	.711	.918	.805	.943	.781	.809	.841	.506	.806	.884	.780	.939	.912	.877	**.000**
APCC-Diss-HFC	.810	.640	.858	.816	.925	.760	.847	.844	.471	.810	.860	.747	.886	.927	.826	**.000**
APCC-Diss	.858	**.783**	.921	.860	.931	.795	.835	.875	.512	.874	.942	**.854**	.916	.918	.907	**.000**
APCC-FC	.811	.713	.954	.857	.970	.739	.839	.885	.468	.811	.907	.818	.982	.952	.922	**.000**
APCC-HFC	.833	.751	.901	.862	.954	.766	.844	.881	.444	.848	.912	.788	.933	.954	.864	**.000**
APCC	**.890**	.746	**.963**	**.905**	**.981**	**.840**	**.911**	**.931**	**.533**	**.931**	**.976**	.834	**.983**	**.966**	**.946**	.000
Diss-FC	.742	.621	.889	.767	.896	.673	.687	.768	.385	.755	.796	.738	.716	.863	.831	**.000**
Diss-HFC	.696	.705	.806	.738	.885	.759	.754	.799	.523	.771	.731	.716	.728	.905	.726	**.000**
Diss	.758	.713	.772	.729	.838	.609	.736	.795	.491	.783	.771	.769	.749	.796	.832	**.000**
FC	.597	.593	.921	.712	.924	.493	.648	.824	.252	.707	.747	.633	.823	.818	.804	**.000**
HFC	.642	.503	.682	.720	.864	.665	.804	.772	.227	.662	.738	.545	.825	.925	.563	**.000**

Table 6.3.: Comparison of Average Values of the APFD for DENSEA and Objectives

Objectives	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17
APCC-Diss-FC	.745	.653	.808	.709	.811	.687	.794	.773	.306	.731	.848	.702	.898	.842	.801	**.000**
APCC-Diss-HFC	.749	.656	.773	.729	.773	.670	.801	.791	.371	.703	.853	.700	.877	.852	.818	**.000**
APCC-Diss	**.816**	.691	.838	.679	.814	.711	.812	**.832**	.466	.815	.884	.702	.879	.842	.845	**.000**
APCC-FC	.769	.619	**.877**	.777	.878	.675	.793	.795	.319	.741	.888	.696	.960	.878	.848	**.000**
APCC-HFC	.756	.636	.830	.726	.856	.708	.822	.822	.426	.778	.868	.699	.903	.898	.844	**.000**
APCC	.809	**.725**	.819	**.828**	**.926**	.746	**.872**	.828	**.509**	.794	**.965**	.693	**.963**	**.947**	**.901**	**.000**
Diss-FC	.782	.599	.796	.768	.799	.670	.764	.746	.365	.697	.810	.629	.826	.812	.796	**.000**
Diss-HFC	.718	.537	.800	.674	.788	.662	.745	.738	.334	.769	.805	.707	.816	.826	.789	**.000**
Diss	.758	.641	.703	.659	.742	**.771**	.787	.796	.419	**.819**	.806	**.768**	.863	.738	.797	**.000**
FC	.735	.536	.834	.680	.780	.607	.757	.727	.307	.704	.811	.585	.809	.827	.806	**.000**
HFC	.724	.615	.781	.662	.706	.609	.799	.746	.316	.709	.744	.616	.749	.833	.703	**.000**

The best result for each product variant is marked with bold numbers. The results in the tables indicates that the most highest results for every product variant can be achieved with the objective *APCC*. As shown in Table 6.2 the *NSGA II* algorithm has for 12 out of 16 product variants the best average APFD value for objective *APCC*.

The results for *DENSEA* shown in Table 6.3 indicates that for 11 out of 16 product variants the best average APFD value is achieved for objective *APCC*. Since we introduce one failure to the BCS study into changed parts of the architecture this result was expected. The *APCC* prioritizes test cases which covers changes in the architecture and, hence, this objective will cover the introduced failure fast. While this thesis focuses multi-objective prioritization and the result was expected we additionally computed the differences between the best average APFD result and the average APFD results for the other objective combinations. The results for the APFD differences for the *NSGA II* algorithm are shown in Table 6.4 and for the *DENSEA* algorithm in Table 6.5.

Table 6.4.: Comparison of Average Differences for NSGAII and Objectives

Objectives	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17
APCC-Diss-FC	.098	.072	.045	.100	.039	.060	.102	.090	.028	.125	.092	.074	.044	.053	.069	**.000**
APCC-Diss-HFC	.079	.143	.104	.089	.056	.081	.064	.087	.063	.121	.116	.107	.097	.039	.120	**.000**
APCC-Diss	.032	**.000**	.042	.046	.050	.045	.076	.056	.022	.057	.035	**.000**	.067	.047	.039	**.000**
APCC-FC	.079	.070	.009	.049	.011	.101	.072	.046	.066	.120	.070	.036	.001	.013	.024	**.000**
APCC-HFC	.056	.032	.062	.043	.027	.074	.068	.050	.090	.083	.064	.067	.050	.012	.082	**.000**
APCC	**.000**	.037	**.000**	**.000**	**.000**	**.000**	**.000**	**.000**	**.000**	**.000**	**.000**	.021	**.000**	**.000**	**.000**	**.000**
Diss-FC	.148	.162	.074	.138	.086	.168	.224	.163	.149	.176	.180	.116	.267	.103	.115	**.000**
Diss-HFC	.194	.078	.157	.167	.097	.082	.158	.133	.010	.161	.245	.139	.255	.061	.220	**.000**
Diss	.132	.070	.191	.176	.143	.231	.175	.136	.042	.148	.206	.085	.234	.169	.115	**.000**
FC	.293	.190	.042	.193	.057	.348	.264	.107	.281	.224	.229	.221	.160	.148	.142	**.000**
HFC	.248	.280	.281	.185	.117	.175	.108	.159	.307	.269	.238	.309	.158	.041	.383	**.000**

Table 6.5.: Comparison of Average Differences for NSGAII and Objectives

Objectives	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17
APCC-Diss-FC	.072	.072	.069	.119	.115	.084	.078	.059	.203	.089	.116	.067	.065	.105	.100	**.000**
APCC-Diss-HFC	.067	.069	.104	.099	.153	.100	.070	.041	.138	.116	.111	.068	.085	.094	.083	**.000**
APCC-Diss	**.000**	.034	.039	.149	.113	.060	.060	**.000**	.043	.004	.081	.066	.084	.105	.056	**.000**
APCC-FC	.048	.106	**.000**	.052	.048	.096	.079	.037	.190	.078	.076	.072	.003	.068	.054	**.000**
APCC-HFC	.060	.089	.047	.102	.070	.063	.050	.010	.083	.041	.096	.069	.060	.048	.057	**.000**
APCC	.008	**.000**	.058	**.000**	**.000**	.024	**.000**	.004	**.000**	.025	**.000**	.076	**.000**	**.000**	**.000**	**.000**
Diss-FC	.034	.126	.081	.061	.127	.101	.108	.086	.144	.122	.155	.139	.137	.135	.105	**.000**
Diss-HFC	.099	.188	.077	.154	.138	.109	.126	.094	.176	.050	.160	.061	.147	.121	.112	**.000**
Diss	.058	.084	.174	.169	.184	**.000**	.085	.036	.091	**.000**	.159	**.000**	.100	.209	.104	**.000**
FC	.081	.189	.044	.148	.146	.164	.115	.105	.202	.116	.153	.183	.154	.120	.095	**.000**
HFC	.092	.111	.096	.166	.220	.162	.073	.086	.193	.110	.220	.152	.214	.113	.198	**.000**

The marked bold numbers are the best results of the average APFD values and got the value 0. The other values are the differences between the best average value and the average APFD values for the objective combination. Comparing these differences we observe that the objective combinations *APCC-FC* and *APCC-HFC* have the smallest differences to the best average APFD value. Therefore, we will evaluate our approach with these two objective combinations in the following.

The resulting average APFD values for product variants P2-P17 for objective combination *APCC-FC* is shown in Figure 6.4.

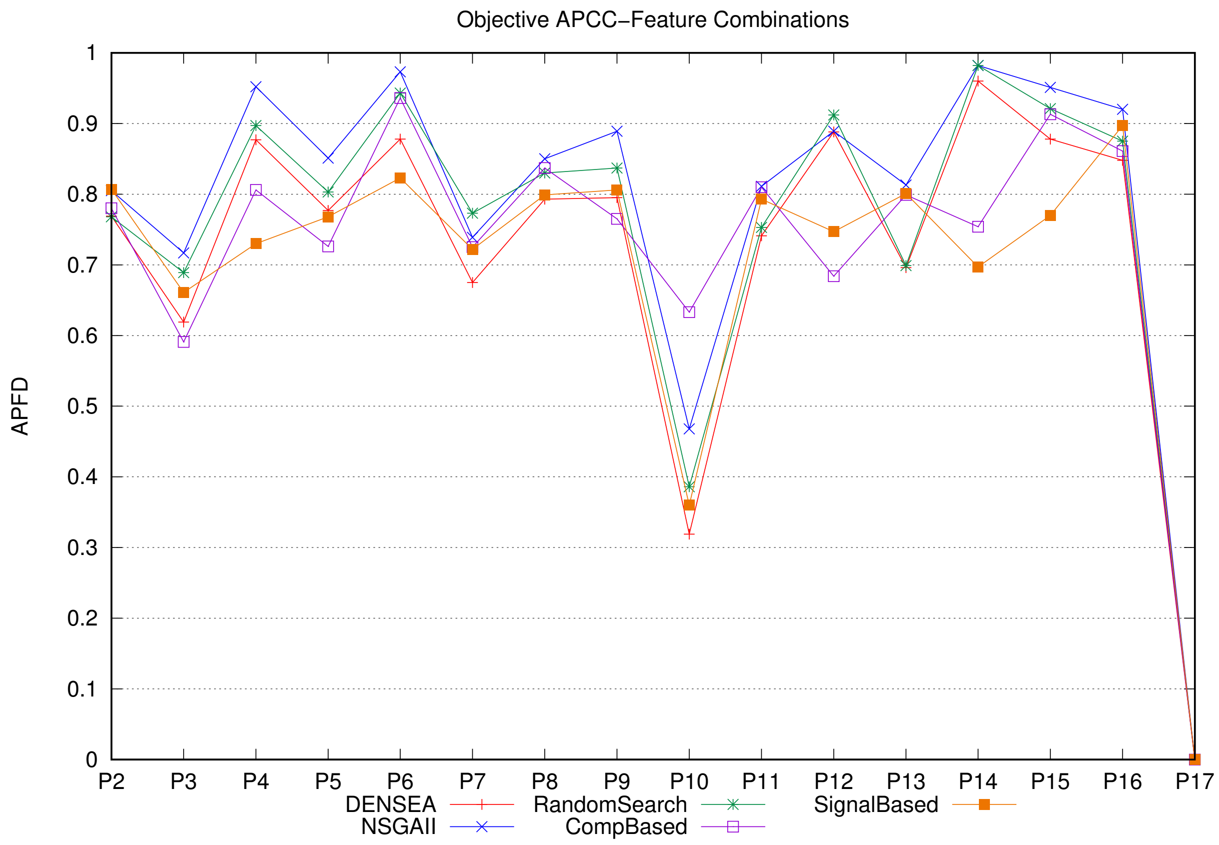


Figure 6.4.: Comparison for Objective Combination APCC and FC with different Approaches

For objective combination *APCC-FC* the results indicates that the *NSGA II* algorithm has the best APFD value for almost every product. Comparing the values for *NSGA II* for objective combination *APCC-FC* and *APCC-Diss-FC* the results indicates that the combination shown in Figure 6.4 got higher APFD values. Escpecially, for product variant P8 we can observe that for the objective combination *APCC-FC* the APFD value is better than for the component-based approach. For objective combination *APCC-Diss-FC* the APFD value for the *NSGA II* algorithm is worse than for the component-based approach. Additionally, the results of the APFD value for objective combination *APCC-FC* for algorithm *DENSEA* are better than for objective combination *APCC-Diss-FC*.

In conclusion, the objective combination *APCC-FC* is better than objective combination *APCC-Diss-FC*.

After discussing the results for objective combination *APCC-FC* we will discuss the results for objective combination *APCC-HFC* in the following. The resulting average APFD values for product variants P2-P17 for objective combination *APCC-HFC* is shown in Figure 6.5.

For objective combination *APCC-HFC* the results indicates also that the approach with algorithm *NSGA II* almost always has the best APFD values. Likewise, the *DENSEA* algorithm has worse APFD values.

After discussing the results for objective combinations *APCC-FC* and *APCC-HFC* in isolation we will discuss which objective combination is better.

The average APFD values of the two objective combinations *APCC-HFC* and *APCC-FC* are shown

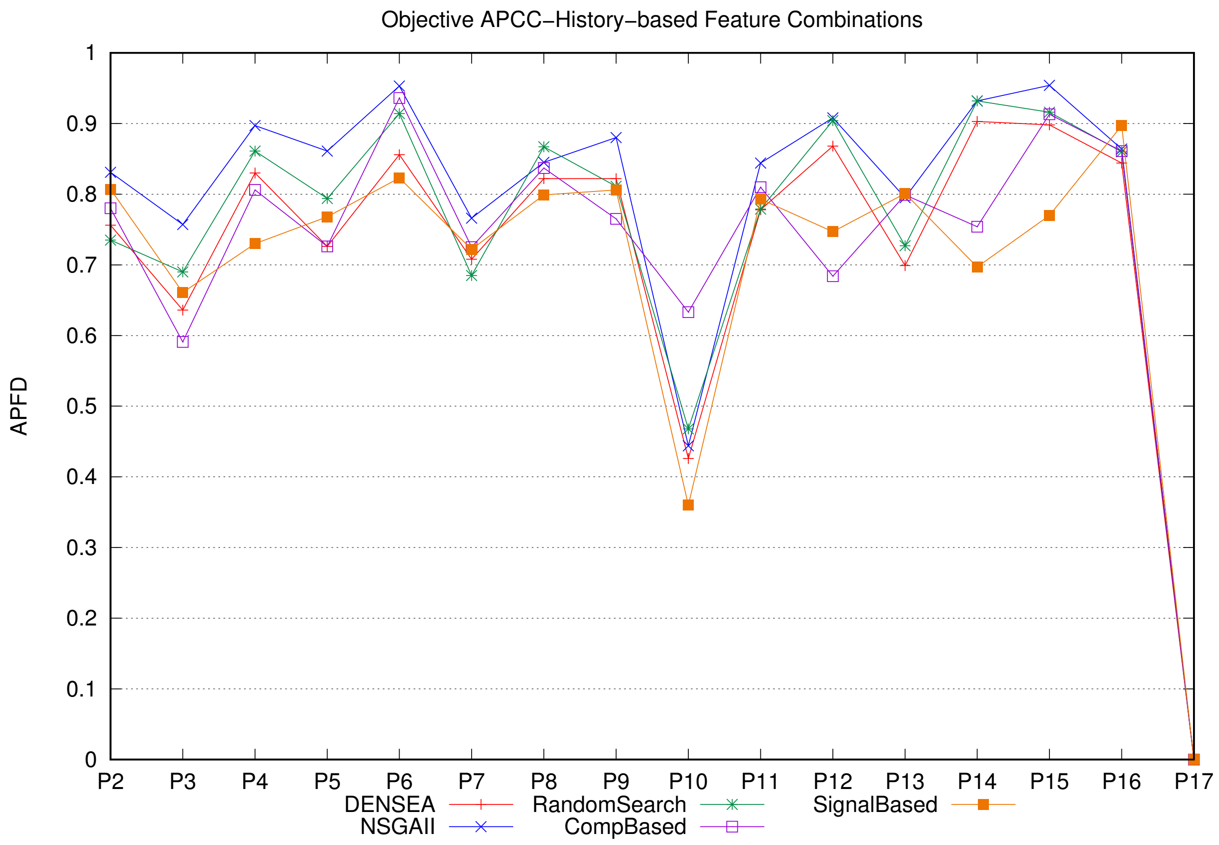


Figure 6.5.: Comparison for Objective Combination APCC and HFC with different Approaches

in Table 6.6 for the *NSGA II* algorithm and in Table 6.7 for the *DENSEA* algorithm.

Table 6.6.: Comparison of average APFD values for NSGAII and APCC-FC and APCC-HFC

Objectives	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17
APCC-FC	.811	.713	.954	.857	.970	.739	.839	.885	.468	.811	.907	.818	.982	.952	.922	.000
APCC-HFC	.833	.751	.901	.862	.954	.766	.844	.881	.444	.848	.912	.788	.933	.954	.864	.000

Table 6.7.: Comparison of average APFD values for DENSEA and APCC-FC and APCC-HFC

Objectives	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17
APCC-FC	.769	.619	.877	.777	.878	.675	.793	.795	.319	.741	.888	.696	.960	.878	.848	.000
APCC-HFC	.756	.636	.830	.726	.856	.708	.822	.822	.426	.778	.868	.699	.903	.898	.844	.000

Comparing these two tables for the objective combinations *APCC-HFC* and *APCC-FC* we can see that for the *NSGA II* algorithm the combination *APCC-HFC* has the best average APFD values for the 8 product variants $\{P2, P3, P5, P7, P8, P11, P12, P15\}$ and the objective combination *APCC-FC* for the other 7 product variants. For algorithm *DENSEA* the combination *APCC-HFC* has the best average APFD values for 8 product variants as well.

Since the difference between the two objective combinations *APCC-HFC* and *APCC-FC* is not significant, both objective combinations with only two objectives can be seen as the best objective combinations.

In Appendix B.3 the other objective combinations are compared with other approaches. We can see that every objective combination with objective *Dissimilarity* is worse than other approaches. This is possible because the objective *Dissimilarity* lead to a more diverse test set and, hence, it is possible that the introduced failure in changed parts of the architecture is covered late in the prioritized test set.

Since this thesis focuses multi-objective prioritization the first evaluation used objective combinations with three objectives. In conclusion it can be said, that the prioritization of test sets with the two objective combinations *APCC-Diss-FC* and *APCC-Diss-HFC* lead to a higher APFD value than a random approach or prioritization approaches introduced by Lachmann et al. [57]. In the second evaluation we have discussed which objective combination has shown the best results. Since the two objective combinations *APCC-FC* and *APCC-HFC* has no significantly difference both can be stated as best objective combination.

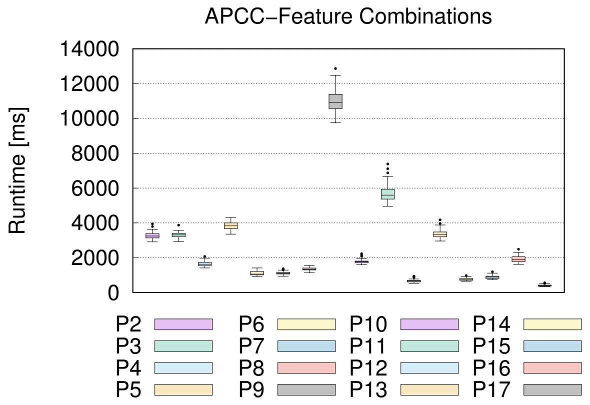
In the following we will discuss how efficient the two used algorithm *NSGAII* and *DENSEA* are and if the prioritization lead to a reduction of the testing effort for SPL.

6.4.2. Results for Research Question 2

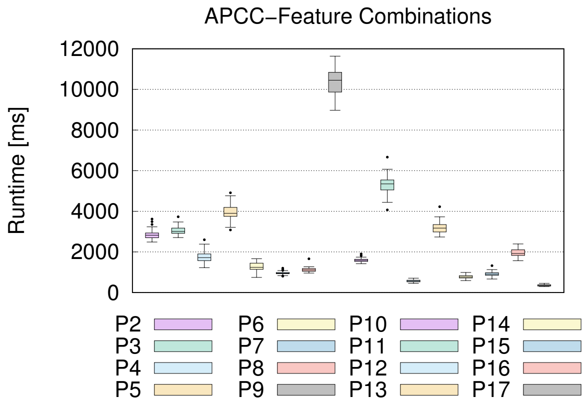
RQ 2: How efficient are the used genetic algorithms?

For the first research question we have discussed the two objective combinations with three objectives and the two best objective combinations. To evaluate if the used GAs are efficient we will discuss the average execution time with regards to these four objective combinations. The box plots of the execution times for the other objective combinations can be found in the appendix B. First we will discuss the average execution times of the two best objective combinations *APCC-FC* and *APCC-HFC*. Afterwards, the average execution time of objective combinations *APCC-Diss-FC* and *APCC-Diss-HFC* are discussed.

The resulting average execution times for objective combination *APCC-FC* and the *NSGA II* algorithm and *DENSEA* algorithm can be found in Figure 6.6.



(a) NSGA II algorithm



(b) DENSEA algorithm

Figure 6.6.: Average Execution time for Objective Combination APCC-FC

Figure 6.6 shows that the execution time median for the objective combination *APCC-FC* for the *NSGA II* algorithm is $\approx$ 11 seconds and for the *DENSEA* algorithm i $\approx$ 10.5 seconds. Additionally, the results indicates an algorithm needs more execution time if more test cases are retestable for a product variant. Product variant P9 which has the most retestable test cases has the slowest execution time and product variant P7 with the least retestable test cases the fastest execution time.

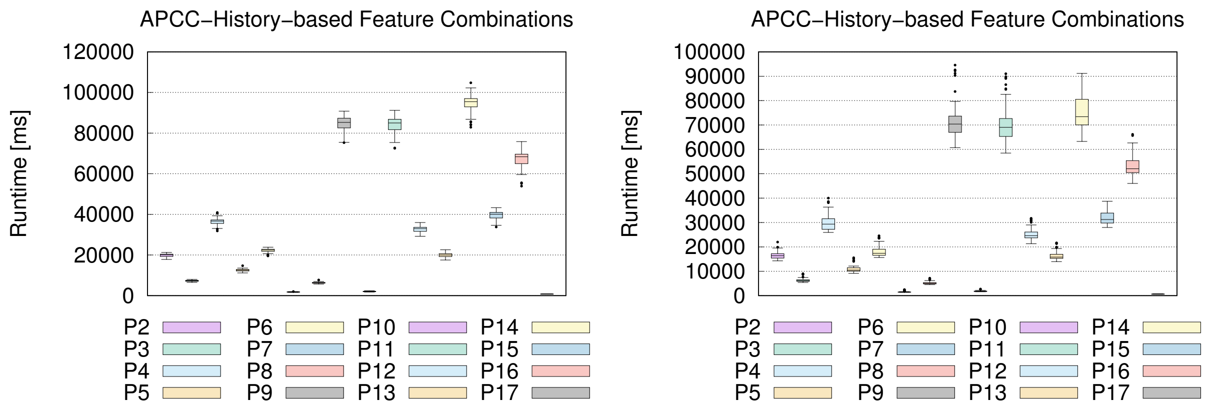
The results from Figure 6.6 lead to the assumption that the execution time of the GA is related to the quantity of retestable test cases for a product variant. The more test cases are retestable the more execution time the GA will need because it has to measure the quality of a higher number of test cases. Table 6.8 shows an overview of the quantity of retestable test cases for the product variants from BCS.

Table 6.8.: Retestable Test Cases for Product Variants in BCS

Product	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17
$\lvert TC_{Retest} \rvert$	1	30	19	37	33	44	13	22	61	14	49	32	27	44	39	48	12

As can be seen in Table 6.8 product variant P9 has the most retestable test cases and product variant P1 the least retestable test cases. However, as mentioned for the evaluation of the first research question, product variant P1 has no prioritized test set due to the fact that it only has one retestable test case. With this fact product variant P17 has the least retestable test cases.

In the previous evaluation we have shown the results of the average execution time for objective combination *APCC-FC*. Now we will show the evaluation for the second objective combination *APCC-HFC*. The resulting average execution times for objective combination *APCC-HFC* and the *NSGA II* algorithm and *DENSEA* algorithm can be found in Figure 6.7.

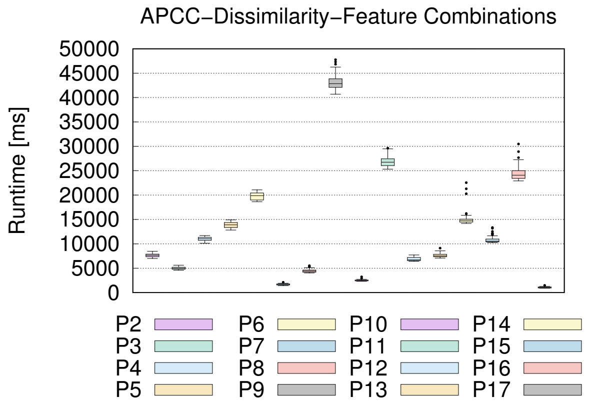


(a) NSGA II algorithm (b) DENSEA algorithm

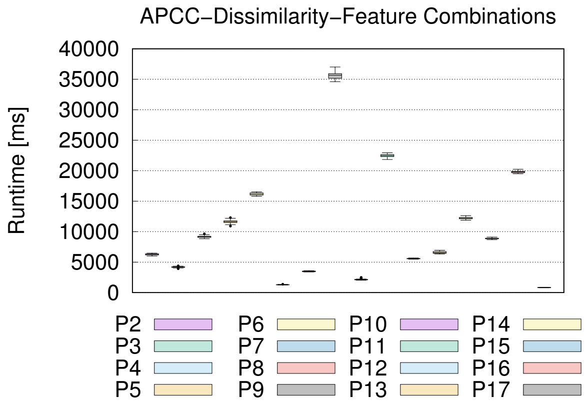
Figure 6.7.: Average Execution Time for Objective Combination APCC-HFC

The resulting execution times in Figure 6.7 of the objective combination *APCC-HFC* indicates that the execution time is not only depending on the number of retestable test cases for a product variant but also depending on the objectives which are used to evaluate a test set. The objective *HFC* is an objective which requires a high computational effort due to the computation of the weights for every feature combination which is covered by a test case. The more feature combinations a test case covers, the more weights have to be summarized as described in Chapter 4.1.4. If we compare the execution time of product variant P9 and P14 and the retestable test cases we can observe that P14 with just 44 retestable test cases has a higher execution time than P9 with 61 retestable test cases. Test cases for product variant P14 covers more feature combinations and therefore, more weights have to be summarized. Additionally, the results in Figure 6.7 indicates that the *DENSEA* algorithm has a faster execution time than the *NSGA II* algorithm. Nevertheless, the execution time for this objective combination is less than 110 seconds which is faster than a manual prioritization of test sets.

In the previous evaluation we discuss the average execution times for objective combinations with two objectives. Other interesting average execution times are the time which is needed for the two objective combinations with three objectives. The resulting average execution times for objective combination *APCC-Diss-FC* and the *NSGA II* algorithm and *DENSEA* algorithm can be found in Figure 6.8.



(a) NSGA II algorithm



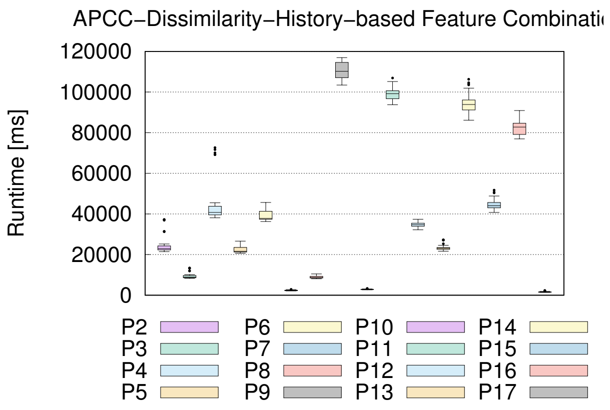
(b) DENSEA algorithm

Figure 6.8.: Average Execution Time for Objective APCC-Diss-FC

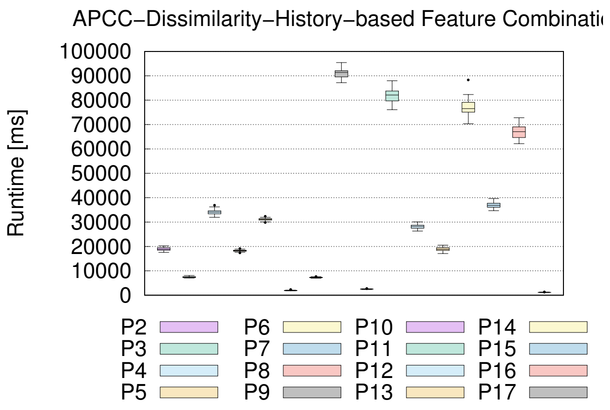
The results of the average execution time for the objective combination *APCC-Diss-FC* indicates that the evaluation of three objectives needs a maximum of $\approx$ 50 seconds. Additionally, it is shown that the *NSGA II* algorithm is less efficient than the *DENSEA* algorithm. The *NSGA II* algorithm needs a maximum of $\sim$ 50 seconds for the evaluation of product variant P9 while the *DENSEA* algorithm needs only a maximum of $\sim$ 40 seconds. However, the average execution time for the objective combination *APCC-Diss-FC* is under $\sim$ 50 seconds and, hence, faster than a manual prioritization of test sets.

After discussing the execution time of the objective combination *APCC-Diss-FC* we will discuss the execution time of the objective combination *APCC-Diss-HFC*. The resulting average execution times for objective combination *APCC-Diss-HFC* and the *NSGA II* algorithm and *DENSEA* algorithm can be found in Figure 6.9.

For objective combination *APCC-Diss-HFC* the results for the average execution time indicates that this objective combination has the highest execution time. For product variant P9 this combination has the highest average execution time and needs up to $\sim$ 120 seconds. For objective combination *APCC-HFC* product variant P14 has the highest average execution time with up to $\sim$ 100



(a) NSGA II algorithm



(b) DENSEA algorithm

Figure 6.9.: Average Execution Time for Objective APCC-Diss-HFC

seconds. Additionally, the results indicate that product variant P9 with the most retestable test cases has the highest average execution time once again. This is due to the fact, that the objective *Diss* measures the dissimilarity to all previous test cases in a test set and, hence, need more time if more test cases are in the test set. Nevertheless, the average execution time for objective combination *APCC-Diss-HFC* is up to $\sim$ 120 seconds and faster than a manual prioritization as well.

The evaluation of the second research question has shown that the average execution time in the worst case is up to $\sim$ 120 seconds for the objective combination *APCC-Diss-HFC*. The objective combination *APCC-Diss-HFC* has the two objectives Diss and HFC with a high computational effort to evaluate the test sets. The best objective combinations *APCC-FC* and *APCC-HFC* for this thesis have an average execution time under 110 seconds. However, this average execution time is faster than a manual prioritization of test sets.

6.4.3. Conclusion of Evaluation

The conclusion of the evaluation of the two research questions in this chapter is that with the introduced concept of this thesis the testing effort can be reduced compared to a random approach and the two approaches from Lachmann et al. [57]. The evaluation of the first research question has shown that the introduced objective combinations will lead to an effective test set. The prioritized test sets from our approach got a higher average APFD value than the prioritized test sets from a random approach and two approaches introduced by Lachmann et al. [57]. It is worth to mention that the combinations with three objectives are not better than the objective combinations *APCC-FC* and *APCC-HFC*.

Additionally, the evaluation of the efficiency of the algorithms has shown that the prioritization of the test sets is achieved in a maximum of 120 seconds. For the prioritization of test sets the algorithm *NSGA II* leads to better APFD results with a similar execution time like the *DENSEA* algorithm. Therefore, we suggest to use the *NSGA II* algorithm.

6.5. Threats to Validity

The experimental results of this thesis are based on the BCS SPL study. Since this is the only study which is used to evaluate the concept this can affect the accuracy of the evaluation. However, for the first evaluation of the concept the used study is a good starting point and we have achieved satisfying results in both quality and runtime.

The BCS SPL study only consists of 92 test cases which is a rather small quantity. Therefore, a random algorithm has a higher chance to select test cases which get good values for the specified objectives. Additionally, we reduce the number of possible test cases in the test set for product variants, as we use only retestable test cases. However, even on this small scale our approach has shown better results for the *NSGAII* algorithm, which indicates that it would have an even higher impact on a more diverse case study.

Likewise, the study does not provide failures and we have to introduce random failures. The random introduction of a failure affects the generalizability of the results, because we only introduce failures in changed parts of the architecture. The APCC measures at which position changes are covered and ranks test cases higher which cover changes ealier. Introducing failures in changed parts can lead to a better APFD value of objective combinations with the APCC objective. The other three objectives focus on more realistic scenarios. In the results we have seen that the objective APCC is mostly the best objective. However, we have shown which other objective combinations are likely to the APCC.

The introduced concept has been prototyped for this thesis and, thus, might contain bugs. To this end, we performed intensive testing and used the framework jMetal to define the objectives which were specified in this thesis. The framework jMetal is well tested and, hence, the implementations of the GAs are accurate.

7 Conclusion and Future Work

The purpose of this thesis was to design and evaluate a search-based delta-oriented test set prioritization concept. For the prioritization of test sets meta heuristic search techniques, i.e., genetic algorithms, should be used. Genetic algorithms are capable to optimize a problem for multiple objectives which should be used to prioritize a test set for more than one objective. With the information of previous tested product variants the redundancy in testing should be reduced. Previous covered parts of the architecture or covered feature combinations should become a lower prioritization than not covered elements.

Therefore, the background was presented at first. The fundamentals of software product lines, delta-oriented software product lines and delta-oriented modeling were described. Afterwards, the principles of software testing and especially software product line testing were explained. In this context regression testing and model-based testing were described in particular. Since, this thesis uses GAs to prioritize test sets, the fundamentals of search-based testing and genetic algorithms were explained.

Afterwards, related research work was discussed. Different existing techniques, which use search-based approaches to select or prioritize test data, e.g., test cases, product variants, to test were presented and this thesis was dissociated from the presented work.

Based on the fundamentals described in 2 the concept of this thesis was described. Four objectives were presented to support the prioritization of a test set. The first objective adapts the underlying idea of sampling methods. Sampling methods use the observation that most faults in SPL testing occur due to the interaction between a small number of features [7], hence, first objective focuses on retesting pairwise feature combinations. This means if a feature combination was not covered by a previous tested product variant and the retestable test cases this feature combination should be covered earlier. The second objective is an alternative of the first objective. The objective adapts a history-based approach. This approach uses the information how often a feature combination was covered in previous tested product variants. With these information test cases are prioritized which covers feature combinations which were not covered often. The third objective focuses on changes between product variants and ranks test cases higher which cover many of changes. Therefore, the APCC metric from Lachmann et al. [57] is used. The fourth objective aims to create a more diverse test set. Some prioritization techniques lead to clustering of similar test cases. But similar test cases have the similar capability of revealing faults [5]. Therefore, the fourth objective measures the dissimilarity of test cases and orders them in a way that they are more diverse.

Afterwards, the prototype implementation *DoMoRe Testing* which realizes the concept in Java is described. A wizard is provided to select several options. It is possible to select which product variants should be optimized, which algorithm should be used and which objectives are used to prioritize the test set. To implement the concept the framework jMetal [13] was used. This framework provides implementations of several genetic algorithms and an easy way to implement own problem specifications.

The developed concept was evaluated based on the *Body Comfort System* study. Therefor all possible product variants and its retestable test cases were prioritized. The prioritization of the test cases for a product variant was executed using every possible objective combination with the introduced objectives in the concept. To measure the efficiency of each objective combination the *Average Percentage of Faults Detected* (APFD) metric introduced by Rothermel et al. [18] was measured for each test set.

In the following we discuss the results and insights which were made in this thesis.

7.1. Conclusion

The evaluation of the concept introduced in this thesis has shown that the testing effort for SPLs can be reduced using GAs. Overall, the comparison to other approaches has shown that the prioritization of test sets with multiple objectives leads to a better fault detection rate.

To achieve this improvement many informations about the delta-oriented SPL are necessary because the introduced objectives use this informations to evaluate a test set. Important informations are the test cases for the SPL and the delta descriptions. The delta informations are used to check which feature combinations are covered by a test case. Without this informations the two objective feature combinations and history-based feature combinations are not applicable. Additionally, the features for a SPL and feature configurations for each product variant are used to check which absent features a test case covers. The evaluation has shown that the execution time of the GAs increase with the number of test cases in the test set. Therefore, the categorization of test cases for each product variant is necessary.

The evaluation has shown that objectives which measures the quality of a test set by comparing it with all previous test cases in the test set, e.g, dissimilarity, has a higher execution time. Likewise, an objective which measures the quality of a test set using a high number of informations, e.g., history-based feature combinations, has a high execution time. Therefore, the design of objectives is an important task. If the performance of a computation of the fitness for an objective is low this objective has a low scalability for the number of test cases in the individual. Overall it can be stated that the time for the prioritization with a GA is inferior than a manual prioritization.

Additionally, the introduced approach in this thesis can be extended with more objectives and operators for the GA.

7.2. Future Work

In future work, it is possible to define more objectives for the prioritization of test sets. It is conceivable to define objectives which focus on coverage criteria, e.g., structural coverage criteria like all connections. Likewise, the implemented prototype can be easily enhanced with new objectives because it uses well defined interfaces.

The evaluation has shown that the introduced approach for the history-based feature combination is not ideal because the values are between -1 and 1. The presented metric to measure the quality of a test set leads to not ideal values and, hence, the metric should be revised. In the current metric the weights w_{FC_i} of each feature combination for a test case are summarized. A possible modification of this metric is to summarize the weight and then normalize the overall weight with the quantity of covered feature combinations.

Additionally, further evaluations of the presented concept are necessary to proof the results from the evaluation in this thesis. More complex evaluations using industrial data are required to assess the actual improvements on realistic data. Likewise, for a genetic algorithm a lot of refinements, e.g., the mutation probability, the population size, are possible in further evaluations.

We presented several operators for the process of selection and crossover in a genetic algorithm. Some of these operators are not implemented in the used framework jMetal. However, the framework can be easily extended with the presented operators. It is possible that other operators will lead to another prioritization of test sets.

Bibliography

[1] M. Aigner. *Kombinatorik, 1. Grundlagen und Zähltheorie.* Springer Verlag, 1975.

[2] Andreas Spillner und Tilo Linz. *Basiswissen Softwaretest.* 5. dpunkt.verlag, 2012.

[3] H. Baller, S. Lity, M. Lochau, and I. Schaefer. "Multi-objective test suite optimization for incremental product family testing". In: *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on.* IEEE. 2014, pp. 303–312.

[4] T. Blickle and L. Thiele. "A comparison of selection schemes used in evolutionary algorithms". In: *Evolutionary Computation* 4.4 (1996), pp. 361–394.

[5] E. G. Cartaxo, P. D. L. Machado, and F. G. O. Neto. "On the use of a similarity function for test case selection in the context of model-based testing". In: *Software Testing, Verification and Reliability* 21.2 (2011), pp. 75–100. ISSN: 1099-1689.

[6] D. Clarke, M. Helvensteijn, and I. Schaefer. "Abstract Delta Modeling". In: *SIGPLAN Not.* 46.2 (10/2010), pp. 13–22. ISSN: 0362-1340.

[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. "The AETG system: an approach to testing based on combinatorial design". In: *IEEE Transactions on Software Engineering* 23.7 (07/1997), pp. 437–444.

[8] M. B. Cohen, M. B. Dwyer, and J. Shi. "Interaction testing of highly-configurable systems in the presence of constraints". In: *Proceedings of the 2007 international symposium on Software testing and analysis.* ACM. 2007, pp. 129–139.

[9] D. Beasley, D. R. Bull, R. R. Martin. "An Overview of Genetic Algorithms: Part 1, Fundamentals". In: *University Computing* (1993), pp. 58–69.

[10] L. Davis. "Applying adaptive algorithms to epistatic domains." In: *IJCAI.* Vol. 85. 1985, pp. 162–164.

[11] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II". In: *Parallel problem solving from nature PPSN VI.* Springer. 2000, pp. 849–858.

[12] X. Devroey, G. Perrouin, A. Legay, P.-Y. Schobbens, and P. Heymans. "Search-based Similarity-driven Behavioural SPL Testing". In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems.* VaMoS '16. Salvador, Brazil: ACM, 2016, pp. 89–96. ISBN: 9781450340199.

[13] J. J. Durillo and A. J. Nebro. "jMetal: A Java framework for multi-objective optimization". In: *Advances in Engineering Software* 42 (2011), pp. 760–771. ISSN: 0965-9978.

[14] E. Engström, P. Runeson, and A. Ljung. "Improving Regression Testing Transparency and Efficiency with History-Based Prioritization – An Industrial Case Study". In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation.* 03/2011, pp. 367–376.

[15] E. Engström and P. Runeson. “Software product line testing–a systematic mapping study”. In: *Information and Software Technology* 53.1 (2011), pp. 2–13.

[16] A. Ensan, E. Bagheri, M. Asadi, D. Gasevic, and Y. Biletskiy. “Goal-oriented test case selection and prioritization for product line feature models”. In: *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on.* IEEE. 2011, pp. 291–298.

[17] G. Dong and J. Pei. *Sequence Data Mining.* Springer, 2007.

[18] G. Rothermel, R. H. Untch, C. Chu and M. J. Harrold. “Prioritizing Test Cases For Regression Testing”. In: *IEEE Transactions on software engineering* Vol.27 No.10 (2001), pp. 929–948.

[19] D. E. Goldberg and R. Lingle. “Alleles, loci, and the traveling salesman problem”. In: *Proceedings of an International Conference on Genetic Algorithms and Their Applications.* Vol. 154. Lawrence Erlbaum, Hillsdale, NJ. 1985, pp. 154–159.

[20] D. Greiner, J. M. Emperador, G. Winter, and B. Galván. “Improving Computational Mechanics Optimum Design Using Helper Objectives: An Application in Frame Bar Structures”. In: *Evolutionary Multi-Criterion Optimization: 4th International Conference, EMO 2007, Matsushima, Japan, March 5-8, 2007. Proceedings.* Ed. by S. Obayashi, K. Deb, C. Poloni, T. Hiroyasu, and T. Murata. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 575–589. ISBN: 9783540709282.

[21] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge university press, 1997.

[22] H. Muccini, M. Dias and D. J. Richardson. “Software architecture-based regression testing”. In: *The Journal of Systems and Software* 79 (2006), 1379–1396.

[23] A. Haber, H. Rendel, B. Rumpe, and I. Schaefer. “Delta modeling for software architectures”. In: *arXiv preprint arXiv:1409.2358* (2014).

[24] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake. “Similarity-based prioritization in software product-line testing”. In: *Proceedings of the 18th International Software Product Line Conference-Volume 1.* ACM. 2014, pp. 197–206.

[25] B. H. F. Hasan and M. S. M. Saleh. “Evaluating the effectiveness of mutation operators on the behavior of genetic algorithms applied to non-deterministic polynomial problems”. In: *Informatica* 35.4 (2011).

[26] H. Hemmati and L. Briand. “An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection”. In: *2010 IEEE 21st International Symposium on Software Reliability Engineering.* 11/2010, pp. 141–150.

[27] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. “Multi-objective test generation for software product lines”. In: *Proceedings of the 17th International Software Product Line Conference.* ACM. 2013, pp. 62–71.

[28] F. Herrera, M. Lozano, and J. L. Verdegay. “Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis”. In: *Artificial intelligence review* 12.4 (1998), pp. 265–319.

[29] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[30] *ISO/DIS 26262-1 - Road vehicles - Functional safety.* 2009.

[31] A. Jaaksi. “Developing mobile browsers in a product line”. In: *IEEE software* 19.4 (2002), p. 73.

[32] M. F. Johansen,. Haugen, and F. Fleurey. “An algorithm for generating t-wise covering arrays from large feature models”. In: *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM. 2012, pp. 46–55.

[33] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. “On the Impact of the Optional Feature Problem: Analysis and Case Studies”. In: *Proceedings of the 13th International Software Product Line Conference*. SPLC ’09. San Francisco, California, USA: Carnegie Mellon University, 2009, pp. 181–190.

[34] Klaus Pohl, Günter Böckele und Franck von der Linden. *Software Product Line Engineering*. Springer Verlag, 2005.

[35] A. Konak, D. W. Coit, and A. E. Smith. “Multi-objective optimization using genetic algorithms: A tutorial”. In: *Reliability Engineering & System Safety* 91.9 (2006). Special Issue - Genetic Algorithms and ReliabilitySpecial Issue - Genetic Algorithms and Reliability, pp. 992–1007. issn: 0951-8320.

[36] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak und A. Spencer Peterson. *Feature-oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, 1990.

[37] R. Lachmann. “Konzeption und Evaluation eines delta-orientierten modellbasierten Integrationstestverfahrens für Softwareproduktlinien”. MA thesis. Technische Universität Carolo-Wilhelmina zu Braunschweig, 04/2013.

[38] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic. “Genetic algorithms for the travelling salesman problem: A review of representations and operators”. In: *Artificial Intelligence Review* 13.2 (1999), pp. 129–170.

[39] H. K. N. Leung and L. White. “Insights into regression testing [software testing]”. In: *Software Maintenance, 1989., Proceedings., Conference on*. 10/1989, pp. 60–69.

[40] P. Liggesmeyer. *Software-Qualität, Testen. Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2002.

[41] S. Lischke. *Delta-orientierte Regressionsstrategien für Integrationstests von Softwareproduktlinien*. Bachelor Thesis. 2014.

[42] S. Lity, M. Lochau, I. Schaefer, and U. Goltz. “Delta-oriented model-based SPL regression testing”. In: *Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on*. IEEE. 2012, pp. 53–56.

[43] M. Lochau, S. Lity, R. Lachmann, I. Schaefer, and U. Goltz. “Delta-oriented model-based integration testing of large-scale systems”. In: *Journal of Systems and Software* 91 (2014), pp. 63–84.

[44] M. Lochau, S. Oster, U. Goltz, and A. Schürr. “Model-based pairwise testing for feature interaction coverage in software product line engineering”. In: *Software Quality Journal* 20.3-4 (2012), pp. 567–604.

[45] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. "Incremental model-based testing of delta-oriented software product lines". In: *International Conference on Tests and Proofs*. Springer. 2012, pp. 67–82.

[46] Mary Jean Harrold. "Testing : A Roadmap". In: *ICSE '00 Proceedings of the Conference on The Future of Software Engineering*. ACM New York, NY, USA 2000, 2000-05-01, pp. 61–72.

[47] P. McMinn. "Search-Based Software Testing: Past, Present and Future". In: *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 153–163. ISBN: 9780769543451.

[48] P. McMinn. "Search-based software test data generation: A survey". In: *Software Testing Verification and Reliability* 14.2 (2004), pp. 105–156.

[49] M. Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[50] T. Müller, M. Lochau, S. Detering, F. Saust, H. Garbers, L. Märtin, T. Form, and U. Goltz. *A comprehensive Description of a Model-based, continuous Development Process for AUTOSAR Systems with integrated Quality Assurance*. Tech. rep. Technical Report 2009-06, TU Braunschweig, 2009.

[51] L. Northrop and P. Clements. "Software product lines". In: *URL http://www. sei. cmu. edu/library/assets/Philips* 4.05 (2001).

[52] *OMG Specification Superstructure and Infrastructure*. `http://www.omg.org/spec/UML/2.4.1/`. Stand: 2014-09-03.

[53] S. Oster, A. Schürr, and I. Weisemöller. "Towards software product line testing using story driven modeling". In: *Proceedings of 6th International Fujaba Days* (2008), pp. 48–55.

[54] S. Oster, M. Zink, M. Lochau, and M. Grechanik. "Pairwise feature-interaction testing for SPLs: potentials and limitations". In: *Proceedings of the 15th International Software Product Line Conference, Volume 2*. ACM. 2011, p. 6.

[55] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. "Pairwise testing for software product lines: comparison of two approaches". In: *Software Quality Journal* 20.3-4 (2012), pp. 605–643.

[56] P.N. Tan, M.Steinbach and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2006.

[57] R. Lachmann, S. Lity, S. Lischke, S. Beddig, S. Schulze, I. Schaefer. "Delta-oriented test case prioritization for integration tesing of software product lines". In: *In SPLC* (2015), pp. 81–90.

[58] M. Safe, J. Carballido, I. Ponzoni, and N. Brignole. "On Stopping Criteria for Genetic Algorithms". In: *Advances in Artificial Intelligence – SBIA 2004: 17th Brazilian Symposium on Artificial Intelligence, Sao Luis, Maranhao, Brazil, September 29-Ocotber 1, 2004. Proceedings*. Ed. by A. L. C. Bazzan and S. Labidi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 405–413. ISBN: 9783540286455.

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

[60] Sascha Lity, Remo Lachmann, Malte Lochau and Ina Schaefer. *Delta-oriented Software Product Line Test Models - The Body Comfort System vase Study*. Tech. rep. Institute for Programming and Reactive Systems, Institute for Software Engineering and Automotive Informatics and Real-Time Systems Lab, 2013.

[61] I. Schaefer. “Variability Modelling for Model-Driven Development of Software Product Lines.” In: *VaMoS* 10 (2010), pp. 85–92.

[62] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. “Delta-oriented programming of software product lines”. In: *Software Product Lines: Going Beyond*. Springer, 2010, pp. 77–91.

[63] I. Schaefer and F. Damiani. “Pure delta-oriented programming”. In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. ACM. 2010, pp. 49–56.

[64] I. Schaefer, A. Worret, and A. Poetzsch-Heffter. “A model-based framework for automated product derivation”. In: *1st International Workshop on Model-Driven Approaches in Software Product Line Engineering*. 2009, pp. 14–21.

[65] G. Syswerda. “Schedule optimization using genetic algorithms”. In: *Handbook of genetic algorithms* (1991).

[66] A. Tevanlinna, J. Taina, and R. Kauppinen. “Product family testing: a survey”. In: *ACM SIGSOFT Software Engineering Notes* 29.2 (2004), pp. 12–12.

[67] S. Thiel and A. Hein. “Modeling and using product line variability in automotive systems”. In: *IEEE software* 19.4 (2002), p. 66.

[68] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[69] S. Wang, S. Ali, and A. Gotlieb. “Minimizing test suites in software product lines using weight-based genetic algorithms”. In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM. 2013, pp. 1493–1500.

[70] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, and M. Liaaen. “Multi-objective test prioritization in software product line testing: an industrial case study”. In: *Proceedings of the 18th International Software Product Line Conference-Volume 1*. ACM. 2014, pp. 32–41.

[71] S. Yoo and M. Harman. “Regression testing minimization, selection and prioritization: a survey”. In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120.

[72] M. Zink. *Anwendung von MoSo-PoLiTe in einer Automotive SPL*, 2011.

Abbreviations

SPL Software Product Line

GA Genetic Algorithm

DOM Delta-oriented Modeling

FM Feature Model

PMX Partially-Mapped

APFD Average Percentage of Faults Detected

APCC Average Percentage of Changes Covered

HFC History-based Feature Combinations

FC Feature Combinations

Diss Dissimilarity

A Class Diagram

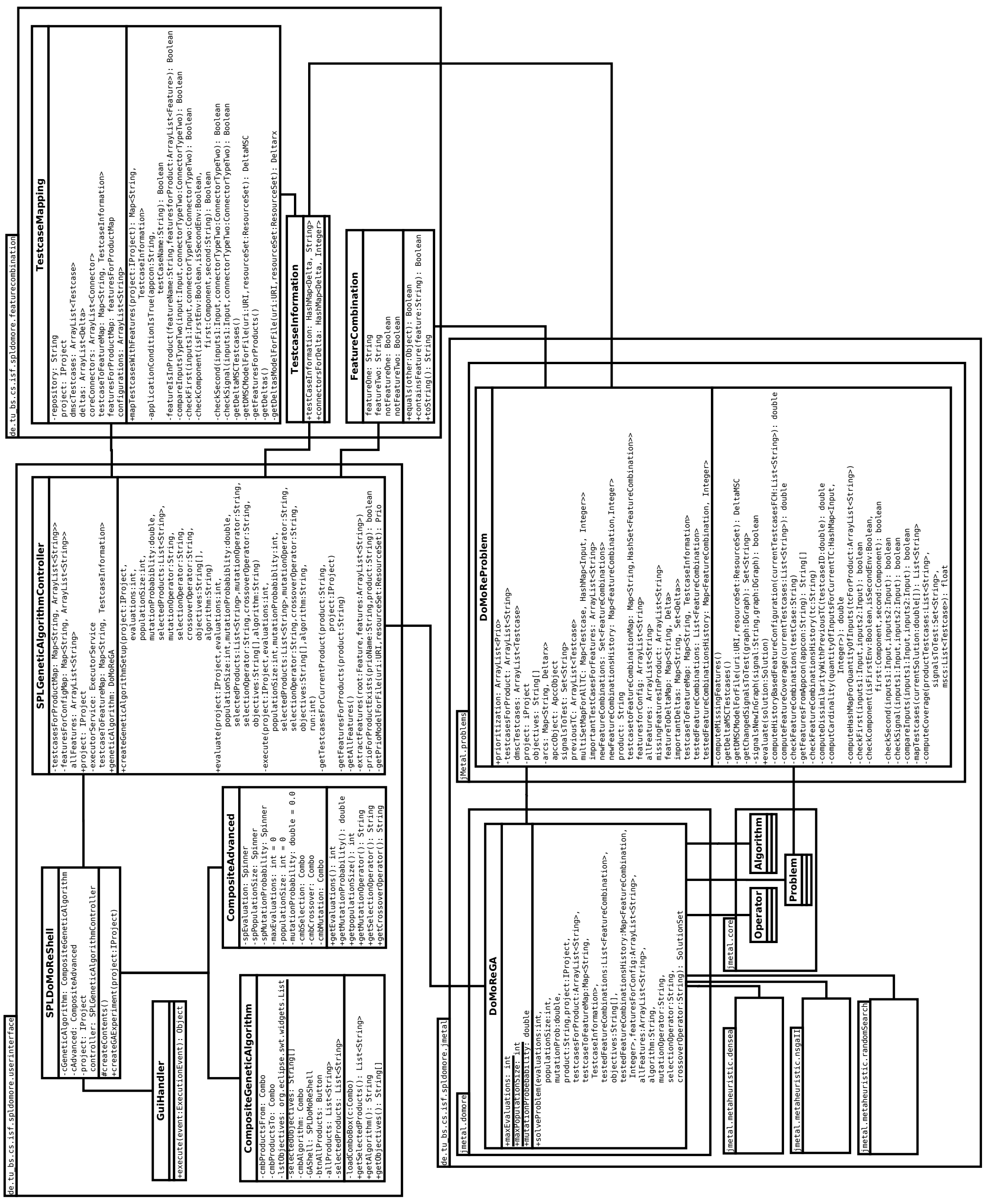

de.tu_bs.cs.isf.spldomore.featurecombination
TestcaseMapping
TestcaseInformation
FeatureCombination
de.tu_bs.cs.isf.spldomore.userinterface
SPLGeneticAlgorithmController
SPLDoMoReShell
GuiHandler
CompositeAdvanced
CompositeGeneticAlgorithm
de.tu_bs.cs.isf.spldomore.jmetal
jmetal.problems
DoMoReProblem
jmetal.domore
DoMoReGA
jmetal.core
Algorithm
Problem
Operator
jmetal.metaheuristic.densea
jmetal.metaheuristic.nsgaII
jmetal.metaheuristic.randomSearch

Figure A.1.: Class Diagram of SPL DoMoRe Tool

B Evaluation

B.1. NSGAII

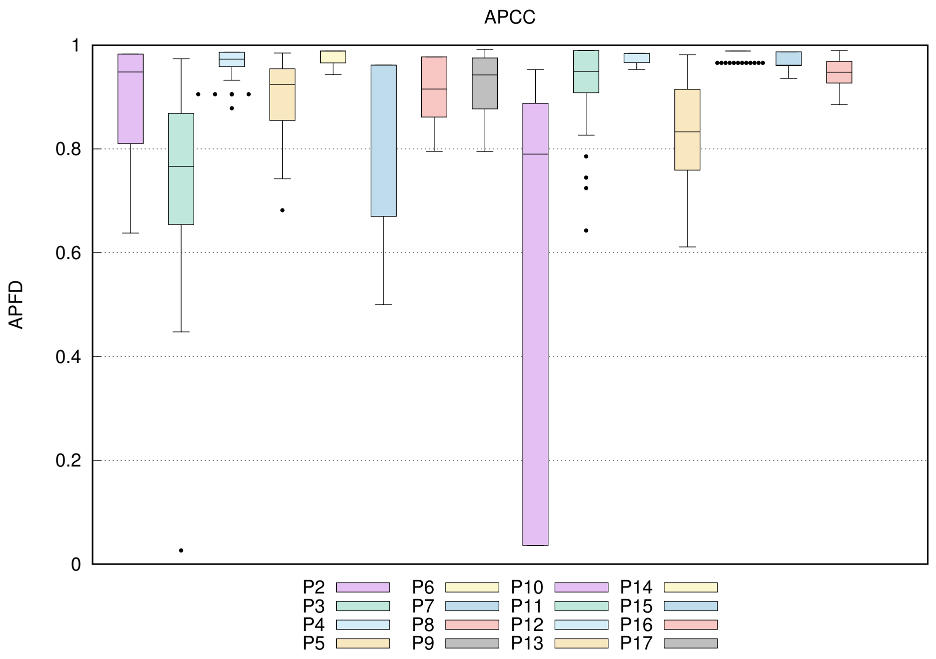


Figure B.1.: Boxplot for NSGAII Objective APCC

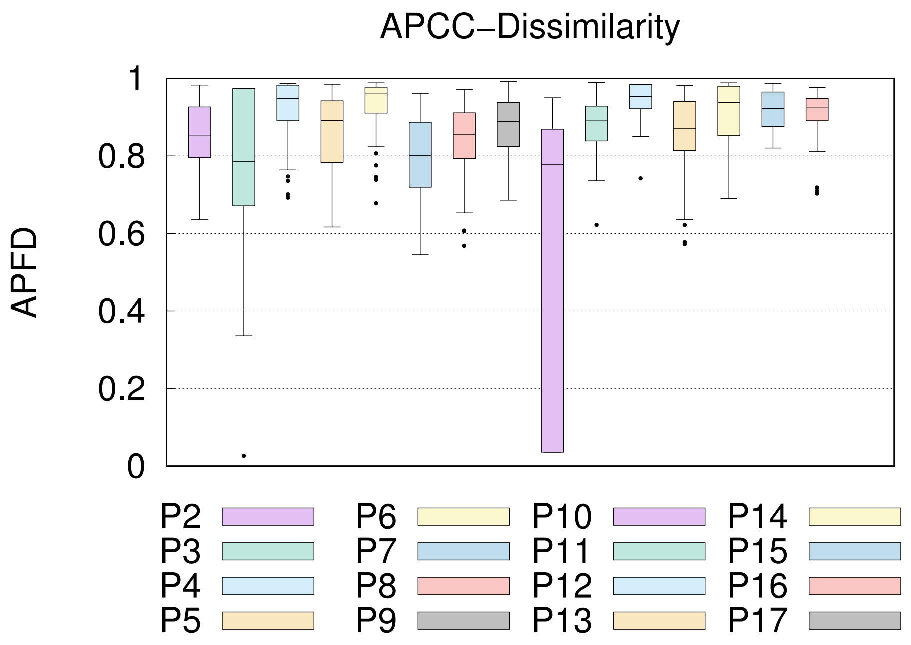


Figure B.2.: Boxplot for NSGAII Objective APCC and Diss

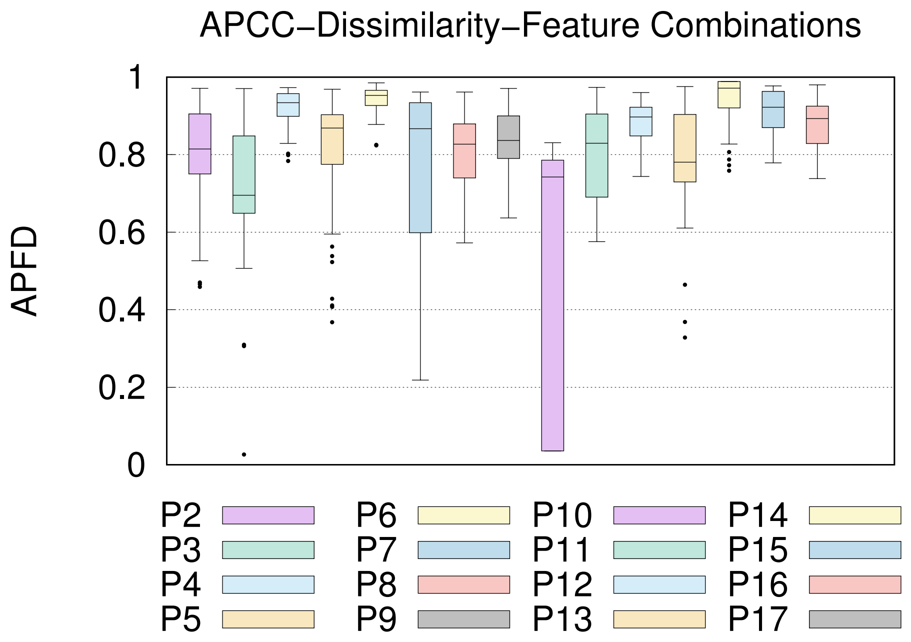


Figure B.3.: Boxplot for NSGAII Objective APCC, Diss and FC

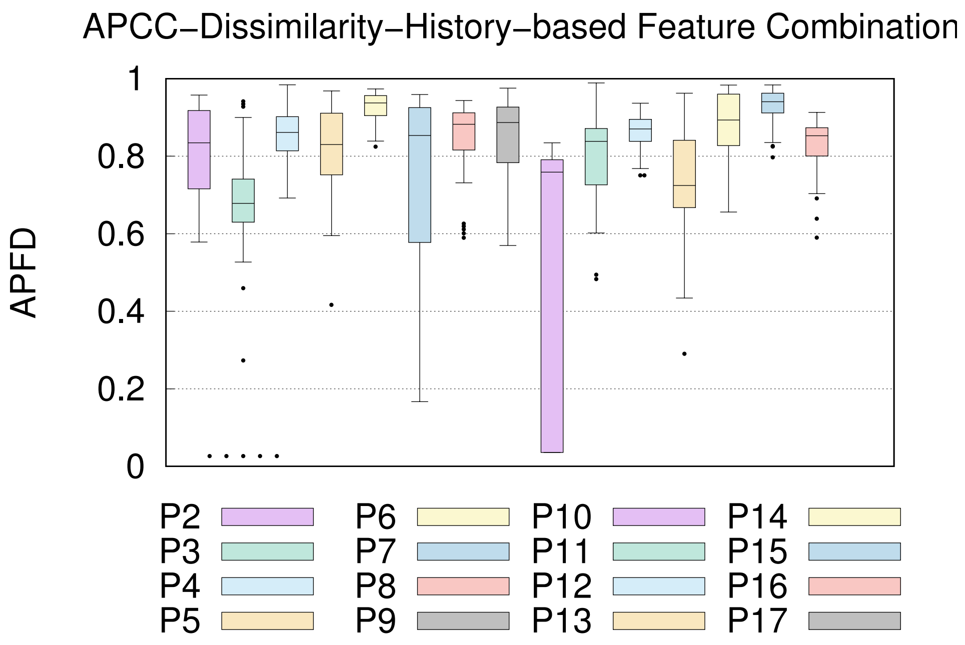


Figure B.4.: Boxplot for NSGAII Objective APCC, Diss and HFC

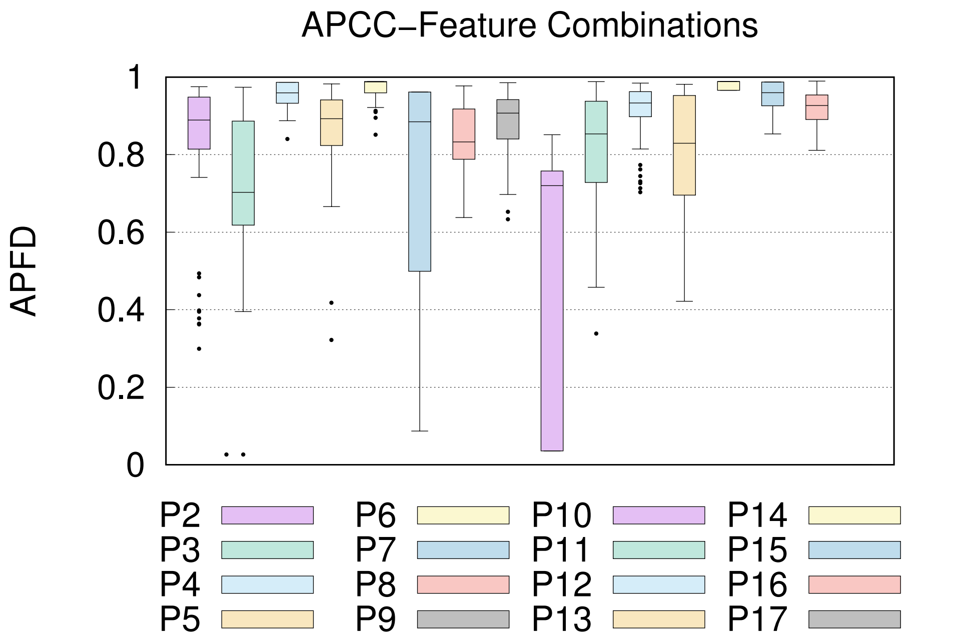


Figure B.5.: Boxplot for NSGAII Objective APCC and FC

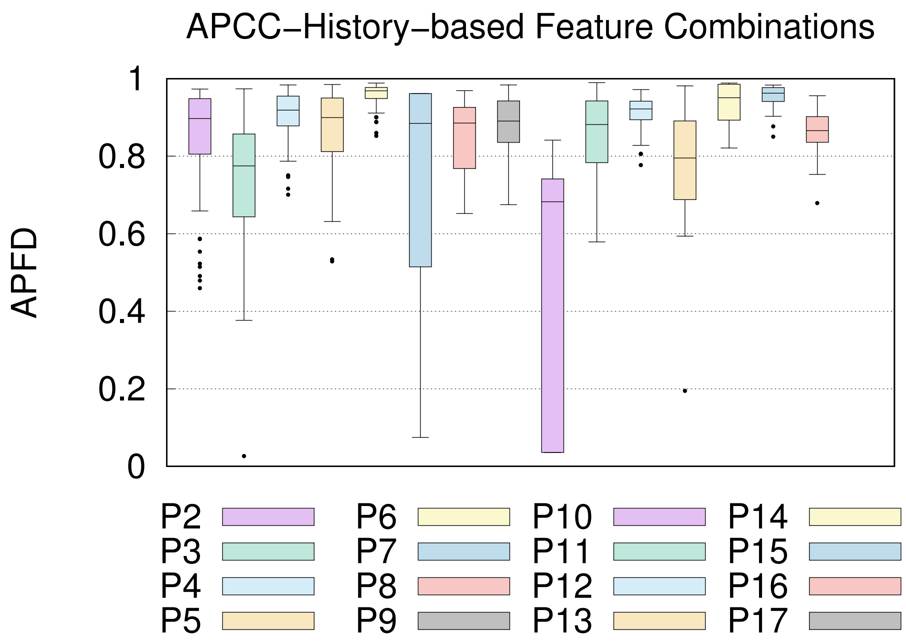


Figure B.6.: Boxplot for NSGAII Objective APCC and HFC

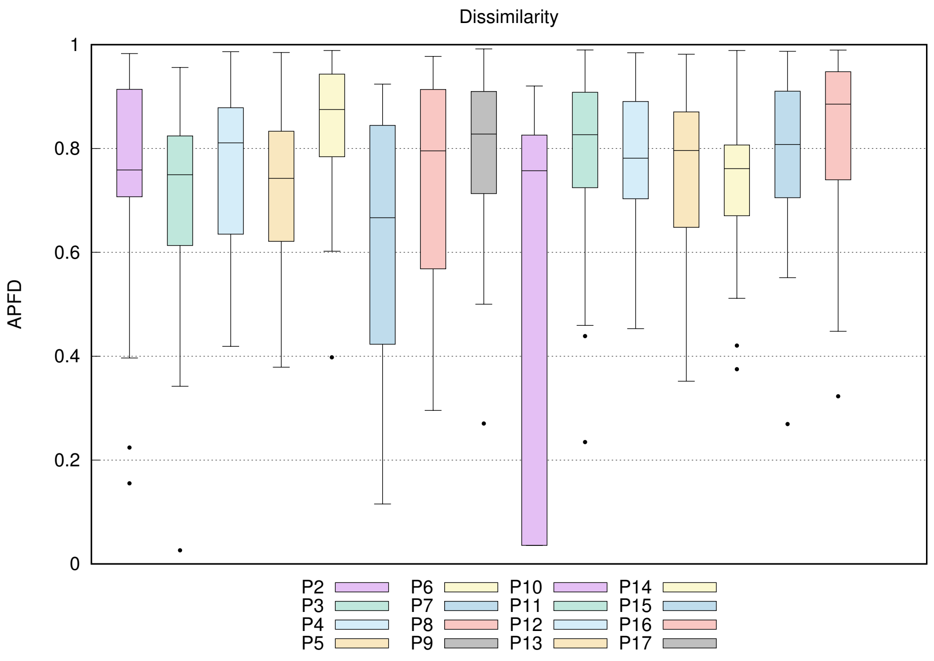


Figure B.7.: Boxplot for NSGAII Objective Diss

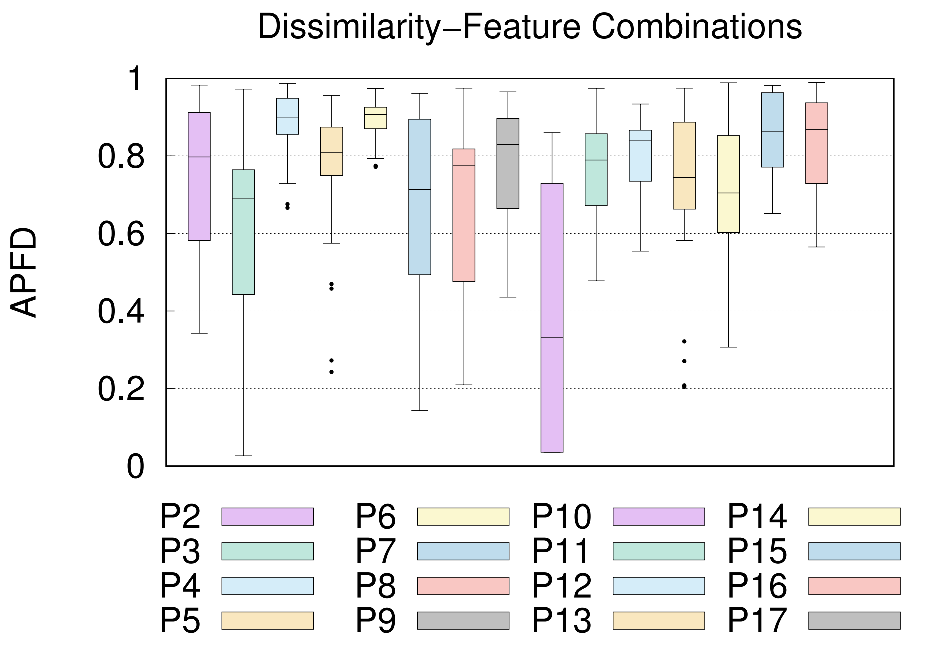


Figure B.8.: Boxplot for NSGAII Objective Diss and FC

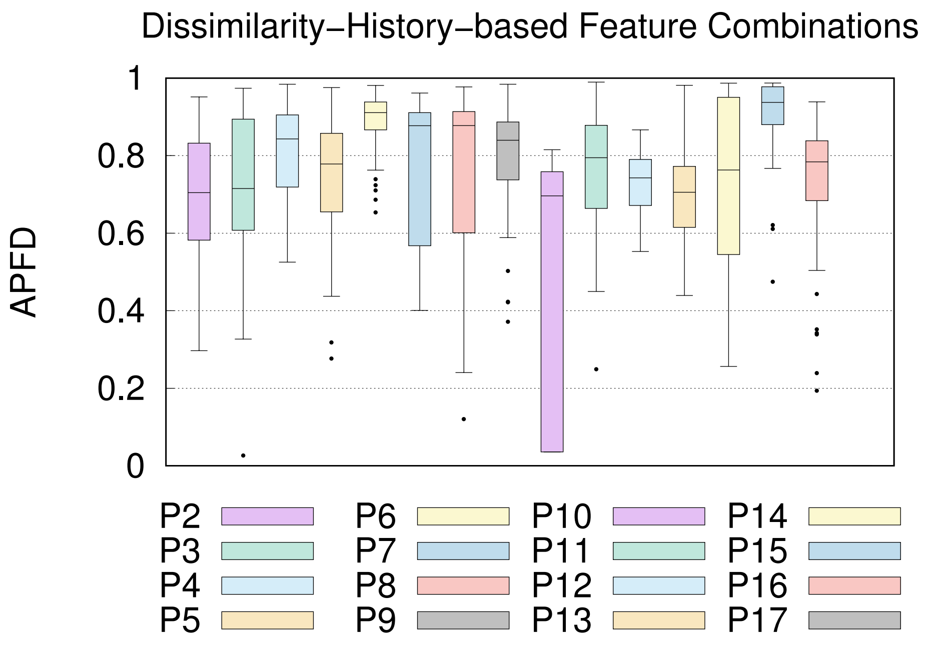


Figure B.9.: Boxplot for NSGAII Objective Diss and HFC

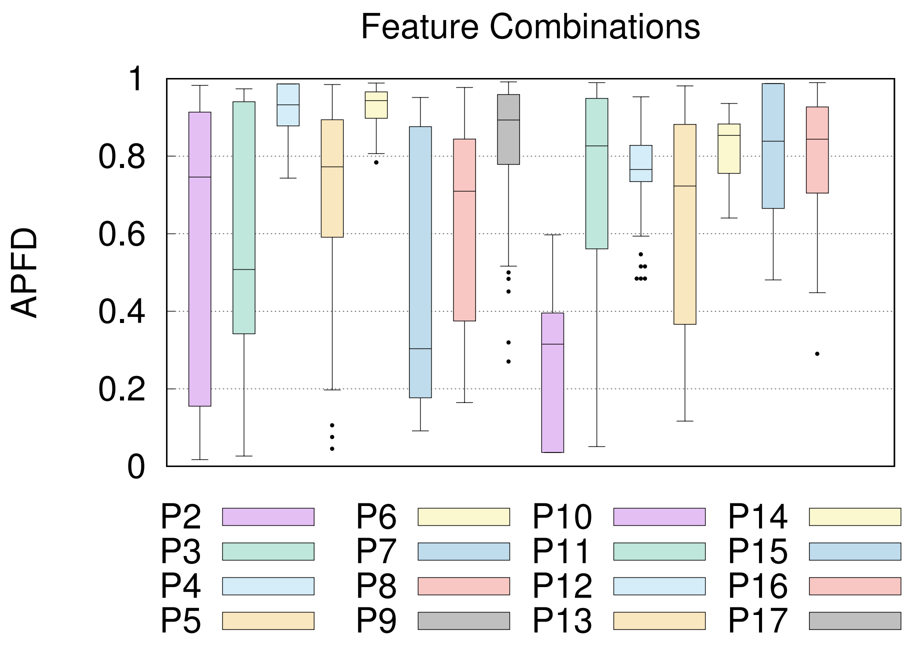


Figure B.10.: Boxplot for NSGAII Objective FC

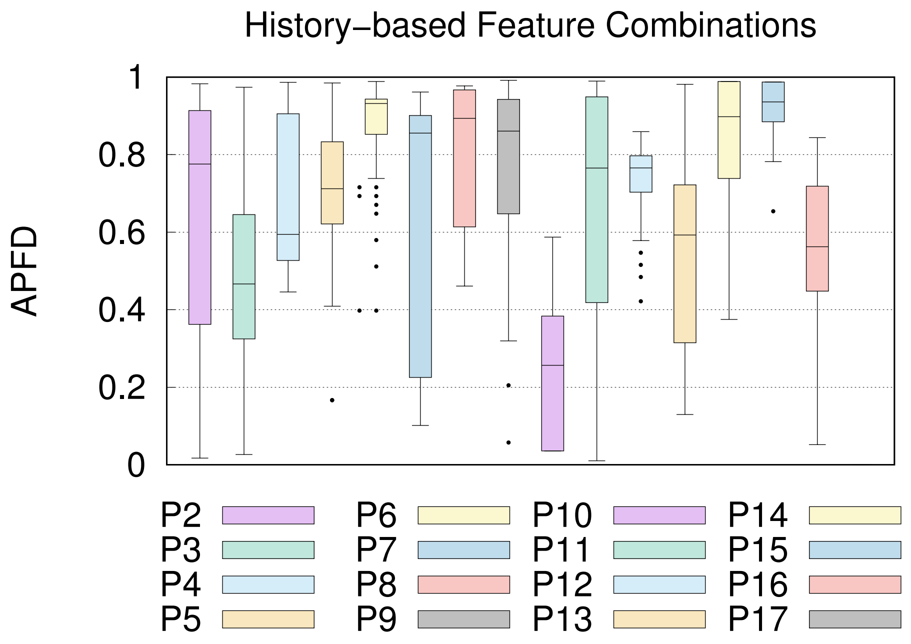


Figure B.11.: Boxplot for NSGAII Objective HFC

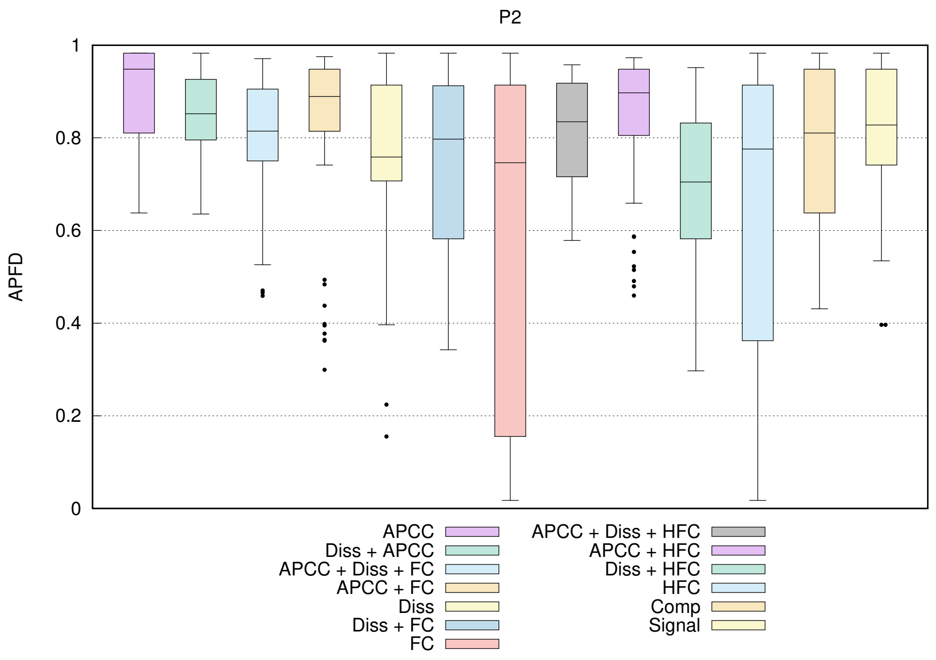


Figure B.12.: Boxplot for NSGAII Product Variant P2

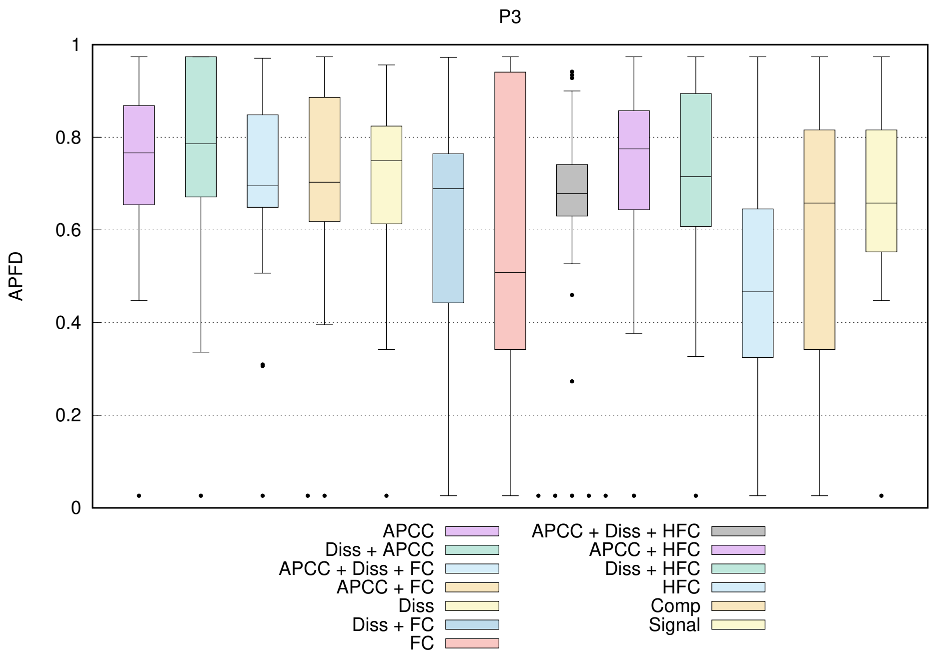


Figure B.13.: Boxplot for NSGAII Product Variant P3

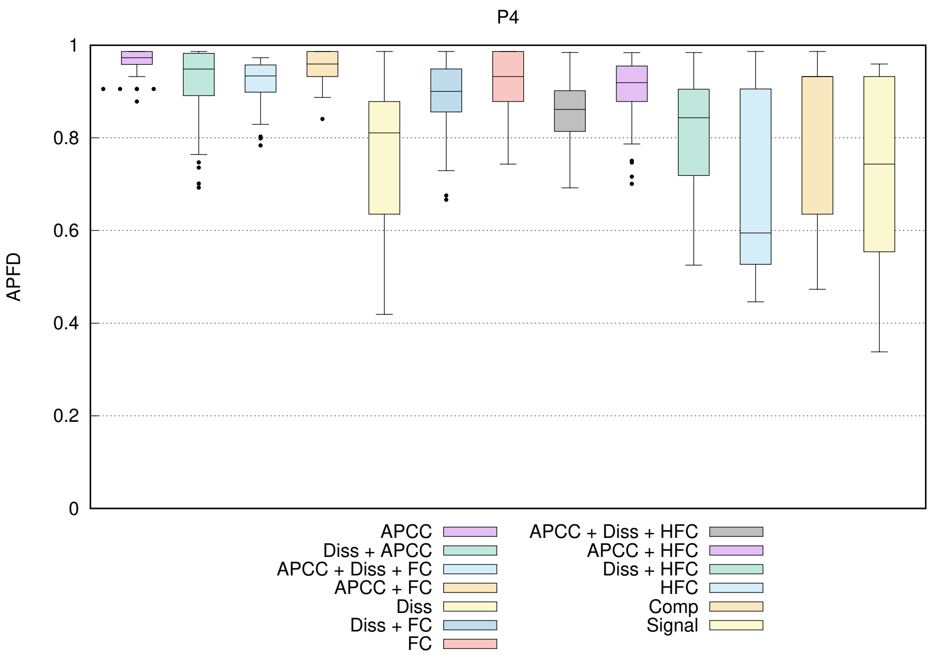


Figure B.14.: Boxplot for NSGAII Product Variant P4

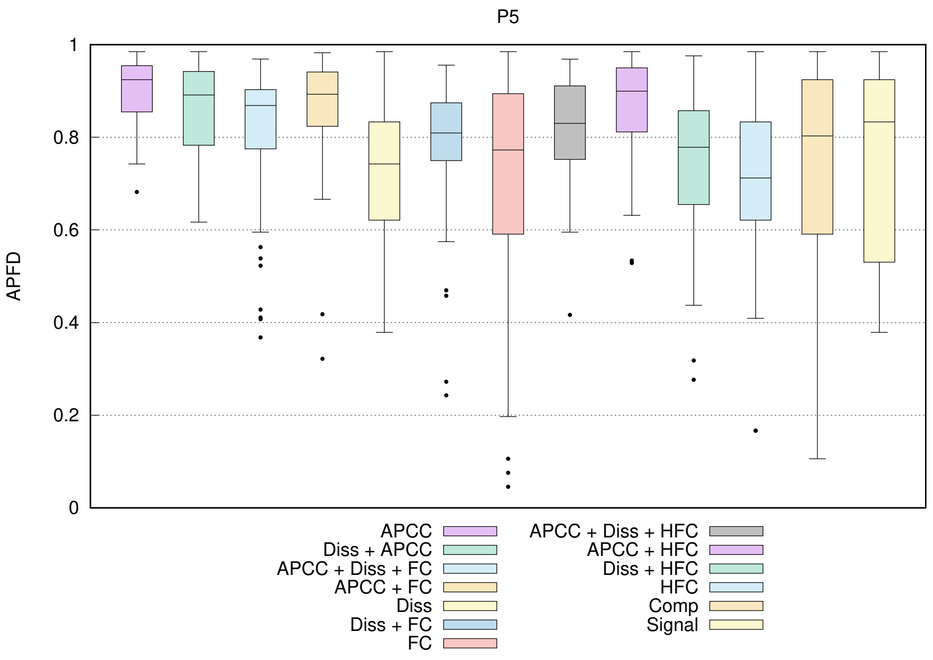


Figure B.15.: Boxplot for NSGAII Product Variant P5

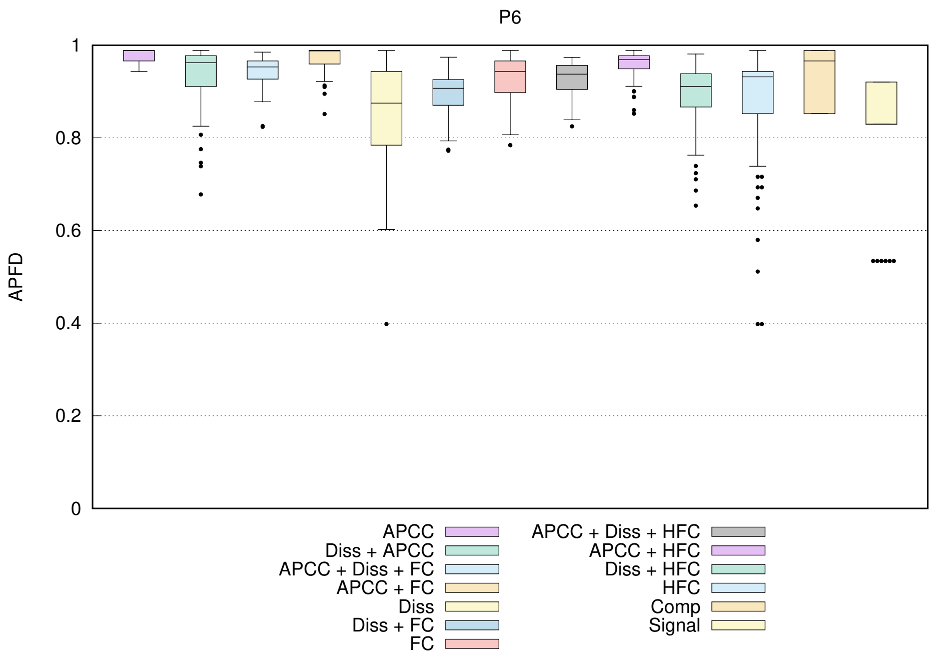


Figure B.16.: Boxplot for NSGAII Product Variant P6

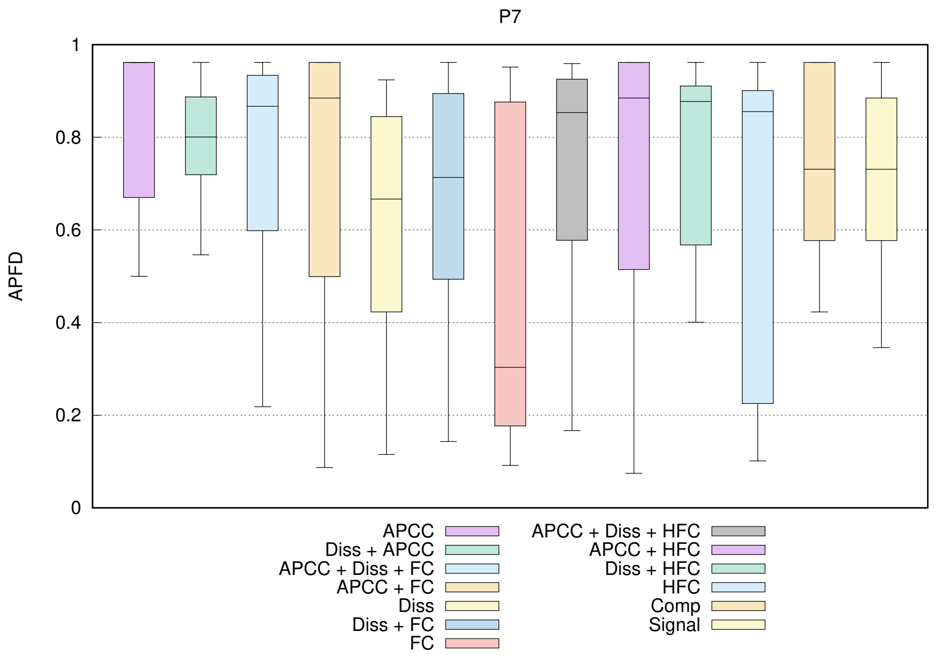


Figure B.17.: Boxplot for NSGAII Product Variant P7

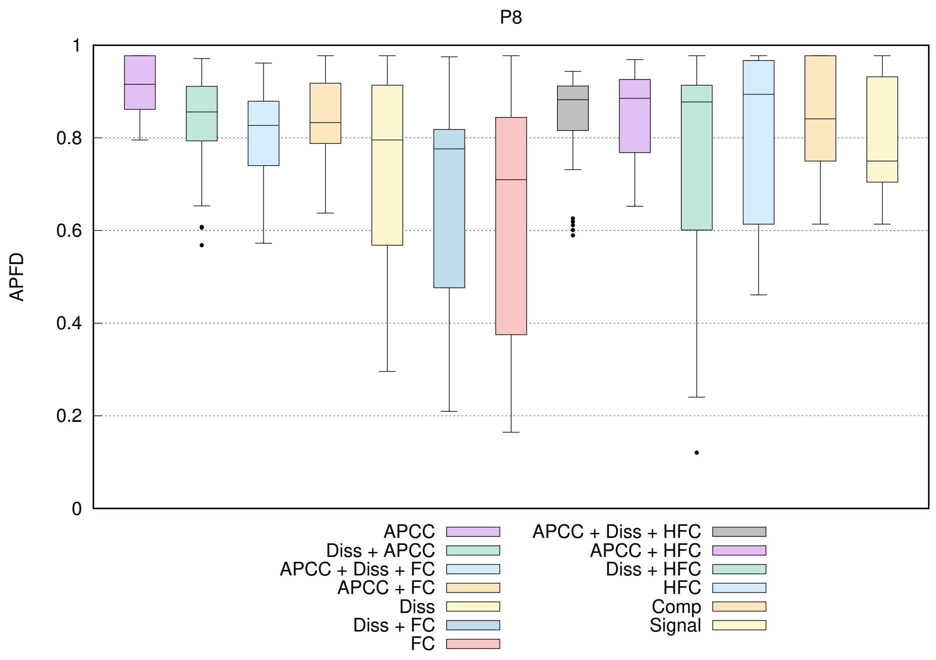


Figure B.18.: Boxplot for NSGAII Product Variant P8

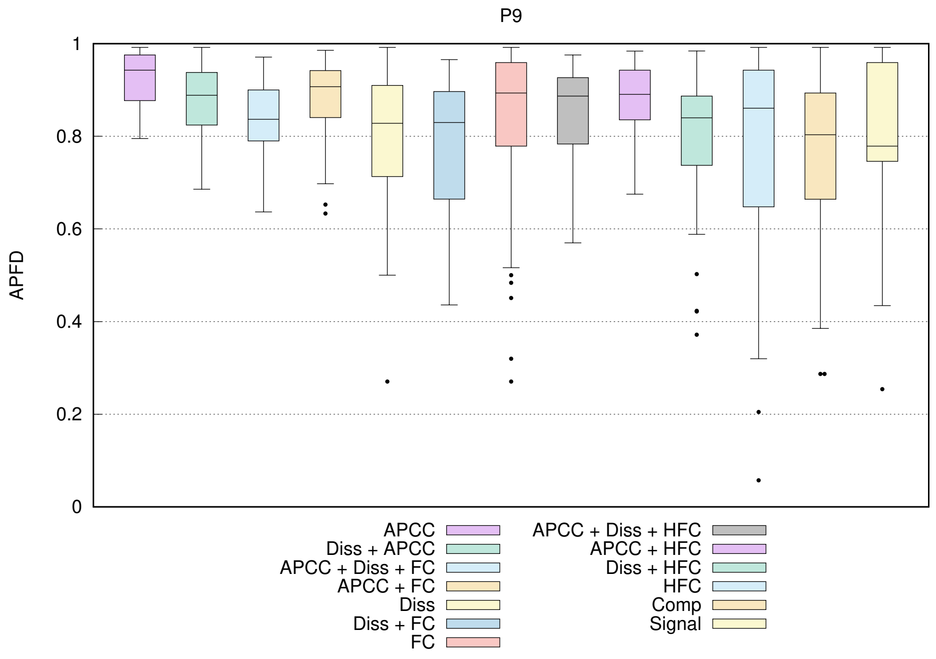


Figure B.19.: Boxplot for NSGAII Product Variant P9

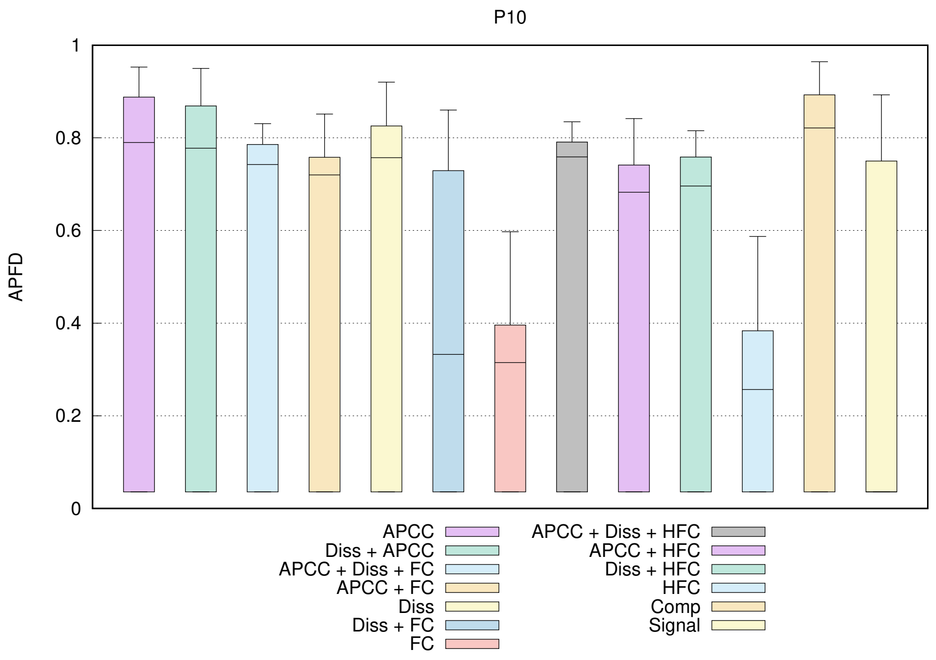


Figure B.20.: Boxplot for NSGAII Product Variant P10

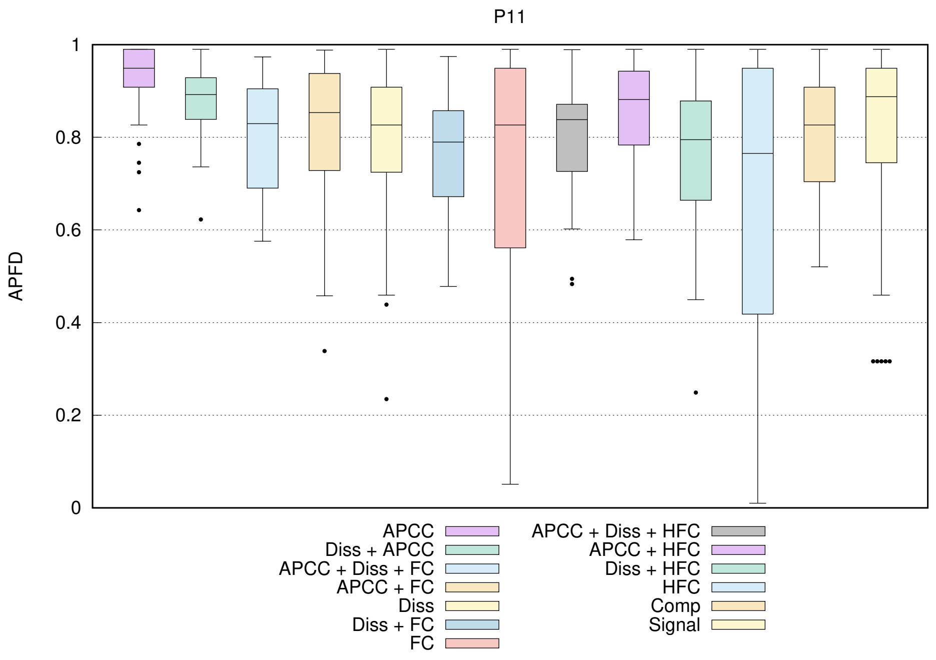


Figure B.21.: Boxplot for NSGAII Product Variant P11

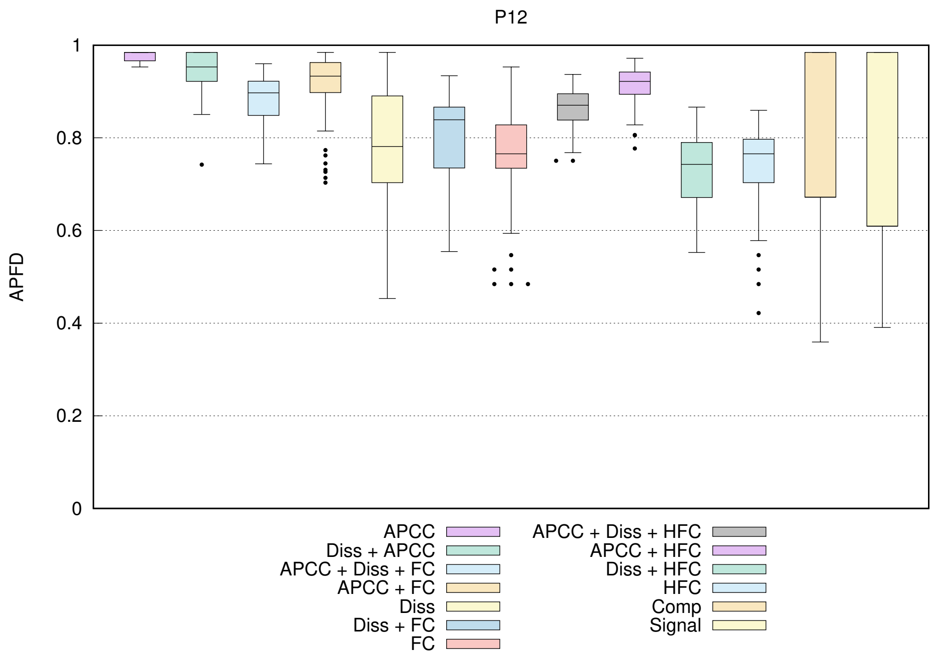


Figure B.22.: Boxplot for NSGAII Product Variant P12

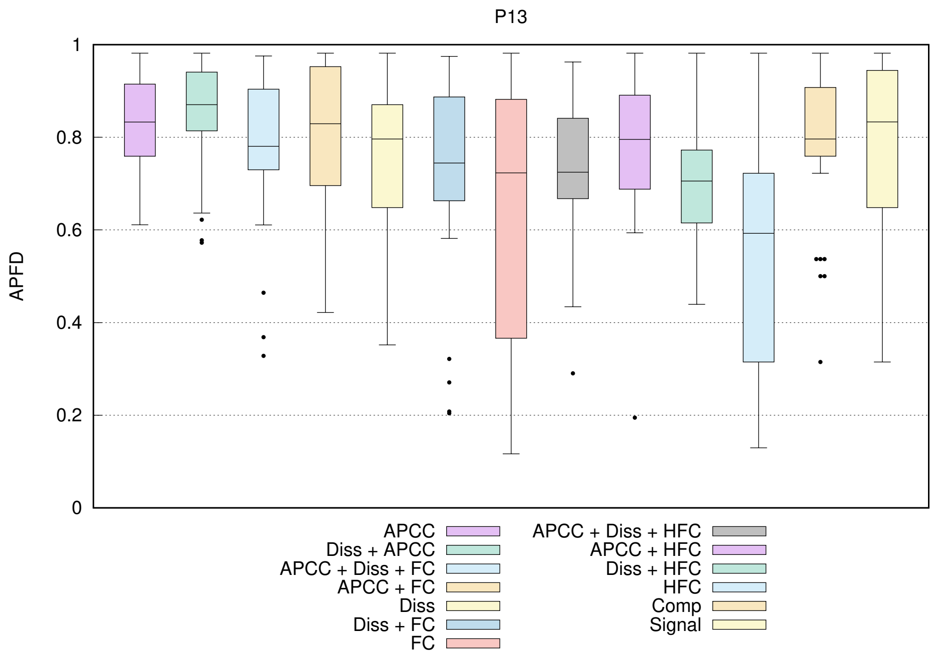


Figure B.23.: Boxplot for NSGAII Product Variant P13

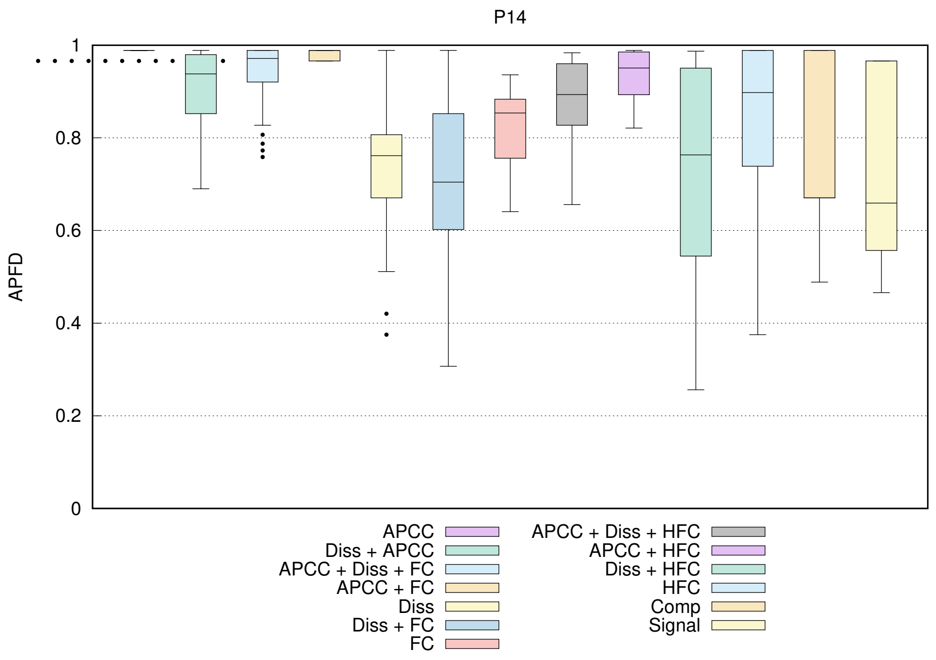


Figure B.24.: Boxplot for NSGAII Product Variant P14

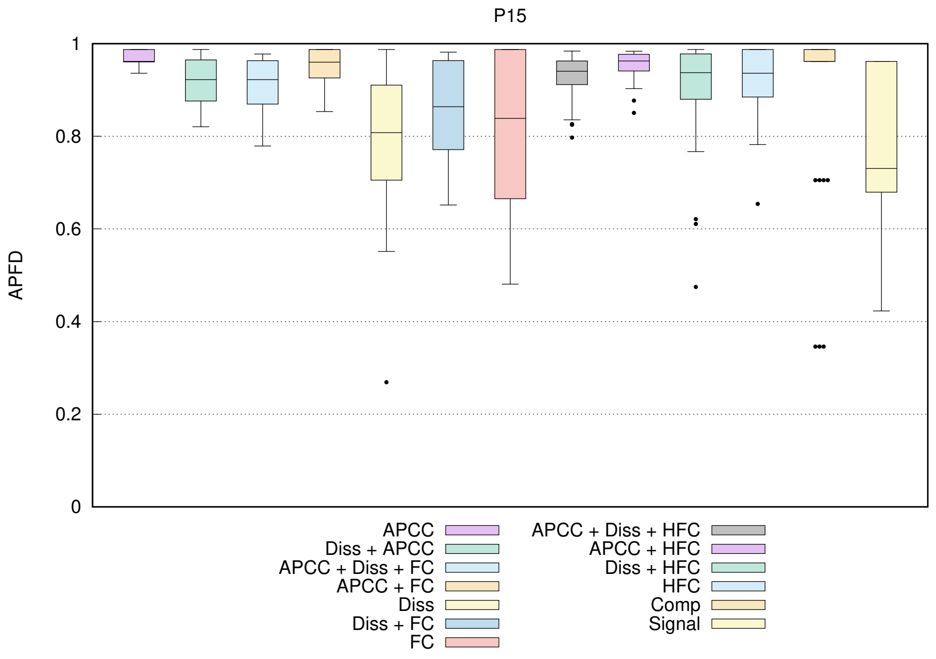


Figure B.25.: Boxplot for NSGAII Product Variant P15

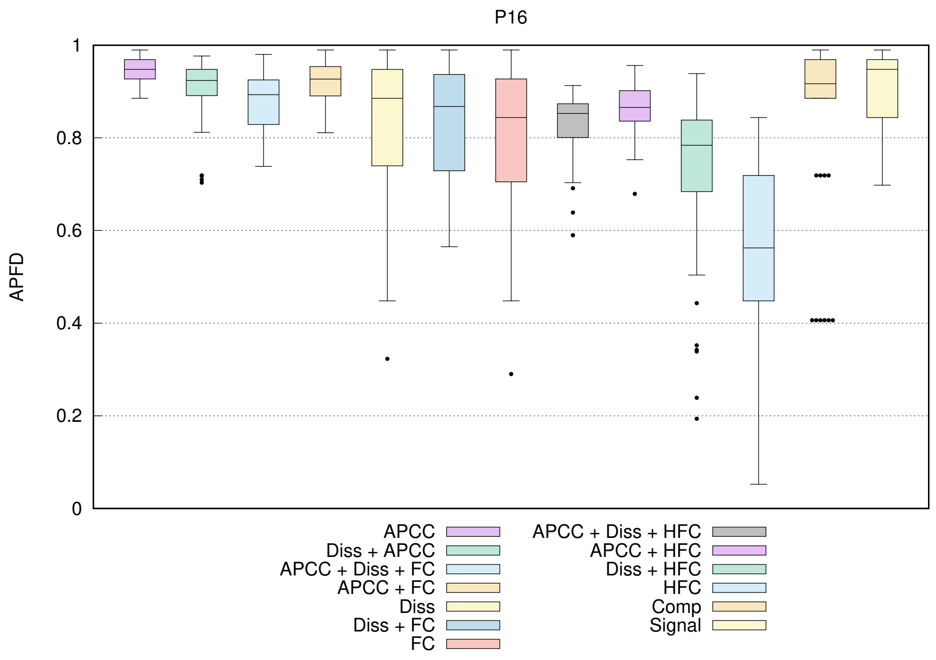


Figure B.26.: Boxplot for NSGAII Product Variant P16

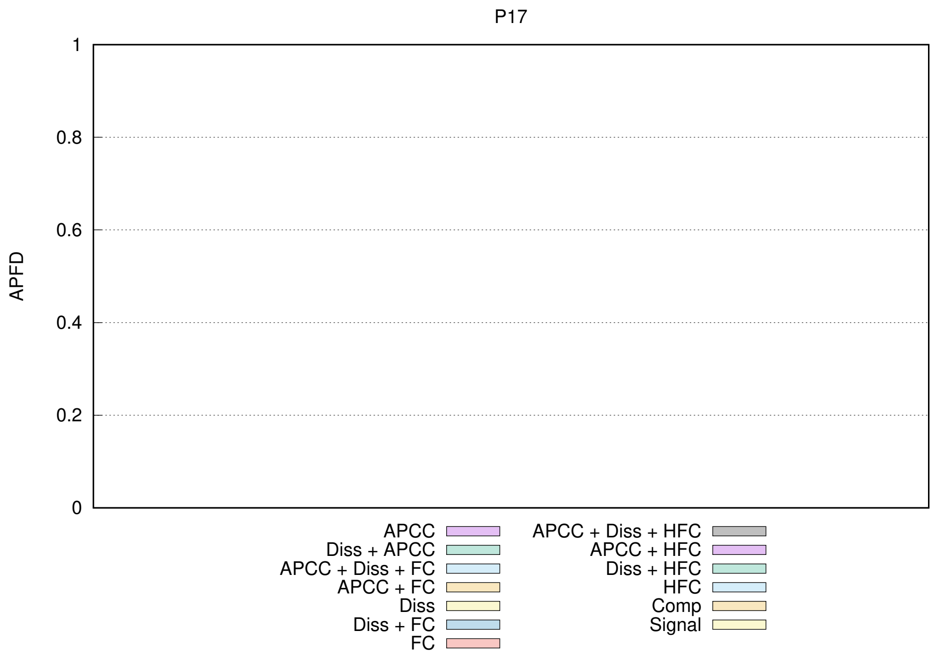


Figure B.27.: Boxplot for NSGAII Product Variant P17

B.2. DENSEA

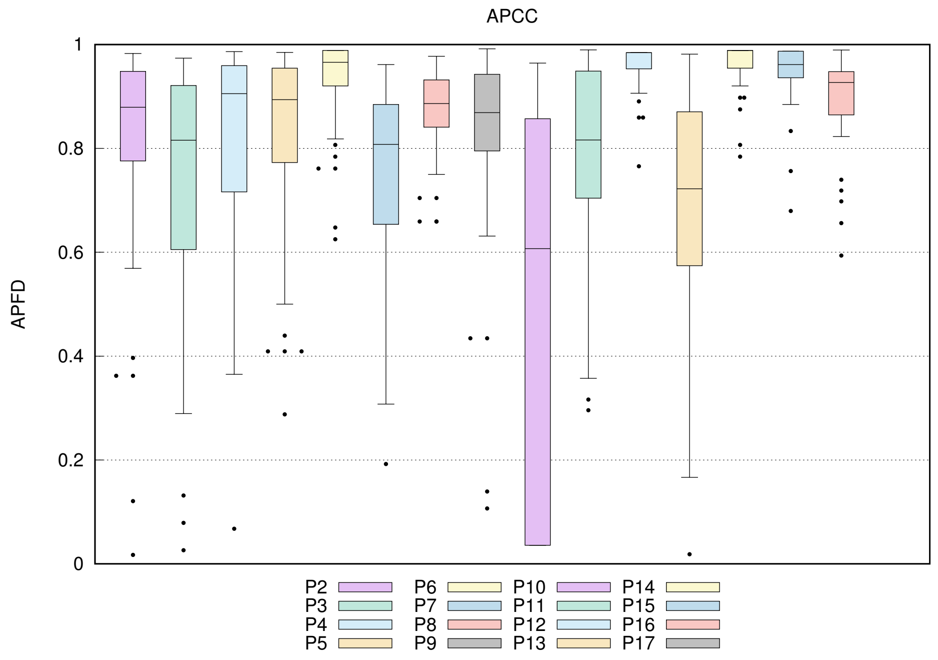


Figure B.28.: Boxplot for DENSEA Objective APCC

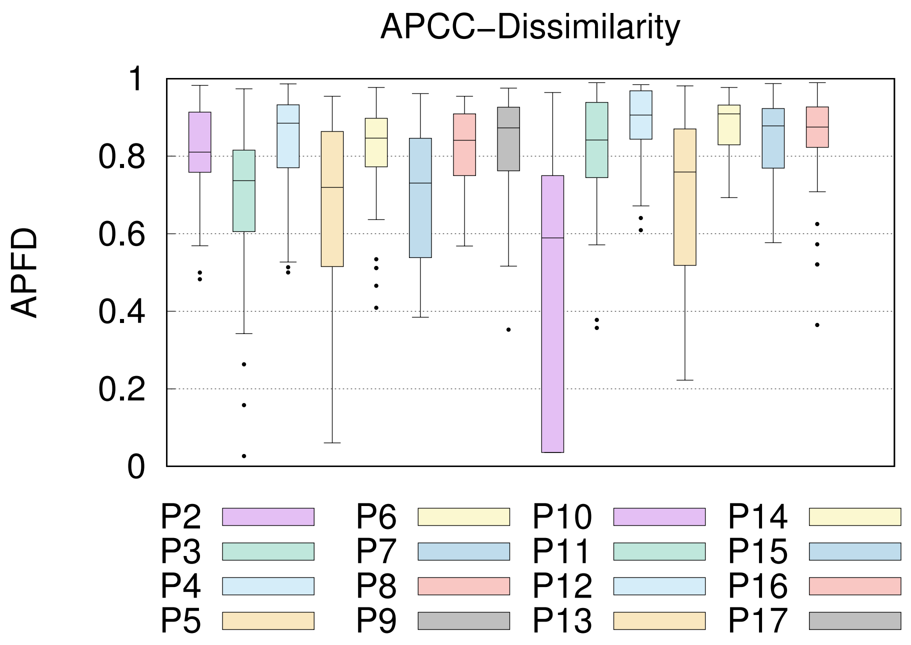


Figure B.29.: Boxplot for DENSEA Objective APCC and Diss

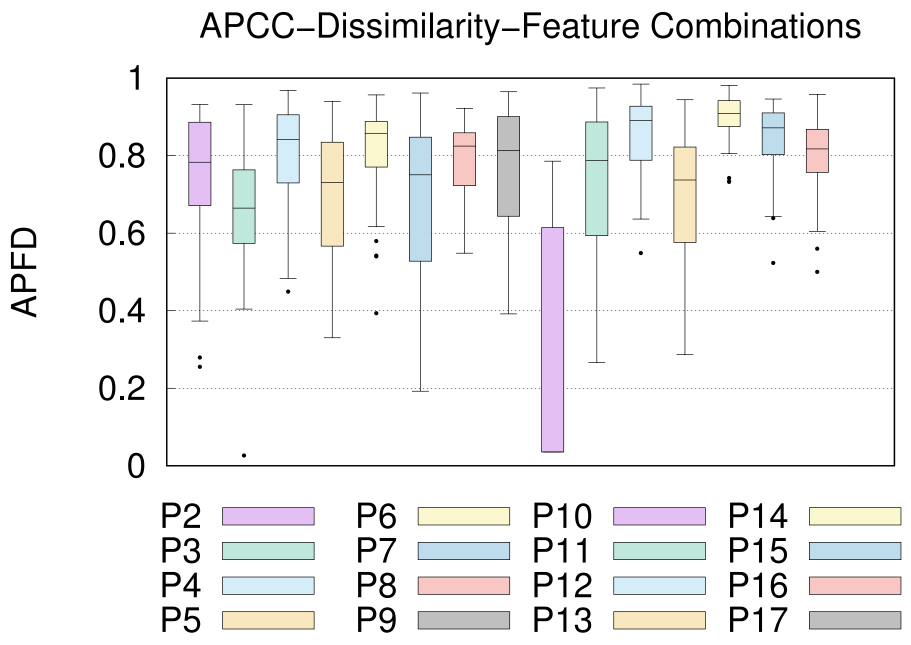


Figure B.30.: Boxplot for DENSEA Objective APCC, Diss and FC

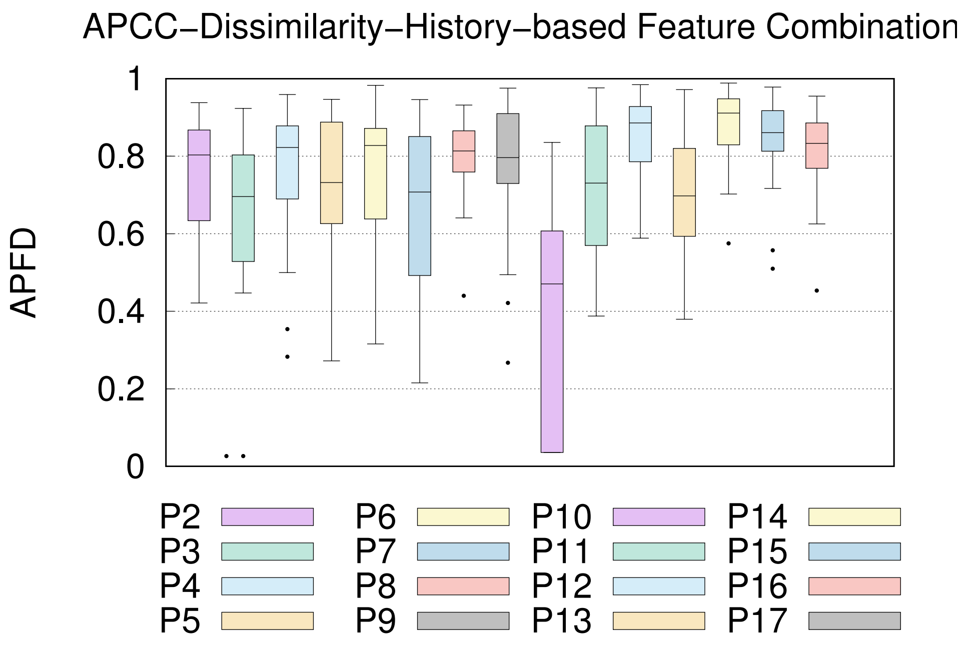


Figure B.31.: Boxplot for DENSEA Objective APCC, Diss and HFC

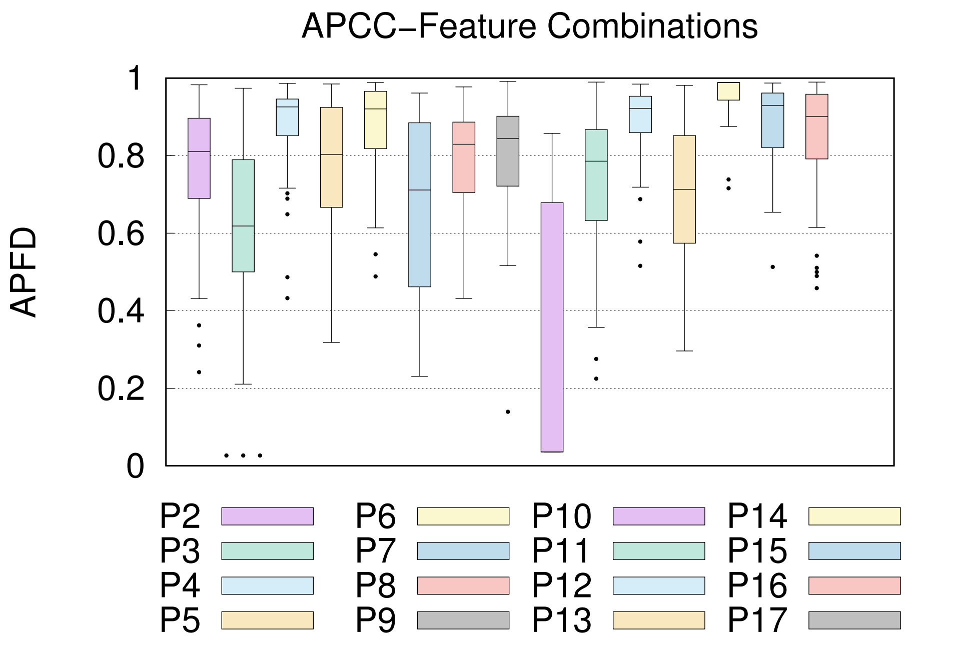


Figure B.32.: Boxplot for DENSEA Objective APCC and FC

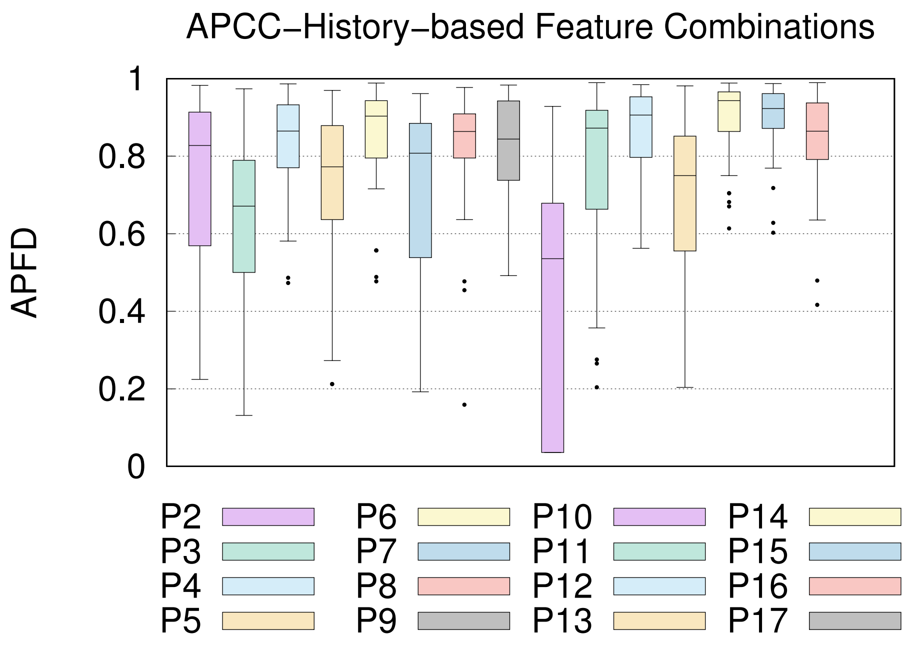


Figure B.33.: Boxplot for DENSEA Objective APCC and HFC

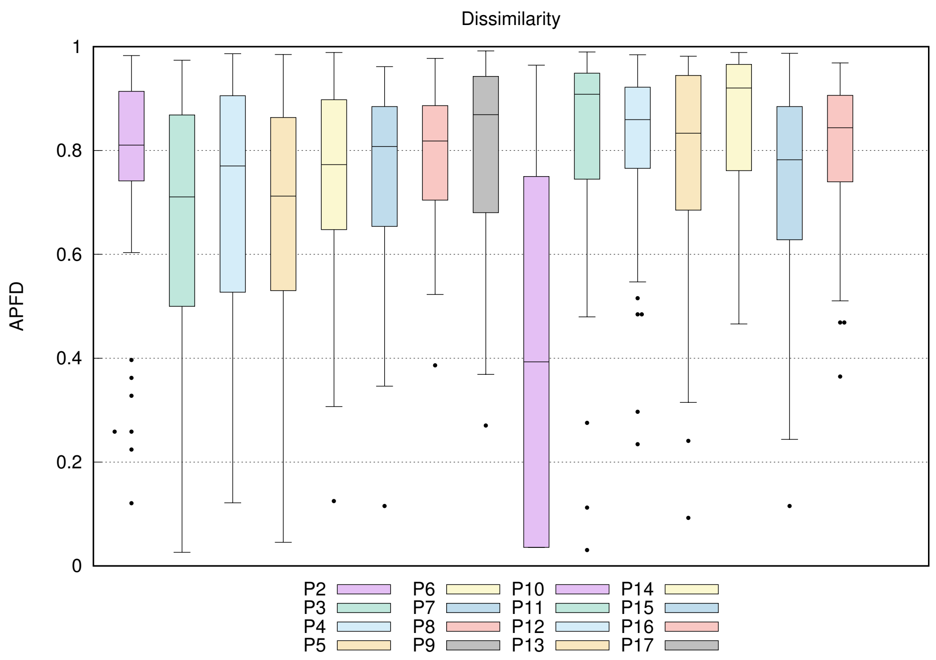


Figure B.34.: Boxplot for DENSEA Objective Diss

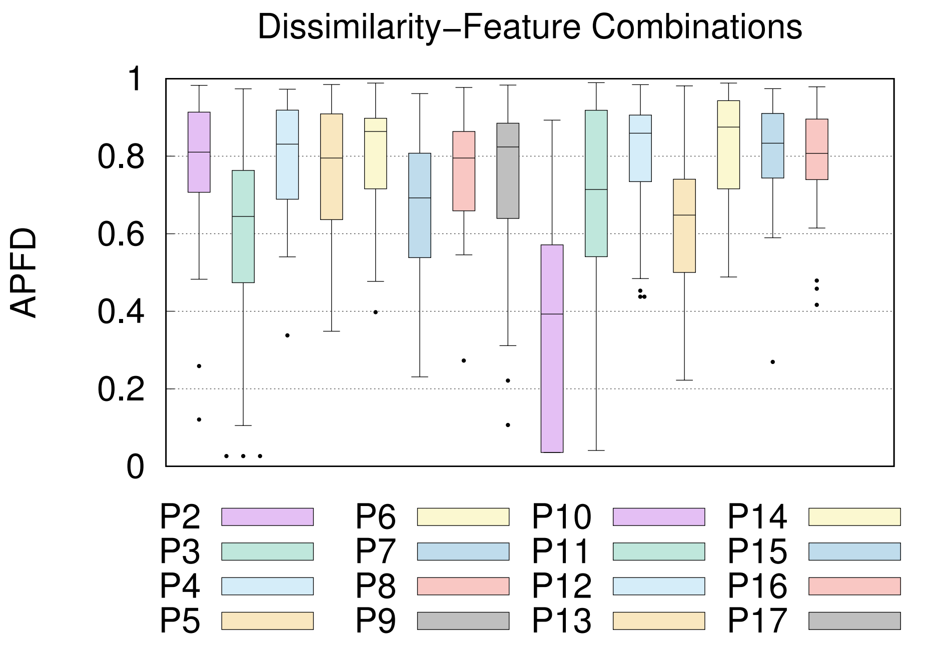


Figure B.35.: Boxplot for DENSEA Objective Diss and FC

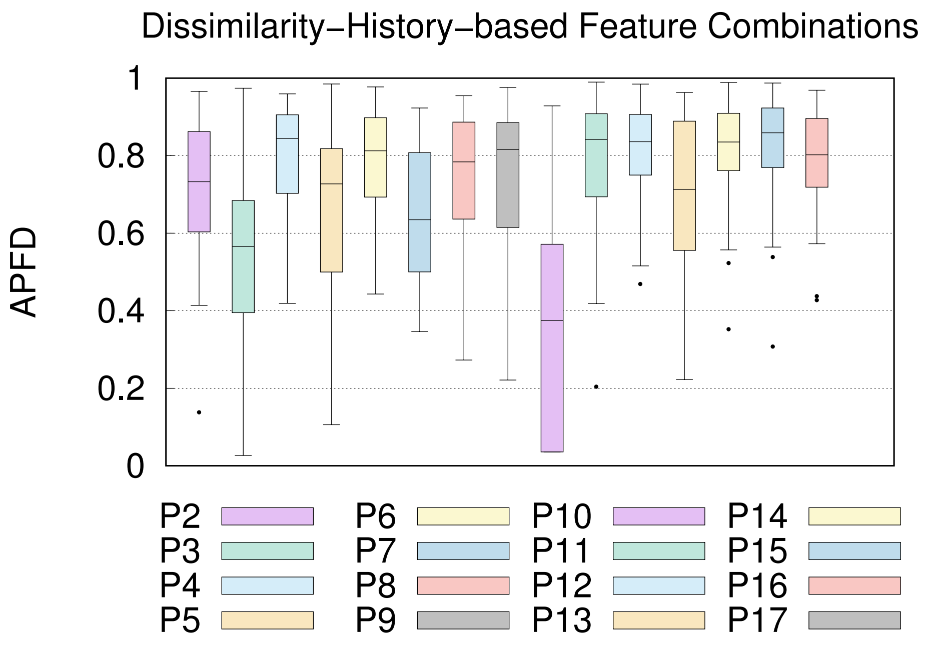


Figure B.36.: Boxplot for DENSEA Objective Diss and HFC

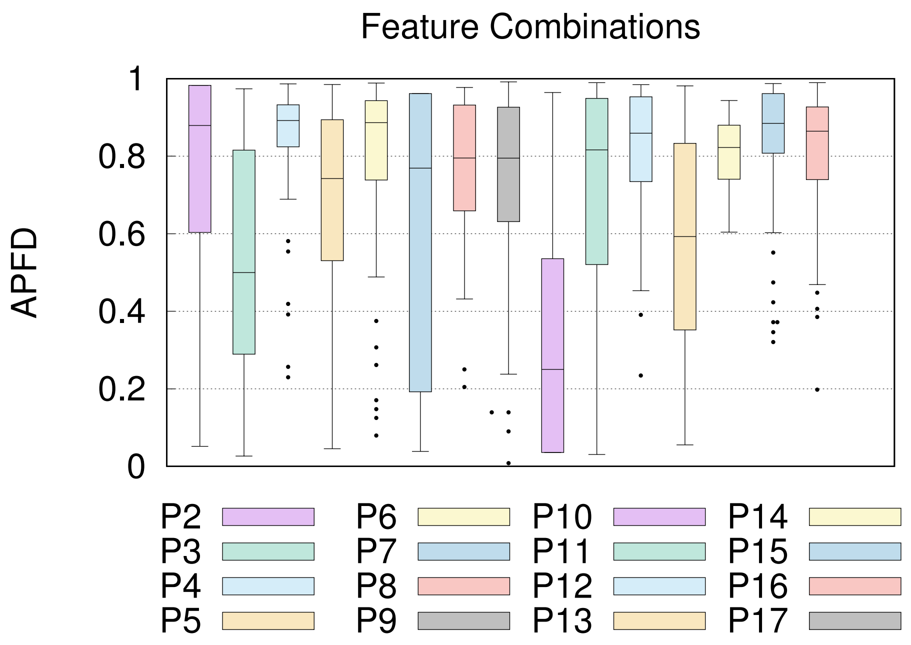


Figure B.37.: Boxplot for DENSEA Objective FC

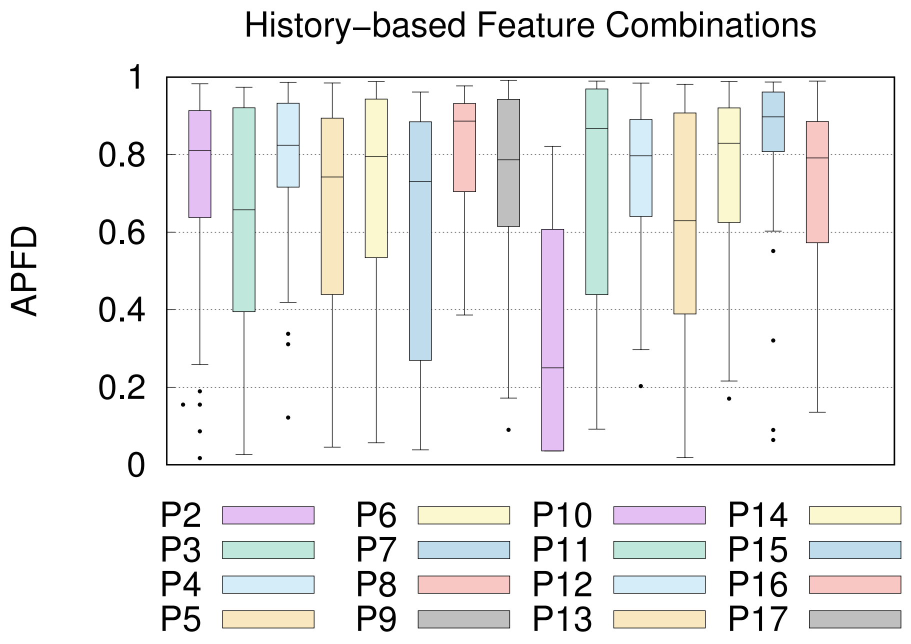


Figure B.38.: Boxplot for DENSEA Objective HFC

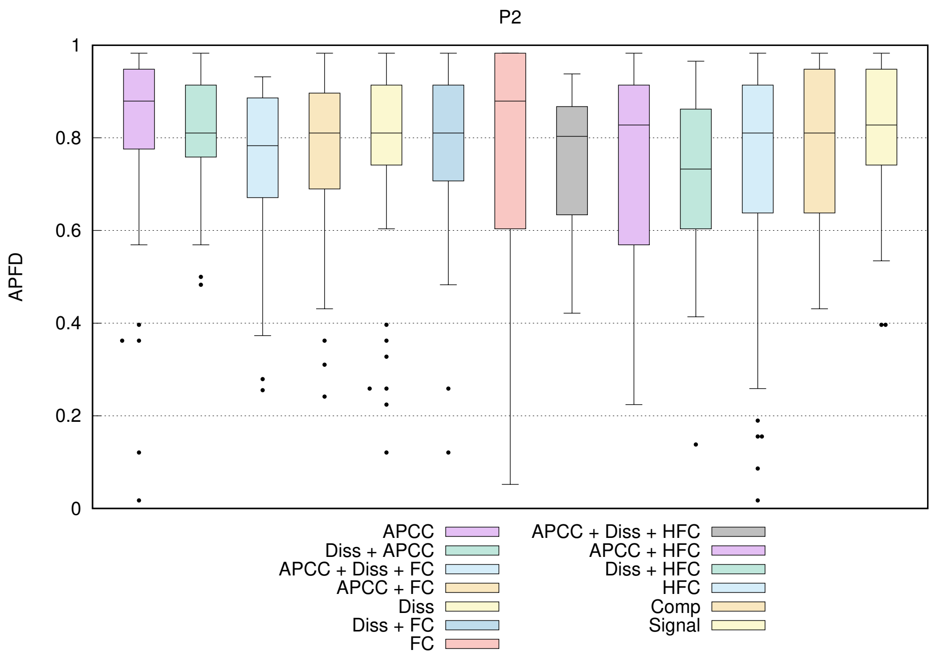


Figure B.39.: Boxplot for DENSEA Product Variant P2

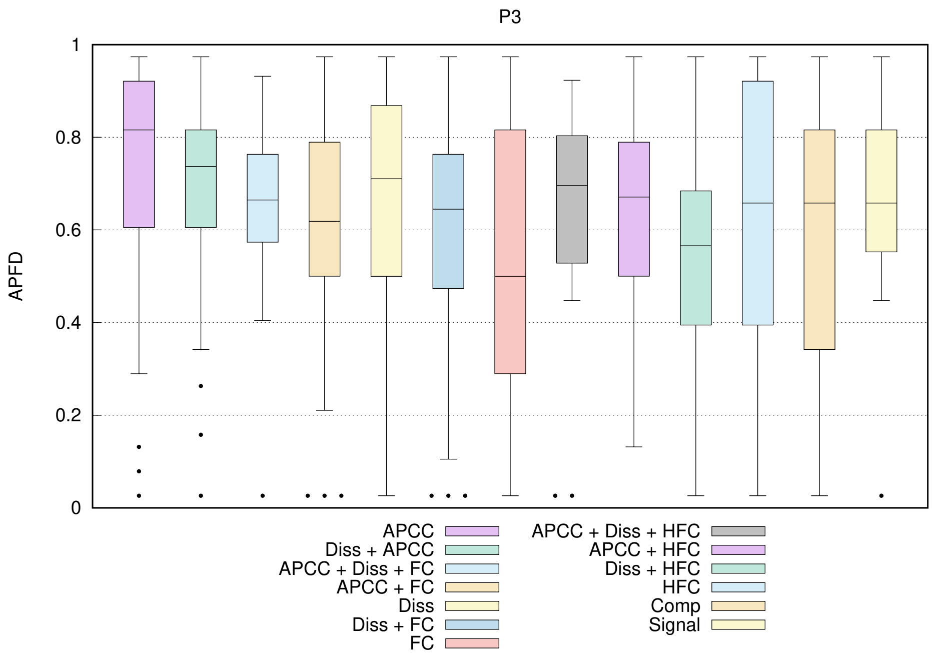


Figure B.40.: Boxplot for DENSEA Product Variant P3

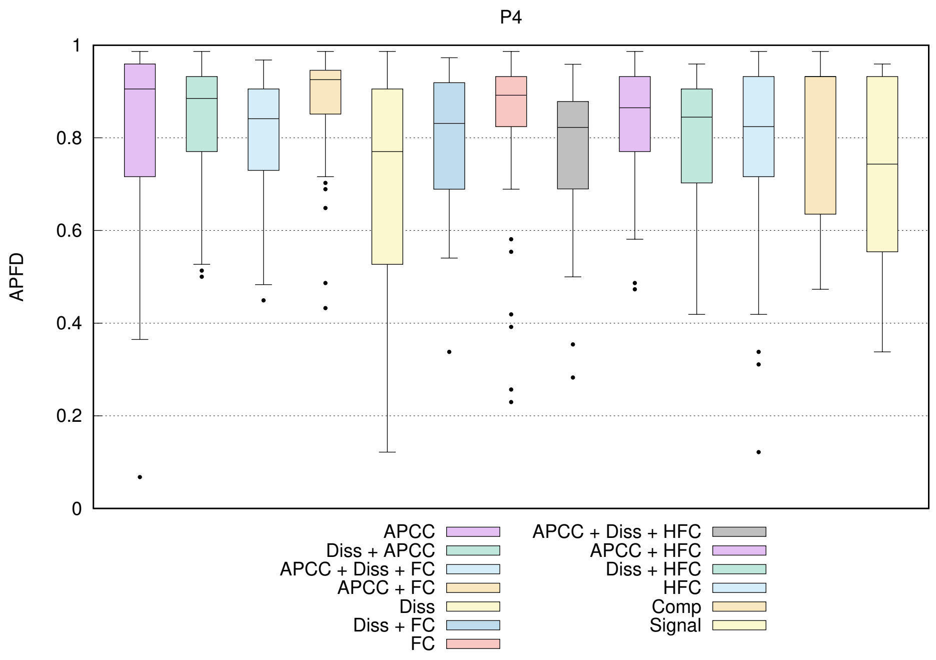


Figure B.41.: Boxplot for DENSEA Product Variant P4

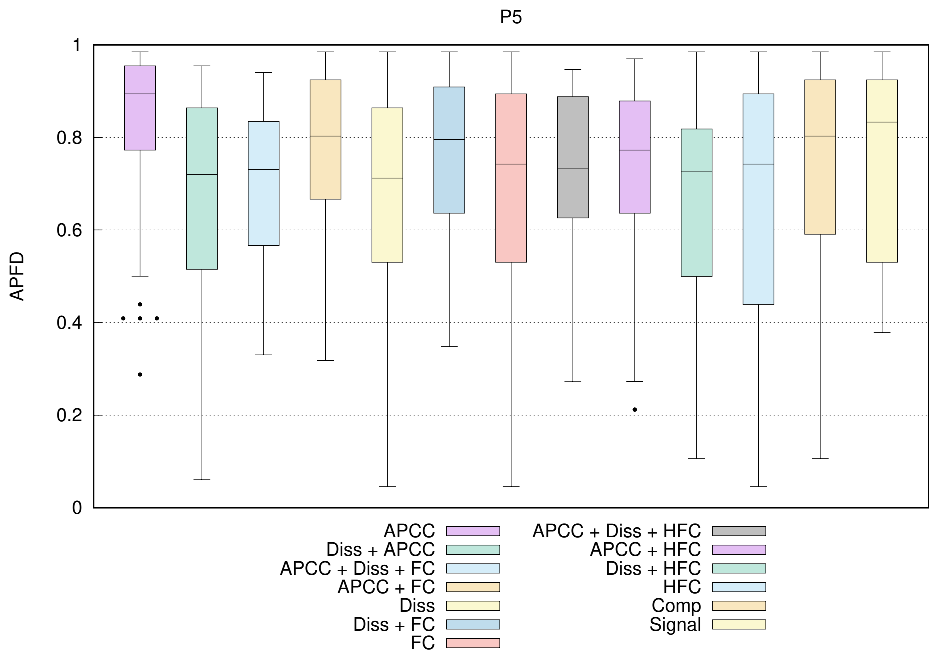


Figure B.42.: Boxplot for DENSEA Product Variant P5

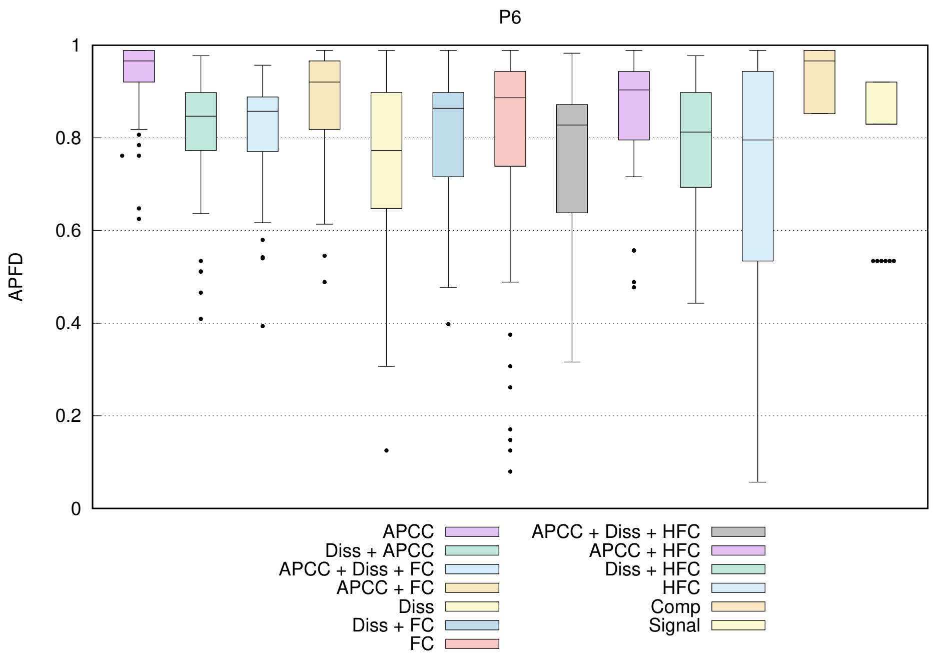


Figure B.43.: Boxplot for DENSEA Product Variant P6

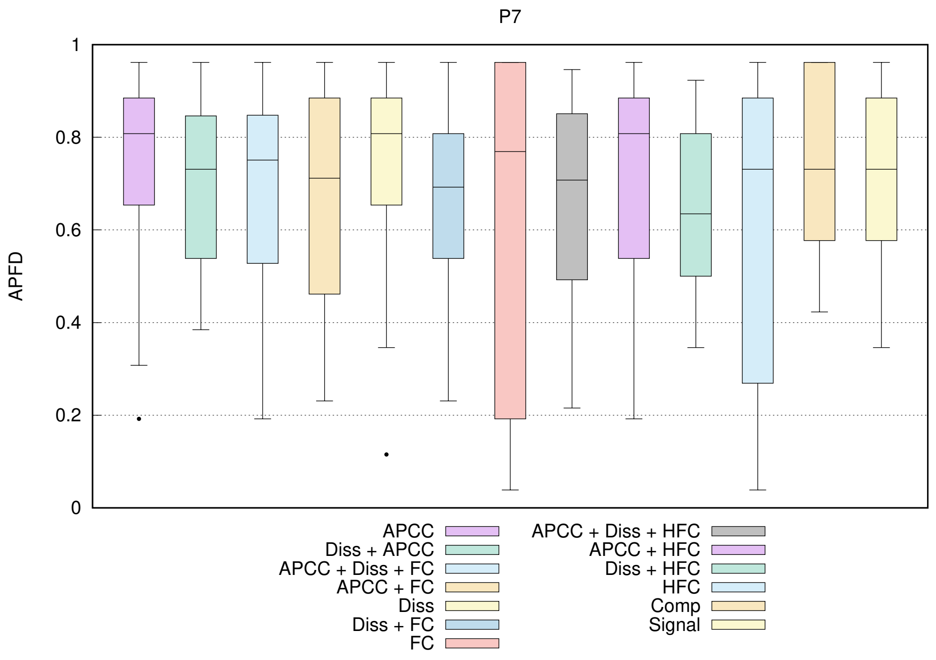


Figure B.44.: Boxplot for DENSEA Product Variant P7

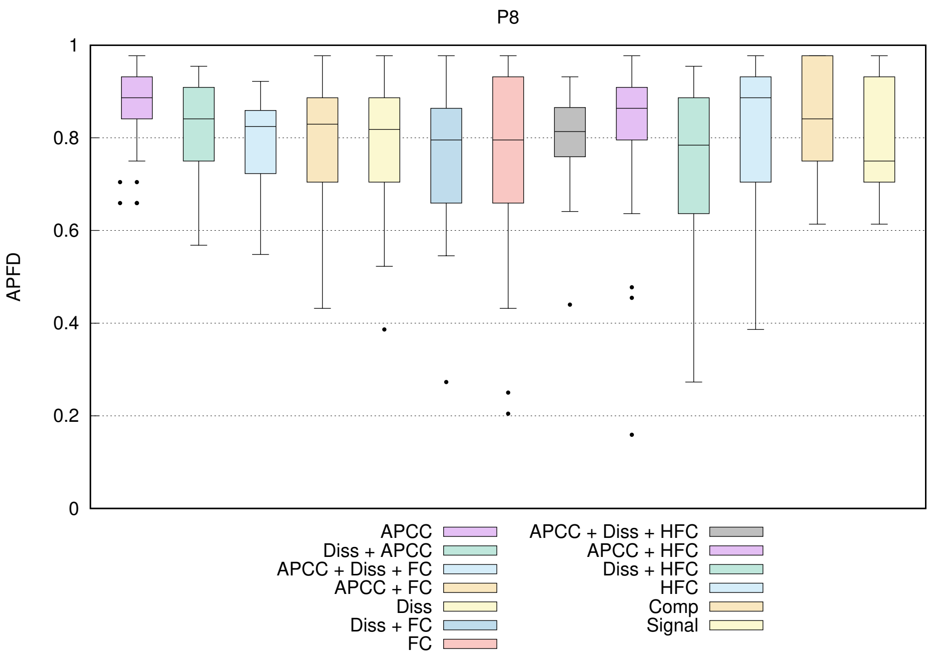


Figure B.45.: Boxplot for DENSEA Product Variant P8

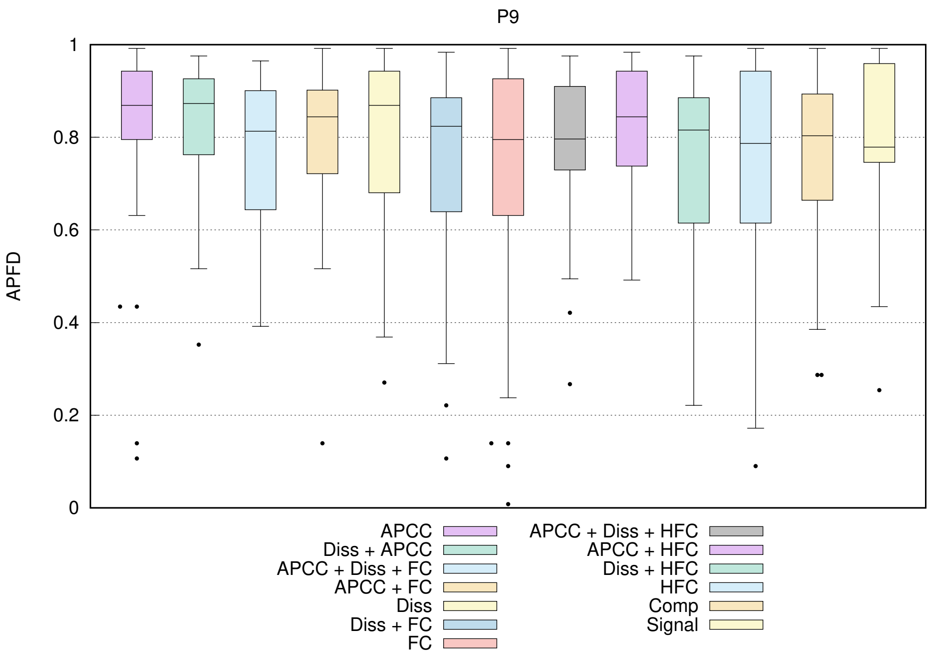


Figure B.46.: Boxplot for DENSEA Product Variant P9

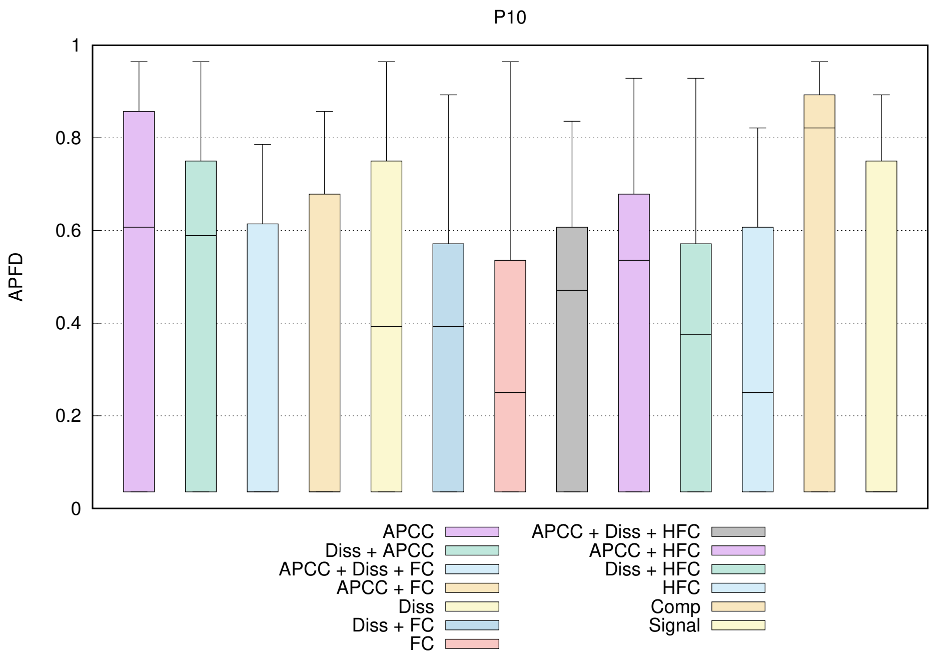


Figure B.47.: Boxplot for DENSEA Product Variant P10

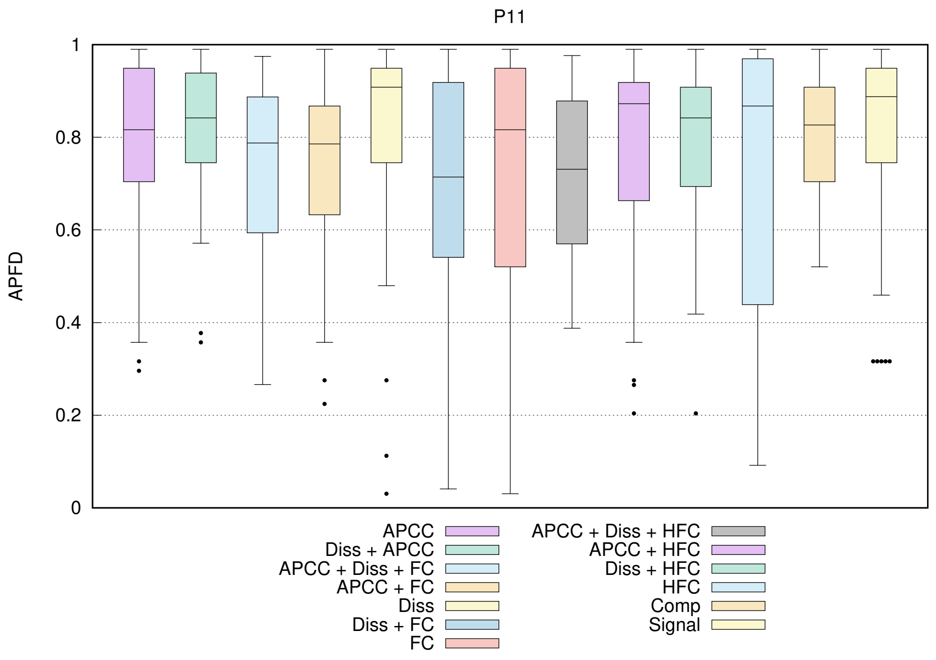


Figure B.48.: Boxplot for DENSEA Product Variant P11

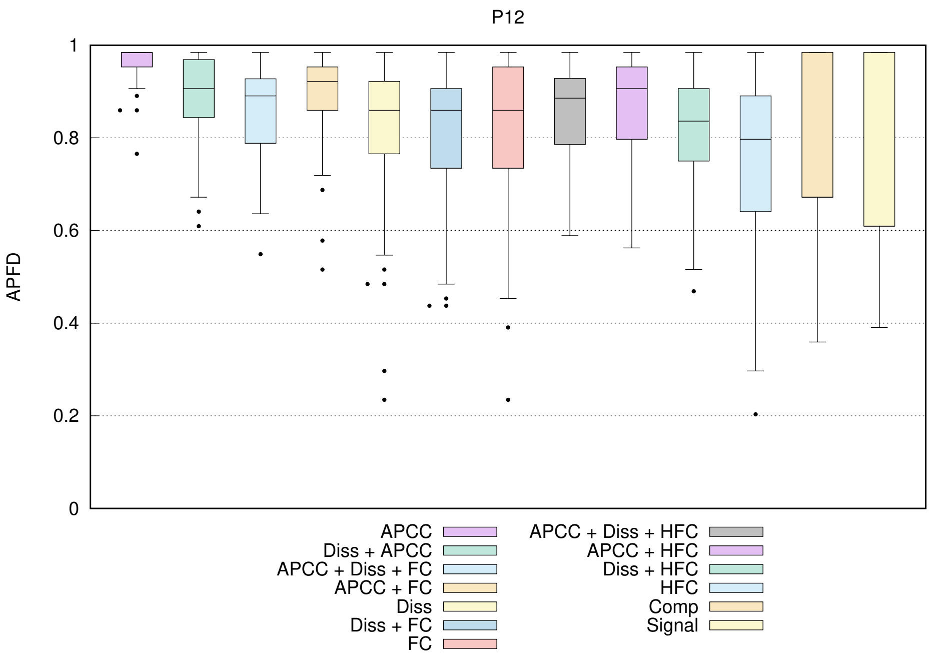


Figure B.49.: Boxplot for DENSEA Product Variant P12

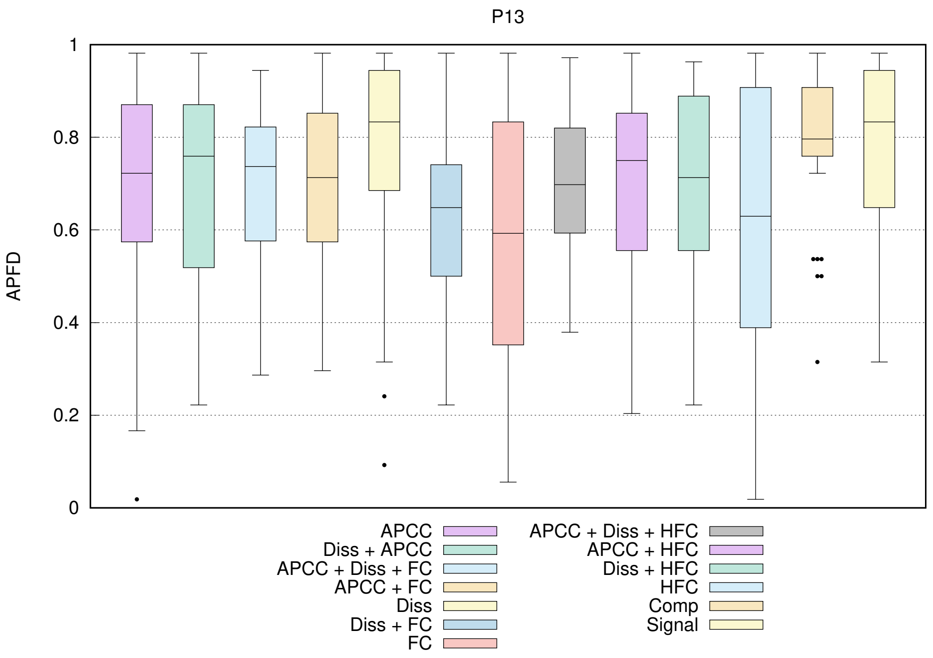


Figure B.50.: Boxplot for DENSEA Product Variant P13

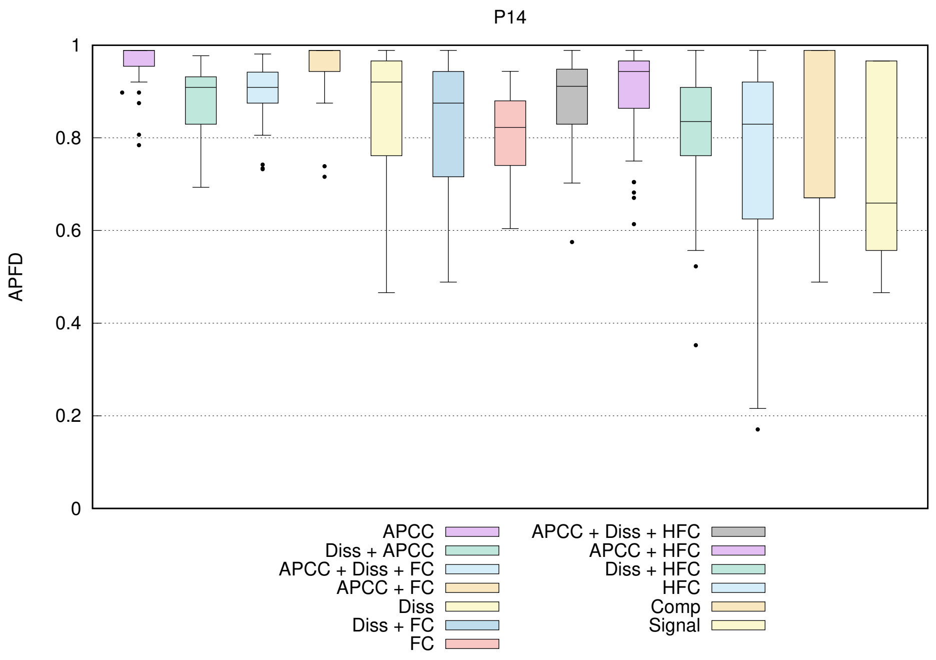


Figure B.51.: Boxplot for DENSEA Product Variant P14

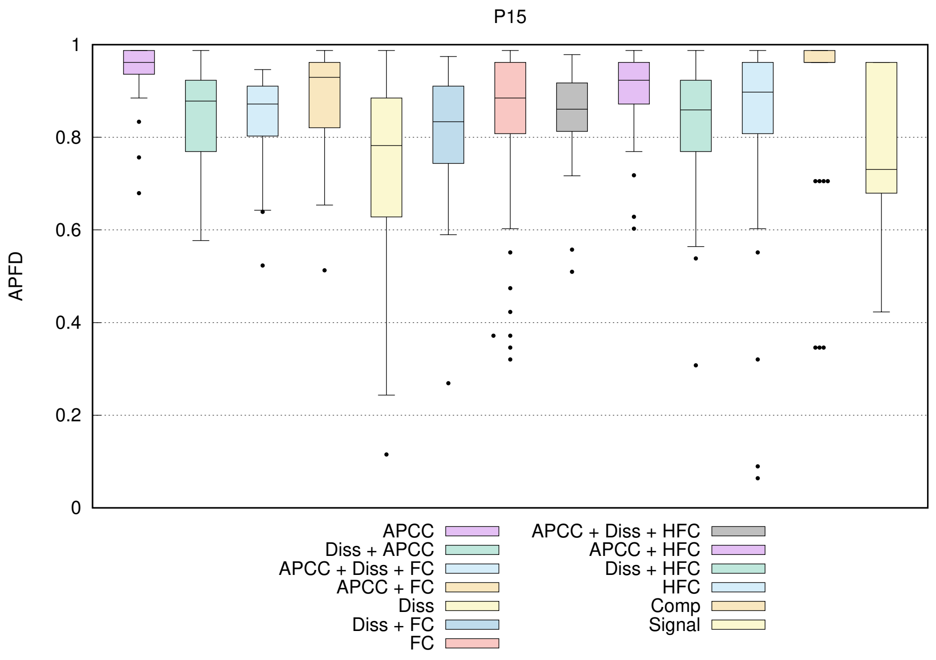


Figure B.52.: Boxplot for DENSEA Product Variant P15

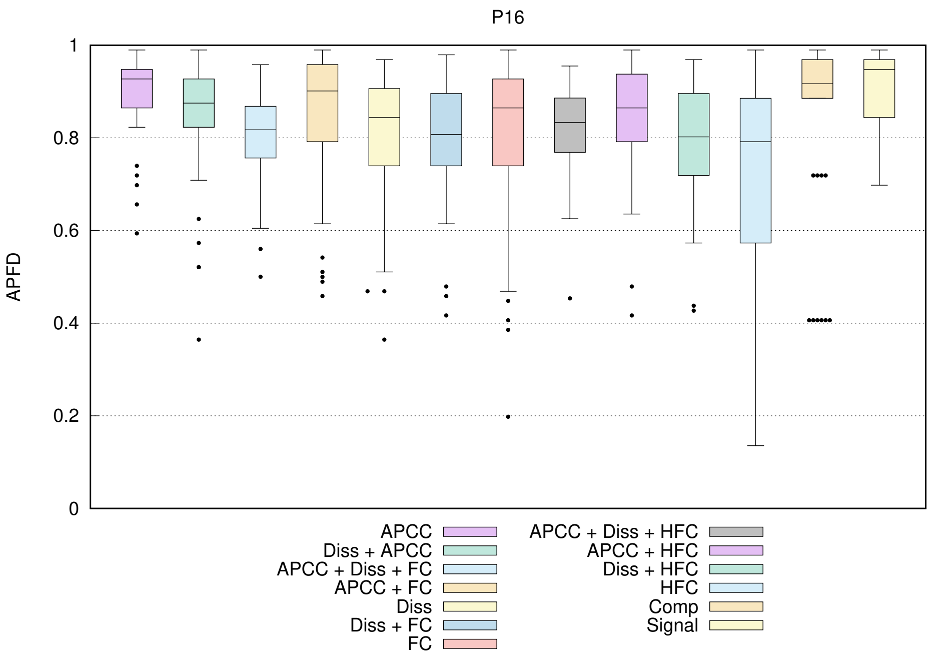


Figure B.53.: Boxplot for DENSEA Product Variant P16

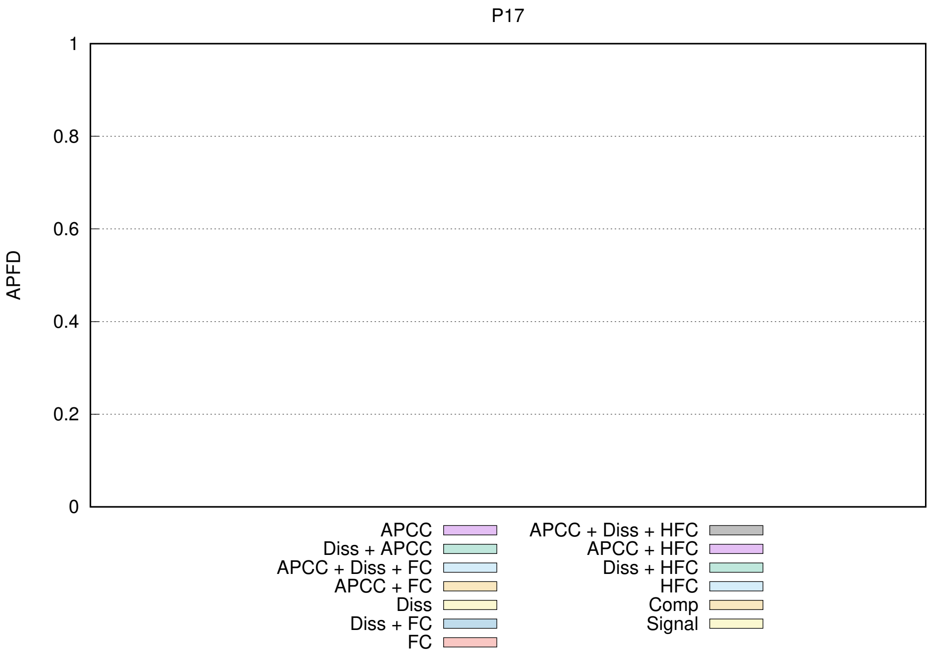


Figure B.54.: Boxplot for DENSEA Product Variant P17

B.3. Comparison of Approaches

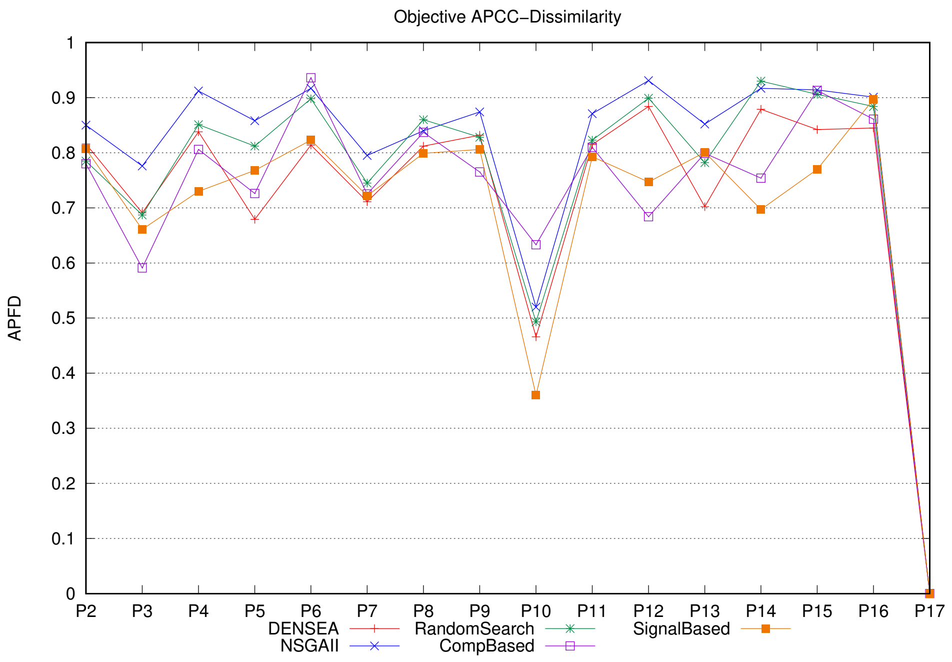


Figure B.55.: Comparison for Objective APCC and Diss and different Approaches

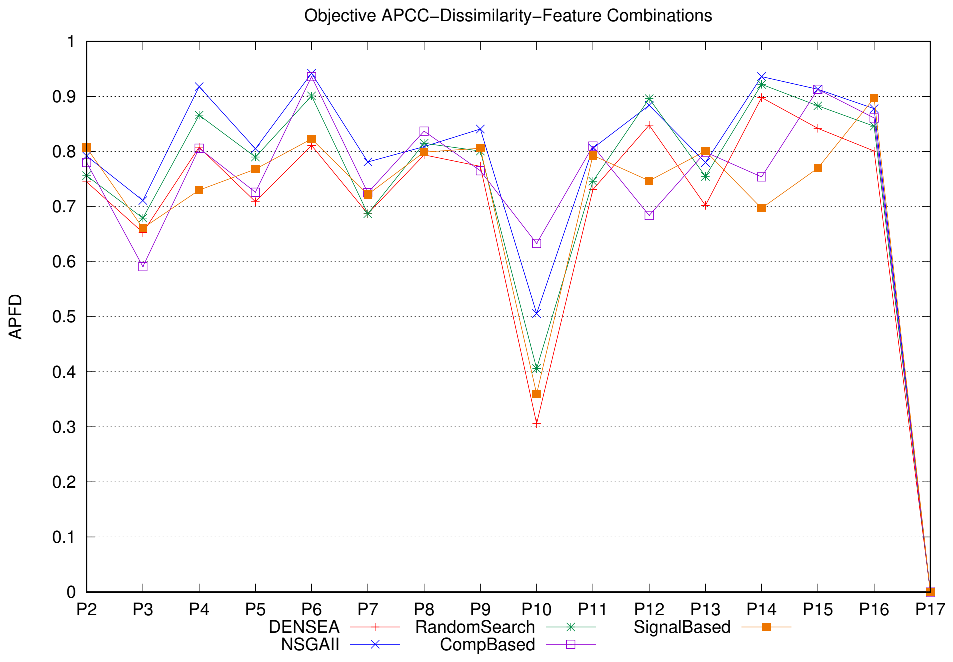


Figure B.56.: Comparison for Objective APCC, Diss and FC and different Approaches

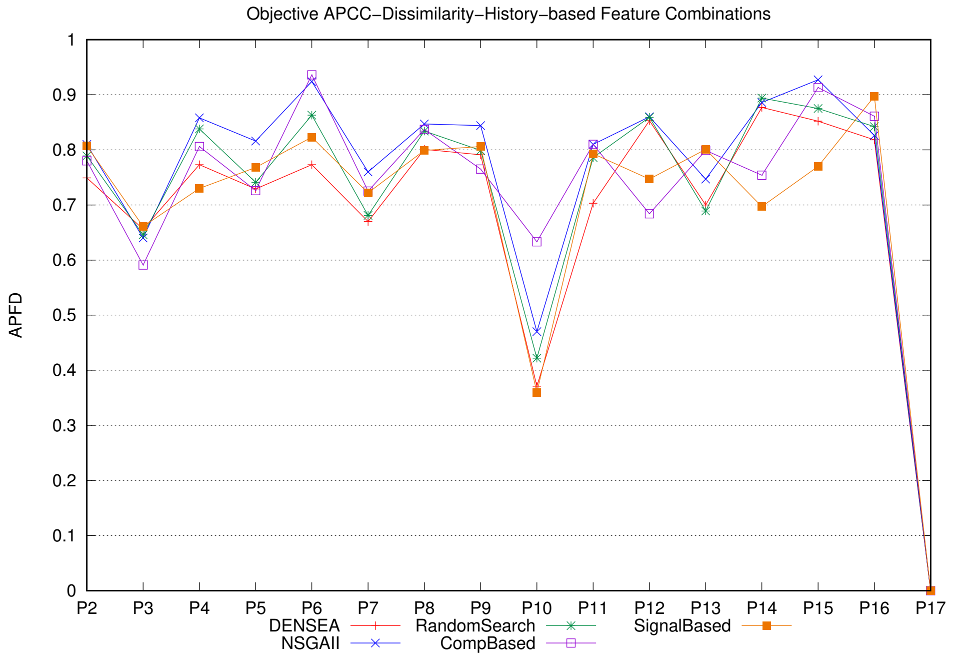


Figure B.57.: Comparison for Objective APCC, Diss and HFC and different Approaches

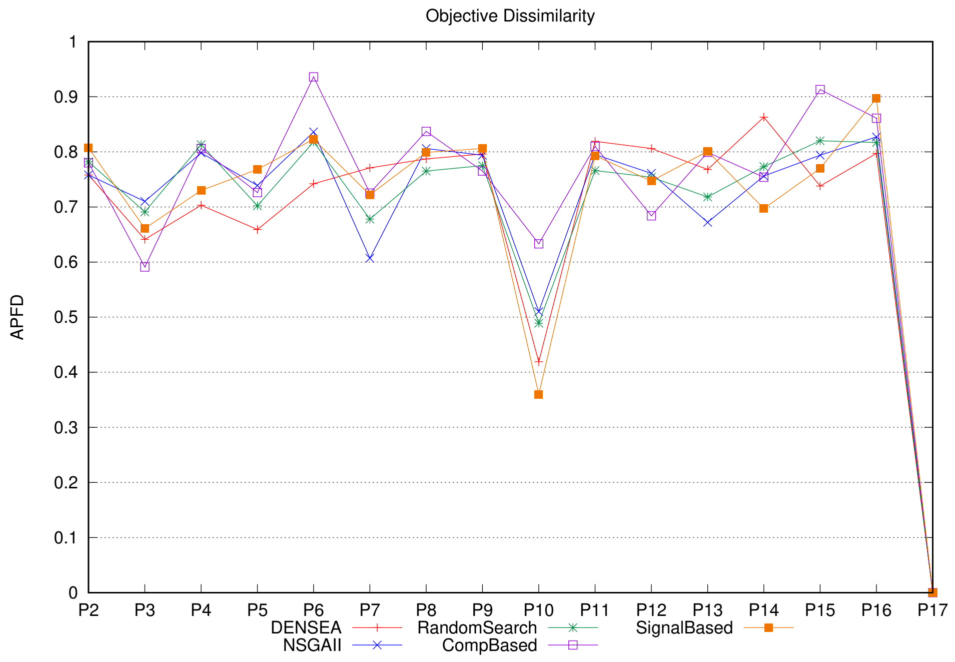


Figure B.58.: Comparison for Objective Dissimilarity and different Approaches

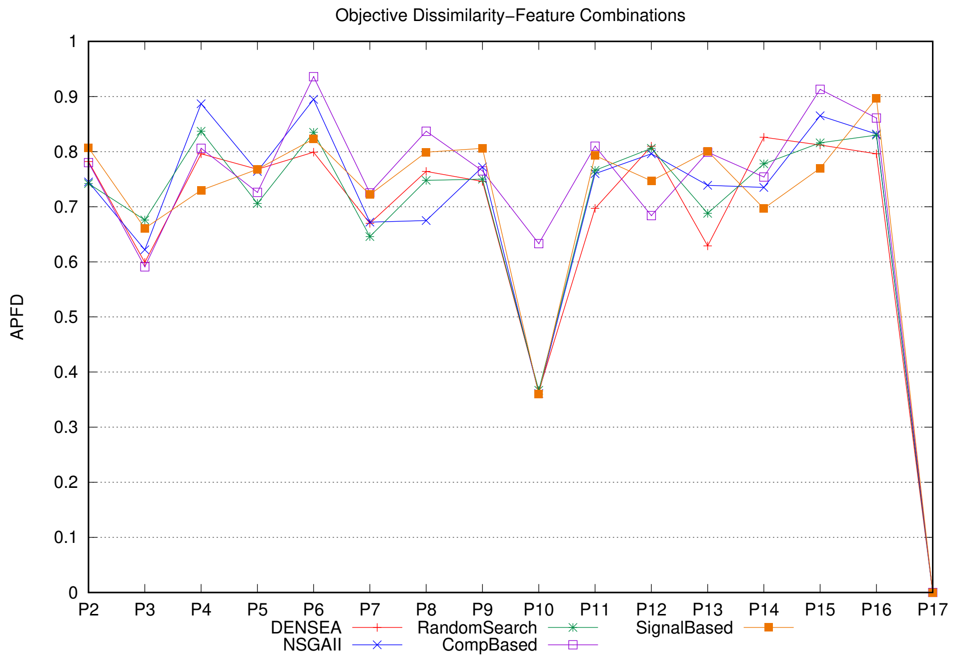


Figure B.59.: Comparison for Objective Dissimilarity and History-based Feature Combination and different Approaches

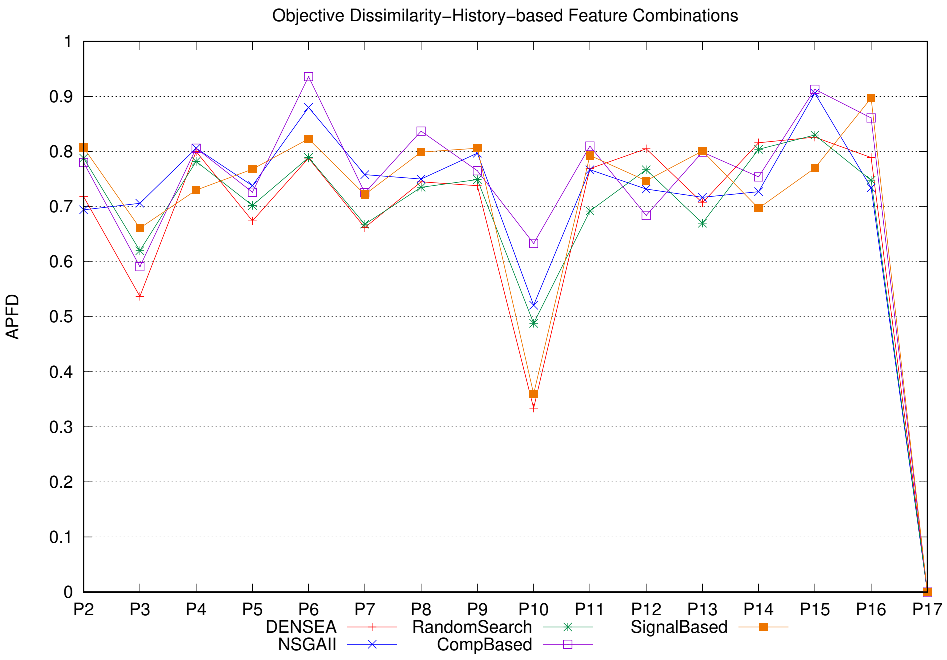


Figure B.60.: Comparison for Objective Diss and HFC and different Approaches

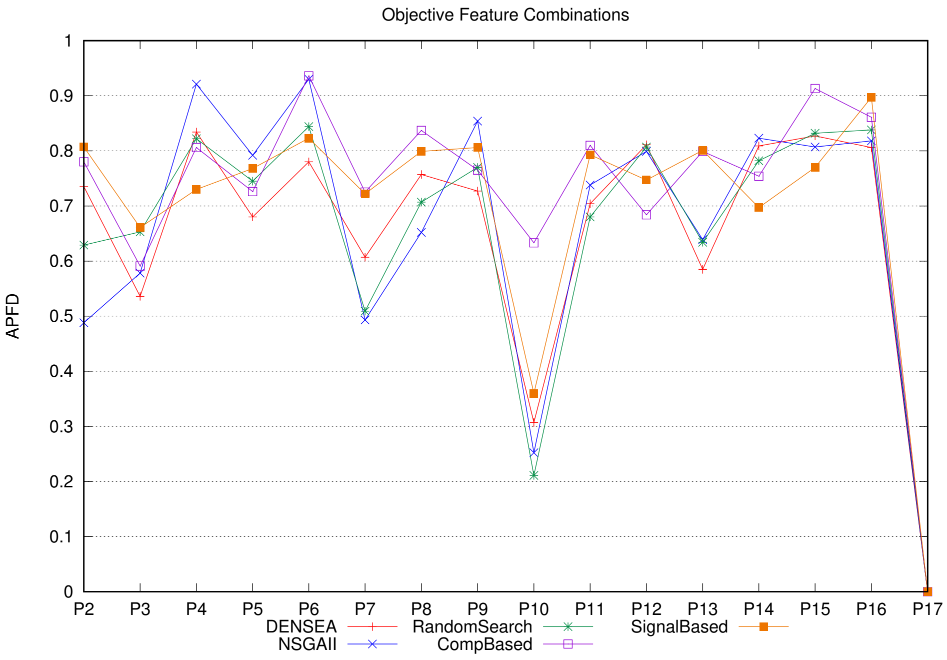


Figure B.61.: Comparison for Objective FC and different Approaches

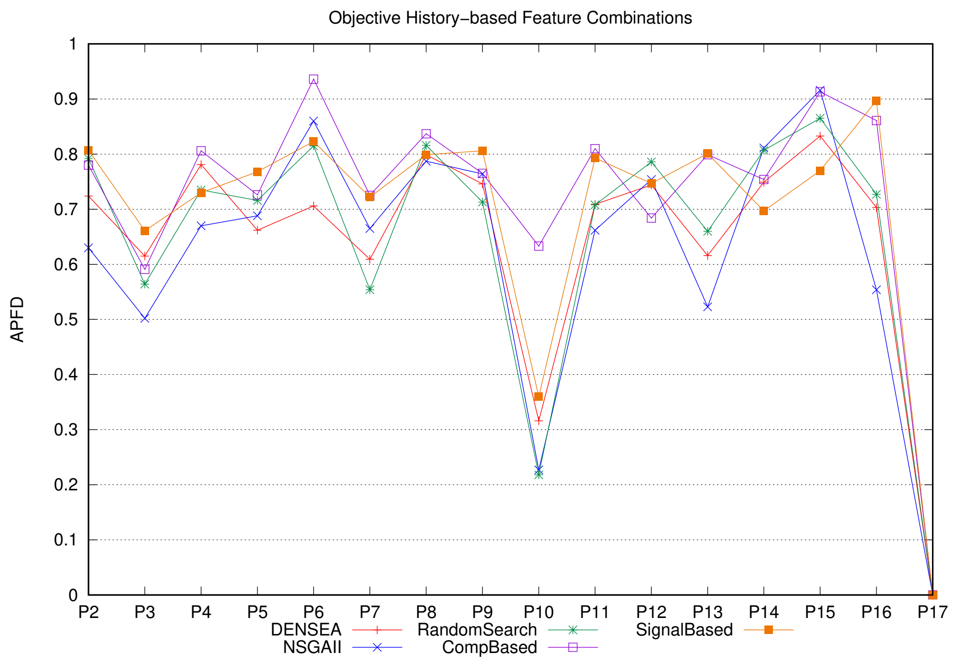


Figure B.62.: Comparison for Objective HFC and different Approaches

C Contents of the CD

```
/
├─ thesis ........................................ Thesis
├─ implementation ................................ Prototype
│  └─ BCS ........................................ Case Study Files
└─ evaluation .................................... Evaluation Results
   ├─ apfd_product_objective ..................... APFD plots
   ├─ lineplot ................................... Lineplots
   ├─ Neue_Evaluationen.tar.gz ................... Results of the evaluation
   └─ runtime_product_objective .................. Runtime plots
```