University of Passau

Department of Informatics and Mathematics

Master's Thesis

Experiments on Type Checking of Software Product Lines

Author:

Claus Hunsen

July 05, 2013

Advisors:

Dr.-Ing. Sven Apel

Software Product-Line Group

Christian Lengauer, Ph.D.

Chair for Programming

Hunsen, Claus:
Experiments on Type Checking of Software Product Lines
Master's Thesis, University of Passau, 2013.

Abstract

Type checking of software product lines is a great challenge, because they inherit variability. There are several strategies that can be pursued in order to type-check a whole software product line: product-based, feature-based or family-based as well as any mix of those strategies. We present our implementation of the bytecode-based strategy—it performs feature composition on bytecode level and bytecode verification for type-checking purposes. Advantages and drawbacks of the bytecode-based strategy are identified and potential challenges are discussed. Furthermore, we compare three strategies (product-, bytecode-, and family-based) in terms of performance. We conducted three type-checkers to a set of 12 feature-oriented product lines implemented in Java as a benchmark. We present and discuss the measurement results that clearly identify the family-based type-checking as the significantly fastest approach. The product- and bytecode-based implementations bare some critical drawbacks apart from their advantages.

Contents

1. Introduction

A *software product line (SPL)* is a family of related software products. These products have a common base and differ in terms of *features* which represent domain artifacts, implement stakeholder requirements or offer configuration options. [AKGL10] This way, the developer designs and provides a family of products but is also able to provide each customer a tailor-made product. A specific product is constructed by selecting a combination of desired features—setting the *configuration*—and composing these features afterwards to synthesize the desired product. [TBKC07] For a SPL, probably not all feature combinations are desired, so that the developer usually provides a *feature model* which describes constraints on feature combinations. This way, only configurations are allowed that are consistent with the feature model—with other words, are *valid* configurations. Moreover, the feature model is able to define a hierarchy of the features so that a pair of two features may be labeled as alternatives, for example.

With the SPL design and implementation approach, some already solved problems of software engineering arise again. New ways have to be found to apply already existing code analysis techniques such as type-checking, model checking or data-flow analysis to variable programs such as SPLs. The existing analyses have to be adapted to be feature-aware, because they are not applicable to whole SPLs. For example, the product line may be organized in a different way than a classical program or contain additional language constructs—such as in feature-oriented programming (FOP). Analysis of all individual products is impractical in most cases as the number of products may exponentially grow with the number of features. An independent analysis of each product does not scale in this case. [TAK+12]

"Product line developers also face the problem of safe composition – whether every product allowed by a feature model is type-safe when compiled and run." [DCB09] Type checking with respect to a given type system is the analysis of well-typedness, so that type checking of a product line ensures safe composition of all valid configurations. [TBKC07] [TAK+12] [Pie02] This way, the *optional feature problem* can be identified, for example—that is, detection of references to undefined elements (such as classes, methods, and variables). [TBKC07] [TAK+12]

In this thesis, three different strategies for type checking of software product lines are evaluated in terms of performance by conducting representative type-checker tools to a benchmark set of 12 product lines. All product lines are implemented using feature-oriented programming (FOP) and the Java language. FOP "organizes programs around features rather than objects" [LKF05] and constitutes a paradigm for program design of software product lines that describes products as stacks of features. With each feature, its implemented functionality is added to the previously added one, so that different compositions of features produce different products. The features are implemented in form of modules that only define refinements to other features or the base program. [BO92]

Although, there are 12 benchmark product lines, all challenges and examples in this thesis are explained by means of the imaginary Editor product line. This product line implements a simple editor for text files and consists of the following three feature modules: The mandatory feature module *Base* provides the basic program window in the form of a tabbed-based editor pane and offers a toolbar which can be filled programmatically by refining the proper methods within other feature modules. New files can be created by using the feature *NewFile* that adds an appropriate button to the toolbar. The tabbed editor interface can be splitted into two views to enable the user to show two files side-by-side if the feature *SplitView* is selected. As the features are totally independent, there are four possible configurations of this product line: `{Base}`, `{Base, NewFile}` `{Base, SplitView}` `{Base, NewFile, SplitView}`. We use this example to highlight differences and challenges for each strategy.

1.1 About This Thesis

In this thesis, we compare three different strategies—*product-based*, *bytecode-based* and *family-based*—and their respective implementations regarding runtime for a type-check of a complete product-line. All strategies support detection of all occurring typing errors, but each strategy has specific advantages and drawbacks which are evaluated in this thesis. We give a short overview on the result of an benchmark conducted on example product lines and propose

provisional hypotheses on the approach best fitted with respect to circumstances and research goals.

We contribute an implementation of the bytecode-based strategy in terms of the FeatureBite toolchain. FeatureBite is a feature-oriented composer that composes feature modules to products on Java bytecode level. It performs type-checking after composition by means of bytecode verification as well. The toolchain is evaluated with respect to type checking completeness and challenges that occur due to the presence of bytecode.

The thesis is structured as follows:
In Chapter 2 "Methodology", we give an overview on the different strategies and their considered implementations. Moreover, the experiment setup and run-time measurement procedure for the performed benchmark are outlined.
In Chapter 3 "The Bytecode-Based Type-Checking", the bytecode-based strategy is described—that is, composition and type-checking of Java bytecode. The toolchain of FeatureBite is presented as well as challenges and peculiarities which are illustrated by means of the Editor product line.
In Chapter 4 "Results and Evaluation", the results of the benchmark are evaluated by detailed description of three product lines followed by conveyance of an overall picture for all product lines. Afterwards, we give advice what strategies are preferable for certain circumstances.
An overview on related work is given in Chapter 5. Finally, we conclude this thesis and outline potential future work regarding the bytecode-based strategy in Chapter 6.

2. Methodology

In this chapter, we explain different type-checking strategies and their respective implementations that are compared in terms of performance. Next, the experimental setup, the benchmark set of product lines, the measurement procedure, and the testing environment are presented.

2.1 Type-Checking Strategies

There basically exist three different type-checking analysis strategies besides from sampling and their mixed forms. The *family-based strategy* applies type-checking to the whole product-line, the *product-based* one analyzes all individual products separately and the *feature-based* analysis performs type-checks on all features individually. The *feature-product-based* strategy is an example for a mixed strategy: Its feature-based type-checking analysis is complemented with a product-based analysis.[TAK+12]

All strategies have their individual advantages and drawbacks and it is vital for users to choose the right strategy in accordance to their needs. Thüm et al. [TAK+12] describe the advantages and disadvantages of the type-checking strategies as follows:
The *product-based* strategy needs all products to be generated and type-checked individually. The main difficulty is to generate and check all products in a brute-force-like manner as the repeatedly parsing and type-checking consumes a considerable amount of time. This approach may not scale for large product lines with many independent features and thus, a large set of possible products. The set of valid products grows exponentially with the number of independent features. Moreover, there are many redundancies during the type-checking process due to the similarities of the individual products as

some are composed from partly the same feature modules. For the feature configurations `{Base, SplitView}` and `{Base, SplitView, NewFile}` of the EDITOR product line, the features *Base* and *SplitView* are type-checked in both configurations, although this is not necessary. More importantly, the product-based strategy is sound and complete in respect to type checking and standard type checkers can be used to check the single products, because these products do not contain any variability during type checking. This way, all existing errors in the product-line code-base can be found. Although, the erroneous feature module can not be identified, because the single products do not contain variability information anymore. If the features' code base changes, only affected products—that is, products that embody any changed feature—need to be type checked again.

The *family-based* strategy analyzes one single "meta-product" that contains the information about all products and is able to incorporate the variability information into the type-checking process, this way. This analysis strategy parses and composes all feature modules and integrates the information from the feature model into the composition process. This way, a more efficient analysis may be achieved, because no products have to be generated and the whole product line is taken into account in one single type-check run. The performance is generally independent of the number of valid feature combinations and solely depends on the number of features and their interactions as well as the size of the code base. The analysis can be optimized by the usage of caching algorithms for solving the feature interactions. This strategy is able to identify all type errors and can track them down to the causing feature module and inconsistencies in the feature model. On the downside, new algorithms have to be implemented to perform a family-based type-check. Considering that all feature modules are considered as a whole, the complete analysis has to be repeated when there are changes to the features' code base or the feature model. This can lead to high memory consumption and, potentially, time-expensive analysis repetition.

The *feature-based* strategy considers only the feature modules: The feature modules are independently type-checked in isolation, that is, without knowledge about other modules and the feature model. In contrast to the family-based strategy, a feature-based analysis supports open-world scenarios: not all features have to be known at analysis time.[LKF02] [LKF05] The drawback is that this strategy is incomplete, because type-checking is usually a non-compositional analysis technique as considers each feature independently. This is due to the fact that single feature modules are mostly only refinements to implementations of other feature modules or the base program. Thus, only errors that are locally restricted to a feature module—the *feature-local* errors such as typing errors caused by typos—can be found during a feature-based type-check. Dangling references to other feature's classes are not detected,

for example. As the features are checked apart from each other, only small repetitions are needed when a feature's code base is changed, only affected features have to be re-checked. Feature-based type-checking can be performed by already existing tools. But due to its incompleteness, the feature-based analysis can complemented with a product- or family-based analysis in order to cover feature interactions and to detect all type-check errors in the whole product line—examples for resulting mixed strategies are the *feature-product-based* and *feature-family-based* strategies.
An overview on this description can be found in Table 2.1.

Technique	**Errors**	**Erroneous Feature**	**Tool Reuse**
product-based	all	no	yes
family-based	all	yes	no
feature-based	feature-local	yes	yes
feature-product-based	all	no	yes
feature-family-based	all	yes	partly

Table 2.1: Overview on the three basic type-checking analyses and two mixed strategies with respect to their capabilities regarding error detection, erroneous feature blaming and tool reuse.

2.2 Type-Checking Strategy Implementations

The mentioned type-checking strategies can be implemented in many different ways. In this thesis, the product-based and the family-based strategy are implemented using the Fuji compiler.[1] Moreover, FeatureBite—which is a tool implemented by us—performs a feature-product-based type-check.[2] The pure feature-based strategy is skipped in order to compare three different implementations that are able to find all errors, especially those that occur as part of feature interactions (cf. Table 2.1).
These three implementations are compared later in terms of performance by measuring the analysis parts *setup*, *composition* and *type-check*. (cf. Section 2.3)

For the *product-based-strategy* implementation, all valid products are generated and type-checked using Fuji. (cf. Figure 2.1) Fuji is an extensible compiler for FOP in Java that is able to compose a selection of features to a single product as source code. Afterwards, this product is type checked using the Java type rules. The procedure is then repeated for each product.

[1] http://fosd.de/fuji/
[2] http://fosd.de/featurebite/

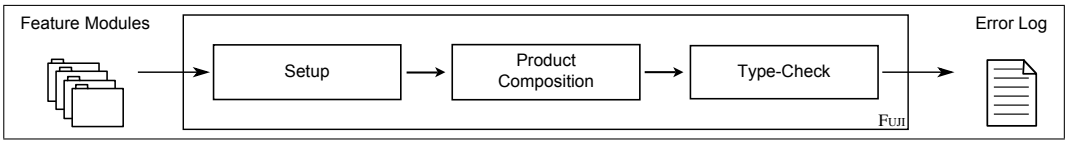

Figure 2.1: Analysis steps for the product-based strategy that have to be performed once per product.

The *family-based* strategy is implemented in the form of a tool called Family-Based Type-Checker.[3] (cf. Figure 2.2) This tool is an extension to Fuji that makes use of Fuji's ability to construct a "meta-product" from all features containing all variability information within the abstract syntax tree so that each node of the syntax tree knows which feature it belongs to. The type-checker implements algorithms on top of this "meta-product" mechanism to perform a family-based type-check which only has to be done once for the whole product line.

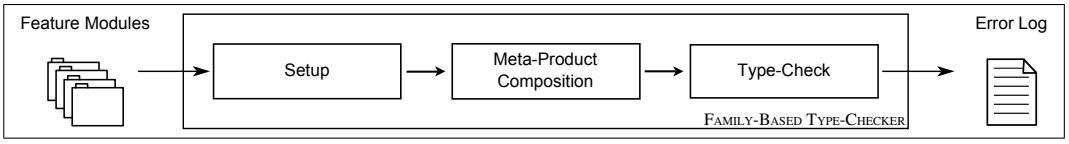

Figure 2.2: Analysis steps for the family-based strategy that have to be performed once per product line.

Applying the mixed *feature-product-based* strategy, initially, the feature modules are independently compiled to bytecode, then these compiled feature-modules are composed and type checked on bytecode instruction level. This sequence is performed using the three tools FeatureStubber, Fuji and FeatureBite. FeatureStubber[4] is a tool that enables isolated compilation of individual feature modules. The tool identifies and generates all lacking type-information for each feature that is needed for the feature compilation similar to mock-objects known from unit testing. Fuji complements the underlying feature-modules with this generated information—the *stubs*—and compiles them both to one single bytecode module. During feature compilation, a feature-based type-check is done by the compiler. Then FeatureBite can handle those bytecode feature modules and applies a composition mechanism that uses the same composition rules as Fuji, but may be faster than the product-based type-checking implementation, because it does not work on basis of an AST. The resulting product is then type-checked again using bytecode verification.[5] This is repeated for all possible products to type-check the whole product line.

[3]See http://fosd.de/fuji/ for details on the Family-Based Type-Checker.
[4]See http://fosd.de/featurebite/ for details on FeatureStubber.
[5]The complete toolchain is described more precisely in Chapter 3.

The implementation of the FeatureBite toolchain is very similar to the definition of the feature-product-based strategy described by Thüm et al. [TAK+12]. Though, it does not fully match the definition presented there due to the fact that the type-checking results from the feature-compilation step are ignored completely during bytecode verification and some type-checks are repeated, thereby. Consequently, the strategy that is actually used is referred to as *bytecode-based*.

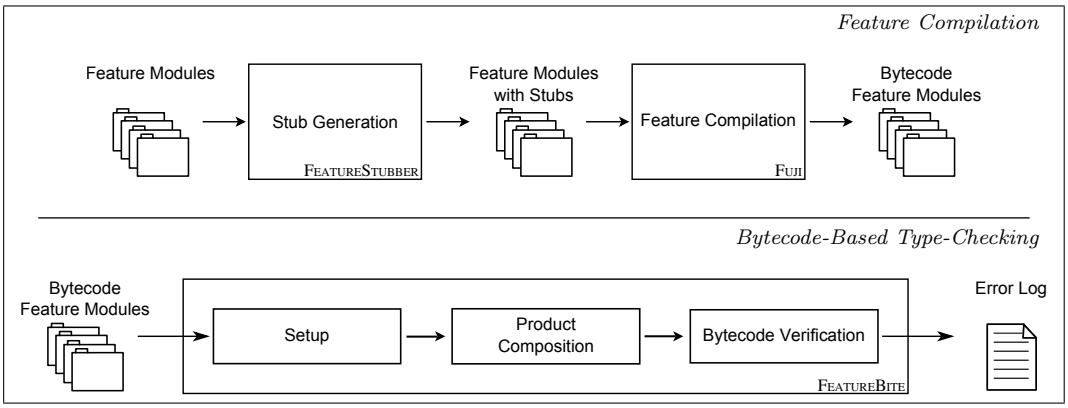

Figure 2.3: Analysis steps for the bytecode-based strategy: *(top)* feature compilation has to be performed once per feature module, *(bottom)* bytecode-based type-checking has to be performed once per product.

2.3 Experimental Setup

To measure performance of each strategy, a set of 12 SPLs is chosen for application of the implemented tools and toolchains described before. For comparison, the measurement procedure is divided in several sub-steps to identify expensive or cheap analysis parts, respectively.

Benchmark Set

In order to compare the implementations of the described strategies in terms of performance, type-checking time for 12 software product lines is measured. The SPLs are implemented in Java using feature-oriented programming. They are widely used throughout the SPL community and are distributed with the Fuji compiler. They were also used in several studies before. [AB11] [AKL+12] [KBK09] [LHB01] These product lines belong to different domains and differ in number of lines of code (LOC), features and products. Table 2.2 on the following page shows a full list of the SPLs, their number of features, products and LOC.

For the product- and bytecode-based strategies, the list of all configurations per product line is generated using FeatureIDE. [TKB+13] The resulting files were sequentially passed to the composition tool during the experiment.

The VIOLET product line has approximately 2^{89} valid products as the features are totally independent. In this thesis, only 40 random products are checked for the product- and bytecode-based strategies, because checking all 2^{89} products is not possible in reasonable time. The FAMILY-BASED TYPE-CHECKER is able to check the whole product line, though.

SPL	**Domain**	**#f**	**#p**	**LOC**
EPL	expression evaluation	12	425	126
GPL	graph library	27	156	2,951
Graph	graph library	5	16	596
GUIDSL	configuration tool	28	24	15,988
Notepad	text editor	13	512	2,732
PKJab	chat client	8	48	5,000
Prevayler	persistence framework	6	32	5,938
Raroscope	compression library	5	16	438
Sudoku	game	8	64	2,130
TankWar	game	37	2,458	5,604
Violet*	model editor	89	$\sim 2^{89}$ (40)	11,006
ZipMe	compression library	13	24	5,096

Table 2.2: Overview on the benchmark set of product lines. (* Violet would have approximately 2^{89} products, but only 40 were type-checked because of time issues.)

Measurement Procedure

To measure the strategies' performance on the benchmark, there are three time intervals of the whole type-checking process that are measured for each analysis run: setup, composition and type-check. *Setup* includes all working steps any implemented analysis tool needs to take in order to prepare for the later steps: for example, the initialization of auxiliary data structures and parsing of configuration files. *Composition* includes steps that are necessary to construct a structure such as an AST to perform the type checking on. For the product- and bytecode-based analyses, this is the time to compose the particular product that is to be type-checked later; for the family-based approach, the composition yields the "meta-product". Lastly, the time for the actual *type-check* of a single product or the meta-product of the family-based analysis and any time needed for error logging is measured. During measurement, no times for code generation are collected, so that access rate for disk writings does not have any impact on the results. In contrary, the read-in of the code is measured, because it is not always separable from the API. It rather is an essential part of the experiment as becomes clear when analysing the measurement results.

The to-be-measured time used for the bytecode-based strategy also includes the time needed for compilation of individual features. This time slot is called *feature compilation*.[6] During feature compilation, the features undergo a partial feature-based type-check by the compiler.

As implied by the definition of the product-based and feature-product-based strategy, measurements of the three time intervals of the bytecode- and product-based analyses have to be repeated for all possible valid configurations. The sum of all these measurements is the time for type checking the whole product line. The family-based analysis is performed only once for the whole product line. A formal description for all analyses can be found in Figure 2.4.

There are upper bounds to the number of runs for each of the strategies which apply when analyzing all products of a product line—as denoted in Figure 2.4, too. A SPL with n independent features has 2^n possible feature combinations. In this context, 2^n runs have to be performed for the product-based strategy as well as the bytecode-based one. Additionally, the bytecode-based analysis compiles all features (n) once. The family-based analysis needs exactly one run and thus, one measurement, because it checks only one single meta-product containing all information about all products. Usually, there are fewer than 2^n feature combinations due to constraints in the feature model. The run-times which are to be measured and the results about the fastest strategy and most accurate one can not be estimated with these formulas as they model just the number of runs for each strategy. These formulas are used, for example, to identify potential drawbacks that a strategy can have. The actual run-time depends on the number of selected features per product, the size of these features, and, eventually, on the feature composition order. The three strategies are compared in Chapter 4.

2.4 Testing Environment

The measurements were carried out under a x64 GNU/Linux OS, using OpenJDK Runtime Environment (IcedTea6 1.12.5, Java 1.6.0_27). The testing machine was a Dell Optiplex 7010 workstation with Intel® Core™ i7-3770 (eight cores, 3.4 GHz), 16.7 GB DRR3 RAM (1600 MHz) and 1.0 TB Serial ATA harddisk.

The time intervals stated before (cf. Section 2.3) were measured using the class `ThreadMXBean`[7] and its method `getCurrentThreadCpuTime()`. The time for JVM start-up is left out of the comparison, because it is difficult to exactly measure the times for JVM start-up, garbage collection or any other automatic Java background service.

[6] See Section 3.1 for details.

[7] `java.lang.management.TreadMXBean` is part of the standard Java API since Java 1.5.

product-based

$$2^n * [\text{setup} + \text{composition} + \text{type-check}] \tag{2.1}$$

bytecode-based

$$n * [\text{feature-compilation}] + 2^n * [\text{setup} + \text{composition} + \text{type-check}] \tag{2.2}$$

family-based

$$1 * [\text{setup} + \text{composition} + \text{type-check}] \tag{2.3}$$

Figure 2.4: Formal description of measured time intervals for all implemented type-checking strategies for a product line with n independent features. (cf. Figure 2.1, Figure 2.2 and Figure 2.3)

The output during the runs is kept minimal: Error messages and warnings for the current composition as well as the measured times are logged to plain-text files, all other debug messages are turned off. This way, the log files can be parsed easily for evaluation and the negative impact on the performance of the analysis is limited to a minimum.

The sequential process of measuring all implemented type-checking analyses is done by a self-built bash script that also manages data and generation output. The script analyzes all sample product-lines from Table 2.2 sequentially and dumps the relevant log for later evaluation.[8]

[8]See digital copy of this thesis for a functional measurement setting and Chapter 4 for measurement results.

3. The Bytecode-Based Type-Checking

In this chapter, we give a detailed overview of the bytecode-based strategy. It explains how stub generation and feature compilation exactly work. Afterwards, bytecode-based feature-composition is described and some challenges concerning the bytecode-based composition are discussed. Finally, bytecode verification and the very implementation within FeatureBite are described.

Considering that the pure feature-based strategy only finds feature-local errors, consequently, it is complemented with a product-based strategy in this thesis. The resulting mixed strategy achieves the same level of full inter-feature error detection for the whole product line as with the other two type-checking strategies. The described mixed strategy is similar to the feature-product-based strategy described by Thüm et al. [TAK+12], but does not pass any type-checking information from the feature-based part to the product-based one as required by the definition. This is why this approach is called *bytecode-based* in this thesis.
Feature compilation and the inherited feature-based type-check are extended with a composition and type-checking process on per product basis. The *feature compilation* makes the independent feature modules compilable by using stub-generation and produces modules consisting of bytecode files. Afterwards, *feature composition* is done by FeatureBite. It works on bytecode files and composes the bytecode modules as Fuji or FeatureHouse do it with source files. FeatureBite is used to generate all valid products of the product-line and performs a bytecode-verification process on each single product which is actually a type check with some additional checks on data-flow, for example.

3.1 Stub Generation and Feature Compilation

Feature modules cannot be compiled independently out of the box, because they are often only fragments of classes that depend on other features' code. Therefore, *stubs* have to be generated or provided by the developer. (cf. Hyper/J [TOS02] and AHEAD tool suite [TBKC07]) Stubs are a bundle of Java interfaces and classes containing member declarations that are needed from the other features. They exactly fit together with the particular feature that they were generated for, so that the feature module can be compiled. In this thesis, the stubs were generated using FeatureStubber.

FeatureStubber uses a closed-world approach for generating stubs for each feature module. The tool scans the *references* and *introduces* files that are generated by Fuji and that contain information about which feature introduces which classes, methods and files and what types are referenced. This way, it is possible to identify what stubs need to be generated in order to compile each feature.

FeatureStubber generates stubs in a way that they model new feature modules that are refining the corresponding original feature modules. The fields and methods within a stub class are marked with the annotation `@Stub`, so they may be easily identified and removed during bytecode composition. Fuji is able to compose a feature module and its stubs and compile them together into one single feature module consisting of bytecode files.

The straight-forward application of stub generation and bytecode composition may result in errors due to the Java language specification. (cf. Section 3.3.2)

3.2 Feature Composition and Type Check

The feature composition is accomplished by FeatureBite. It is a feature-oriented composer for Java bytecode that composes the feature modules using the same composition rules as Fuji.

The composition mechanism of FeatureBite is divided into three parts:

1. renaming of anonymous inner classes (cf. Section 3.3.1),
2. composition using the same rules as applied by Fuji and removal of the stubs annotated with `@Stub`,
3. type checking or rather bytecode verification (cf. Section 3.4).

Within FeatureBite, Java bytecode is handled using the ASM bytecode framework.[9] It handles the bytecode classes via the visitor pattern and in a string-based manner. Class, field and method names as well as any other information are generally handled via String objects so that modification of this information is easy to implement.[10] Though, the bytecode verification is done using the BCEL JustIce verifier framework[11]. Reasons for usage of another bytecode framework are explained in Section 3.4.

3.3 Challenges with Bytecode Composition

There are some challenges that arise during feature compilation and composition of bytecode feature modules. The main three investigated difficulties are discussed below by means of the Editor product line to illustrate composition process and semantics.

3.3.1 Anonymous Nested Classes

During the compilation process, *anonymous classes* become an independent class file. These classes are not named, so that the standard compiler provides class names using numbers. As shown in Figure 3.1, the method getFileChangedListener() within feature *Base* returns an anonymous class instance of the interface ActionListener and getFileComparator() within feature *NewFile* an anonymous Comparator instance. As a result, two class files are produced for each feature during feature compilation: For the *Base* feature, the files *Editor.class* for the main class Editor and *Editor$1.class* containing the anonymous ActionListener implementation are generated; for the feature *NewFile*, also a file named *Editor.class* and *Editor$1.class* are produced where the latter one consists of the anonymous Comparator implementation.

In the following, the generated class files for each feature must be independently treated with a *renaming* process, because it is not guaranteed that *Editor$1.class* within feature module *Base* and *Editor$1.class* within *NewFile* embody the same class. It is generally more likely to assume that the second class file contains a totally different class—as in this example, where the class files are not implemented for the same purpose. In order to differentiate the two classes during the composition process, they must be renamed.

FeatureBite accomplishes the renaming by appending the feature's name to the number of the anonymous class file, similar to the process of renaming original-calls. *Editor$1.class* from feature *Base* is renamed to *Editor$1Base.class*, the class from feature *NewFile* to *Editor$1NewFile.class*, accordingly. Not only the class file's name is changed, the class' name itself

[9] http://asm.ow2.org/
[10] See http://fosd.de/featurebite/ for implementation details.
[11] http://commons.apache.org/proper/commons-bcel/

Feature *Base*

```
1  public class Editor {
2    ActionListener getFileChangedListener() {
3      return new ActionListener() {
4        @Override
5        public void actionPerformed(ActionEvent ae) {/*...*/}
6      };
7    }
8  }
```

Feature *NewFile*

```
9  public class Editor {
10   Comparator<File> getFileComparator() {
11     return new Comparator<File>() {
12       @Override
13       public int compare(File file1, File file2) {/*...*/}
14     };
15   }
16 }
```

Figure 3.1: Example for anonymous classes in different features but the same outer class Editor: *(lines 1–8)* Method getFileChangedListener() returns an ActionListener instance; *(lines 9–16)* Method getFileComparator() returns a Comparator instance.

is altered, too. Therefore, all references in the main class file *Editor.class* to the now-renamed class have to be adjusted for each feature independently. Otherwise, the linking would break.
The renaming mechanism used by FeatureBite guarantees unique class names after renaming, because class names starting with a number are forbidden by the *Java Language Specification* [GJSB13, Section 3.8] and the feature names are unique.

3.3.2 Conflicting `original`-Calls

With bytecode composition, original-calls are another challenge. The call of the original-method expresses a call to the refined method implementation. Because this method with the name original does not exist, the method must be generated by the feature-stub generator in order to be able to compile a feature on its own (cf. Section 3.1). If the original-method-call is used several times within a refining class, one generic stub-method must be generated for all calls together in order to not endanger type safety. An example for the Editor product line using the features *Base* and *SplitView* is explained in the following.
All methods getCurrentFileText(), getCurrentFileTabIndex() and getToolbar() in Figure 3.2 (lines 1–14) use an original-call but with different return types (String, int and JToolBar). Normally, the stub-generator FeatureStubber produces all correspondent stub methods with the method name original. (cf.

Figure 3.2, lines 15 ff.) As the original-methods' signatures conflict with each other with respect to the *Java Language Specification* [GJSB13, Section 8.4.2], the feature compilation will fail with these specific stubs.

Feature *NewFile*

```
1  class Editor {
2    String getCurrentFileText() {
3      String s = original();
4      // ...
5    }
6    int getCurrentFileTabIndex() {
7      int originalId = original();
8      // ...
9    }
10   JToolBar getToolbar() {
11     JToolBar originalToolbar = original();
12     // ...
13   }
14 }
```

Stubs for feature *NewFile*

```
15 class Editor {
16   @Stub String original() {
17     return null;
18   }
19   @Stub int original() {
20     return 0;
21   }
22   @Stub JToolBar original() {
23     return null;
24   }
25 }
```

Figure 3.2: Example for conflicting original calls: *(lines 1–14)* Class Editor of feature *NewFile* has original-calls with different return types; *(lines 15–25)* Generated stubs for the Editor class that conflict with each other.

To solve the problem, a stub with the generic signature Object original() and the return value null can be created. The return values must be down-casted from Object when used, which causes small modifications to the underlying code base of the feature modules. (cf. Figure 3.3) These changes generally should be avoided, but for this problem, they enable bytecode-based composition and analysis for the affected product-line in the first place.
Especially, primitive types cause problems with this generic original-method returning Object. The primitive return type of the method that is calling original must be altered to the corresponding wrapper type, otherwise there will be a conflict within bytecode later after composition. The signature int getCurrentFileTabIndex() in Figure 3.2 implies that the return type of the method that the pseudo-method original symbolizes is int. But the generic original-

Feature *NewFile*

```
1  class Editor {
2    String getCurrentFileText() {
3      String s = (String) original();
4      // ...
5    }
6    Integer getCurrentFileTabIndex() {
7      int originalId = (Integer) original();
8      // ...
9    }
10   JToolBar getToolbar() {
11     JToolBar originalToolbar = (JToolBar) original();
12     // ...
13   }
14   @Stub Object original() {
15     return null;
16   }
17 }
```

Figure 3.3: Class Editor of feature *NewFile* contemplated with its stubs which are fixed with the wrapper-type workaround. (Relevant code is underlined. See Figure 3.2 (lines 1–14) for comparison.)

method suggested above returns a class reference or rather an instance of Object instead. The downcast depicted in Figure 3.3 (lines 3, 7 and 11) is not problematic, because autoboxing resolves the conversion from primitive to wrapper type automatically but the corresponding JVM instructions for the return values are not compatible.[12] So, the return type of the method getCurrentFileTabIndex() must be changed to Integer—here and in all other feature modules that implement or refine this particular method. Otherwise, the composed bytecode will not be runnable.

The usage of wrapper types is the workaround that is used for this thesis, because it was the quickest solution for all affected product-lines to fix the issue and enable bytecode composition. One general fix to avoid the return-type change to wrapper types in the source files demands intervention on bytecode level and adjustment of the bytecode instructions. Another one would be the extension of the name original to a pattern such as `"original(\_\$\_(\w*?))??"`[13].

3.3.3 Fields Initializations

Constructors are composed by adding an original-call to the refining feature's constructor that calls the base's constructor and does not have to be inserted

[12] The bytecode instruction ireturn pushes an int value onto the stack, areturn a object reference.

[13] Denoted as Java pattern construct. See http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html for details.

explicitly by the developer. For bytecode validity, the base's constructor is transformed into an unique private method, similar to the renaming of an original-call or anonymous class' files. (cf. Section 3.3.1)
This mechanism results in a problem with field initializations that is caused by the compilation process and the stub-generation as denoted in Section 3.1. The reasons for the following problem are attributable to the fact that feature compilation is performed before feature composition.

During compilation, all non-static field initializations from the field definitions are transferred to the beginning of the constructor, following the order for creating new class instances given by the *Java Virtual Machine Specification*. [LY13, Section 2.17.6, points 4 and 5] In the product-based approach on the one hand, the explicit constructor statements are composed first and then the field initializations from the declaration area are prepended. In the bytecode-based approach on the other hand, firstly, the initializations in the declarations are prepended to each feature's constructor during the feature's compilation. Secondly, the single constructors are composed using a non-explicit original-call as described above. This results in a changed order of the constructor statements for the bytecode-based composition compared to the product-based one. Hence, the bytecode-based composition output may have a different semantic behavior or even cause a NullPointerException.
Figure 3.4 illustrates such error-prone feature modules on basis of the EDITOR product line, relevant lines are colored. The class Editor from feature *Base* defines a hook method named generateToolbar(). This method can be used to fill the editor toolbar with actions by refining it within other feature modules. The feature module *NewFile* refines this method and just adds a "New File" button to the toolbar.

Feature BASE

```
1  class Editor {
2    JToolBar toolbar = new JToolBar();
3    Editor() {
4      generateToolbar();
5    }
6    void generateToolbar() {
7    }
8  }
```

Feature NEWFILE

```
9  class Editor {
10   Button newFileButton = new NewFileButton();
11   // constructor not defined explicitly!
12   void generateToolbar() {
13     original();
14     this.toolbar.add(this.newFileButton);
15   }
16 }
```

Figure 3.4: Class Editor within the feature modules *Base* ■ and *NewFile* ■ where *NewFile* refines *Base*.

When composing a product with both feature modules *Base* and *NewFile* on source-code level using FUJI (cf. Figure 3.5), the field newFileButton is initialized before the method generateToolbar() is called within the constructor and, as a result, a functional button is added to the toolbar by executing the generateToolbar() method—as intended by the developer of the *NewFile* feature module.

Product {*Base* ■, *NewFile* ■}

```
1  class Editor {
2    JToolBar toolbar;
3    Button newFileButton;
4    Editor() {
5      this.toolbar = new JToolBar();
6      this.newFileButton = new NewFileButton();
7      generateToolbar();
8    }
9    void generateToolbar_$_base() {
10   }
11   void generateToolbar() {
12     generateToolbar_$_base();
13     this.toolbar.add(this.newFileButton);
14   }
15 }
```

Figure 3.5: Composed class Editor using FUJI with configuration {*Base* ■, *NewFile* ■}.

When using FEATUREBITE for product generation (cf. Figure 3.6), the initialization for the field newFileButton is shifted from line 6 to line 7 in comparison to the FUJI composition in Figure 3.5. Thus, the bytecode-based composition results in a different order of the constructor's statements. As a consequence, the field newFileButton is used in the generateToolbar() method before it is initialized. A null-pointer is added to the toolbar, because the according field is not initialized yet. Thus, the toolbar button appears but it is non-functional and the application becomes partly unusable due to thrown NullPointerException instances.
Now, it is simple to think of scenarios where more serious unintended side effects occur that are able to break the program or cause data loss. Thus, it is recommended to generally call generating methods, such as the described generateToolbar() method, after completion of the constructor using a separate method call.

3.4 Bytecode Verification and Type Checking

Generally, bytecode verification according to the *Java Virtual Machine Specification* [LY13] is necessary, because there is no assurance for the Java Virtual Machine (JVM) that the bytecode it is about to load is valid. Classes or rather

Product {*Base* ■, *NewFile* ■}

```
1  class Editor {
2    JToolBar toolbar;
3    Button newFileButton;
4    Editor() {
5      this.toolbar = new JToolBar();
6      generateToolbar();
7      this.newFileButton = new NewFileButton();
8    }
9    void generateToolbar_$_base() {
10   }
11   void generateToolbar() {
12     generateToolbar_$_base();
13     this.toolbar.add(this.newFileButton);
14   }
15 }
```

Figure 3.6: Composed class Editor using FeatureBite with configuration {*Base* ■, *NewFile* ■}. (Excerpt decompiled from bytecode.)

their implementations can be changed while introducing several versions during implementation progress. It is possible to exchange a bytecode file that was used to compile a product with another one with the same name, while it is not guaranteed that those versions are compatible in any way or the new class was changed by an attacker in order to exploit the JVM.
Due to these problems, the JVM has to verify the bytecode before executing it to ensure following properties for safety reasons: *(i)* there are no operand stack overflows or underflows, *(ii)* the usage and storage operations of all local variables are valid, and *(iii)* all arguments passed to JVM instructions are of valid types. [LY13]
Especially, the last two points are necessary for the type-checking research done in this thesis. The bytecode verification actually inherits a type check, but performs some additional checks such as a data-flow analysis for security reasons. So, verification of a class file guarantees type safety and thus, satisfies the requirements of the executed experiment stated in Chapter 2.

In order to show how many checks the bytecode verification does, the verification process is outlined below. Afterwards, the particular implementation for FeatureBite is described.

3.4.1 The Verification Process

The so-called *class file verifier* is language-independent, because many different programming languages can be compiled into bytecode. [LY13, Section 7] The verifier performs four passes on the bytecode to verify the constraints mentioned before. By performing these four passes in order, it becomes clear that the bytecode verification replaces a full type-check.

Pass 1 of the verification process is applied on loading a class file. It ensures that the loaded file satisfies the basic format of a class file. Any class file has to begin with the Java-specific magic-number 0xCAFEBABE. This pass generally checks for basic class file integrity, so that there are no truncated data, extra bytes or other unrecognizable information. This is necessary for any interpretation of the class file.

Pass 2 performs additional verification, when the loaded file is linked. It checks for final modifiers and that this modifier is respected when used. This pass also ensures that any class (except Object) has a direct superclass. After this, the constant pool is checked for its validity as well as that field and method references within the constant pool have valid names. Pass 2 does not check classes, fields and methods for their existence, it checks only for well-formedness.

Pass 3 is the most complex pass of the verification process and consists of two consecutive runs. In the first run, pass 3 checks whether all instructions within the current method are disciplined and are used with an appropriate type. Truncation and improper exception handling is detected and reported. After this run, the verifier knows about the current contents of the local variables table and the stack at each single instruction. Accordingly, the types of all used and needed variables and objects are known. In the second run and on top of the information from the first run, a data-flow analysis is performed. For each instruction, its effect on stack and local variables is modeled. This way, the verifier is able to guarantee that local variables and fields are accessed with proper types and methods are invoked with appropriate arguments. If any condition for the current instruction is missing—wrong type while assigning a field value or missing parameter for a method invocation, for example—, the verification fails. Otherwise, the analysis is then continued at the instruction's successors.
During this pass not all classes are necessarily loaded and checked. A class is only loaded if one of its methods is called with other types than implied by its signature. By passing a subtype object as parameter—as allowed by object-oriented polymorphism within Java [GJSB13, Section 5.3]—, the called class will get loaded and checked, otherwise not. All remaining non-loaded classes are loaded in pass 4.
If the data-flow analysis does not report a failure, the method under consideration is verified by pass 3 of the bytecode verifier.

Pass 4 is a virtual pass that is performed by specific JVM instructions during code invocation. It loads all referenced types that have not been loaded before and checks whether the executing type is allowed to reference the type in regard to access and visibility. Referenced methods and fields are checked for

their existence in the given class, the right signature and visibility. This way, dangling references can be identified.

3.4.2 Implementation of the Verification Process

There are two different libraries for bytecode verification that were considered for implementation into FeatureBite: ASM and BCEL JustIce.
ASM carries a built-in class-file visitor called CheckClassAdapter whose static method verify is able to perform a pass 3 verification. Some errors cannot be found with this single pass—such as non-existing classes and fields. (cf. Section 3.4.1)
A more complete library is implemented in *BCEL*.[14] The BCEL verifier is called JustIce and checks class files according to the specification of JavaSE 1.4. This yields some problems with classes that are compiled with a newer version of the Java compiler. The consequence are some false-positives that correspond to newer versions of bytecode instructions, for example, that are now able to handle additional types on the stack.

For this thesis, the BCEL framework is used, because it performs a more complete verification compared to ASM. It can identify dangling references, for example, because pass 4 of the verifier is implemented. Overall, it models the type-check and its temporal extent more accurately, although some false-positives are expected as the classes are compiled with a newer version of Java than 1.4 (cf. Section 2.4).
A downgrade to Java 1.4 is not possible, because generics which are first introduced in Java 1.5 were used for the chosen product lines quite extensively, for example. Also autoboxing which is used to fix the field-initialization problem (cf. Section 3.3.3), the enum keyword and the enhanced for-loop (also called foreach-loop) were introduced with Java 1.5.

3.5 Summary

As we showed before, the bytecode-based composition and type-checking approach implemented by us is not flawless. There are several points regarding the composition of bytecode that the developer must have an eye on and that sometimes need manual action. (cf. Section 3.3) The products composed products may become unusable, especially, if they exhibit the field-initialization problem (cf. Section 3.3.3), but this is no problem for the type-checking process. Furthermore, bytecode verification performs checks that are not necessarily needed for type-checking, such as the extensive data-flow analysis, but are needed for security reasons. Nevertheless, the used implementation of the verifier, BCEL JustIce, is considered as a good indicator

[14] http://commons.apache.org/proper/commons-bcel/

in respect to run-time for a full Java verifier as it exists within the JAVAC tool.
Altogether and apart from the challenges, the bytecode-based composition and type checking can be a full substitute for the product-based implementation.

4. Results and Evaluation

In this chapter, an evaluation of the *product-based*, the *bytecode-based* and the *family-based* analysis is discussed. Firstly, the results for the GUIDSL, Prevayler and EPL product lines are described with use of appropriate diagrams. (cf. Section 4.1 and Section 4.2) Afterwards, we discuss all results in an overall picture in Section 4.4. Finally, a short advice based on this experiment and the used type-checker implementations which analysis is preferable under which circumstances is given. (cf. Section 4.5)

We evaluate the considered strategies by application of implemented tools and toolchains to the 12 product lines mentioned in Section 2.3. The measurement results are presented in Table 4.3 on page 36. For each benchmark product-line and type-checking analysis, the setup time, the composition time, the type-checking time, and the sum of all these measured times is presented. For the bytecode-based strategy, the time for feature-compilation is also listed.

In the following, these measurement results are visualized by three different types of diagrams for each SPL illustrating different aspects and questions.
A stacked bar plot (*sum*) (cf. left plot of Figure 4.1) shows the total run-time for all products or the whole product line by pointing out the different stages through the type-checking process. The *sum*-labeled diagrams show the general comparison which strategy takes the longest for the current product line. Differences in the whole type-checking process can not be identified, such as single products can not be recognized in this plot. The order of the stacked bar plot is from bottom to top: feature-compilation (■, if present for strategy), setup (■), composition (■) and type check (■).
The second bar plot (*avg*, cf. right plot of Figure 4.1) illustrates the measured run-times for type checking an average product of a product line. It indicates if any strategy's implementation needs much upfront investment.

Those approaches may have a high total run-time for checking an average product. For example, this applies for the feature compilations as it has to be done completely before using the bytecode-based composition. The *avg*-labeled diagram is not applicable for the family-based type-checking strategy as it always checks the whole product line and can not be utilized for only a few products. Hence, the run-times for the whole family-based type-checking process are presented here. Colors and stack order for this kind of diagram are the same as for the *sum* diagram.
The *cumul* graph (cf. Figure 4.2) is a line plot and visualizes the cumulative run-time by the number of products for each product line. The measurement of the family-based analysis is presented as a point at the maximum number of products, because all products are type-checked at once. This point is complemented with a horizontal line to illustrate the measurement in relation to the other strategies and to make comparison easier. This kind of graph is for identification which approach is suitable for how many products. By incrementing the number of products while cumulatively summing up the needed run-time, intersections between the different lines can be found where the break-even point is reached at which one strategy becomes superior to another.

4.1 GUIDSL

The left plot of Figure 4.1 shows the *sum* results of the different type-checking strategies for the GUIDSL product line. It is clear that the family-based strategy is the fastest for this product line, because the run-time for type-checking all 24 products is the shortest. It only takes 3.26 seconds, wheras the bytecode-based analysis (49.74 s) needs almost twice the time that the product-based one uses (28.86 s). Thus, the family-based type-checking analysis is 8.85 times faster than product-based and even 15.26 times faster than the bytecode-based one. The feature compilation alone takes 12.94 seconds which is around 26% of the whole bytecode-based strategy's run-time and almost 4 times the amount of time the family-based analysis needs for type-checking the whole product line.
The right plot of Figure 4.1 (*avg*) supports the assumption that feature compilation is a great upfront investment to the bytecode-based technique, especially, when analysing a small number of products. It takes nearly 90% (12.94 of 14.473 seconds) of the total run-time for an average product. Also as assumed, the product-based strategy needs less time (1.203 s) than the family-based (3.26 s), though no average product can be calculated for the latter one.

Figure 4.2 shows the *cumul* diagram in which the break-even point (3 products) for the family-based strategy is indicated here by the intersection of the

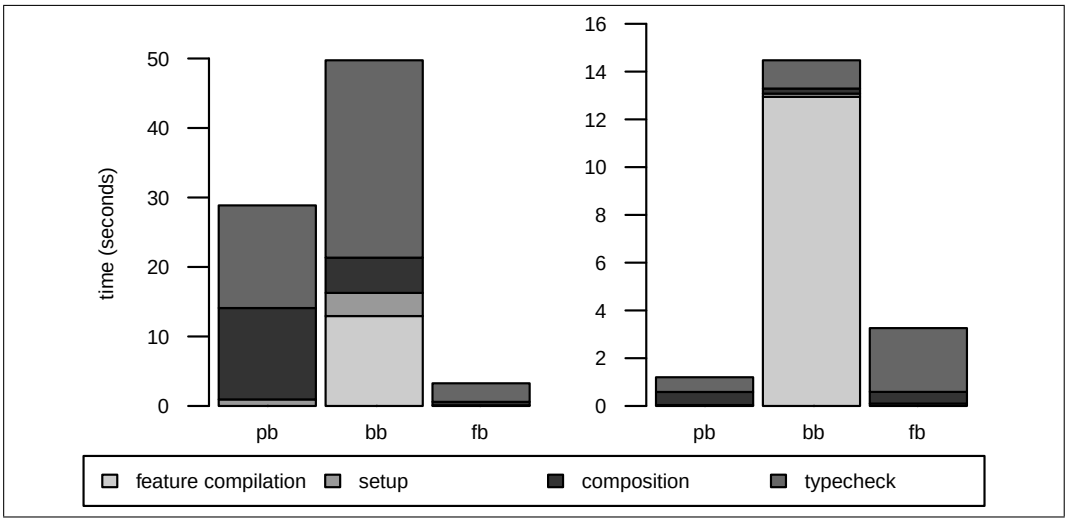

Figure 4.1: Plots for the run-time measurements results (in seconds) of GUIDSL: (left) *sum* diagram, (right) *avg* diagram. (pb = product-based, bb = bytecode-based, fb = family-based)

corresponding line with the product-based one. When type-checking three or more products, consequently, the family-based technique is superior to the other ones. (cf. Table 4.2 on page 33) The bytecode-based strategy is the slowest for any number of products that are checked. Referring to Table 4.3, setup and composition together are faster than for the product-based strategy, thus, the cause for the long overall run-time of the bytecode-based strategy lies in the feature compilation and the type-check. The table of measurements also indicates that the type-check in particular is the longest step of the bytecode-based strategy (28.40 s), just slightly shorter than the whole type-checking process of the product-based strategy (28.86 s). The huge amount of time can be explained by the relatively large code base of GUIDSL that makes the bytecode verification and especially its extensive data-flow analysis expensive.

4.2 Prevayler

The *sum* diagram of Figure 4.3 illustrates that the bytecode-based strategy is the slowest and the family-based analysis is the fastest for Prevayler. Type checking of all 32 valid products (generated on basis of 6 features) using the family-based analysis strategy is even faster (1.94 s) than the feature compilation of the bytecode-based analysis for itself (3.58 s). The feature compilation can take extremely long—though the time for it only makes up around 9% of the whole run-time for the bytecode-based strategy. The type-check itself as a bytecode-verification with BCEL JustIce can get expensive, because the methods within the single feature modules are complex and long in terms of LOC (6 feature modules and 5,938 lines of code) which induces

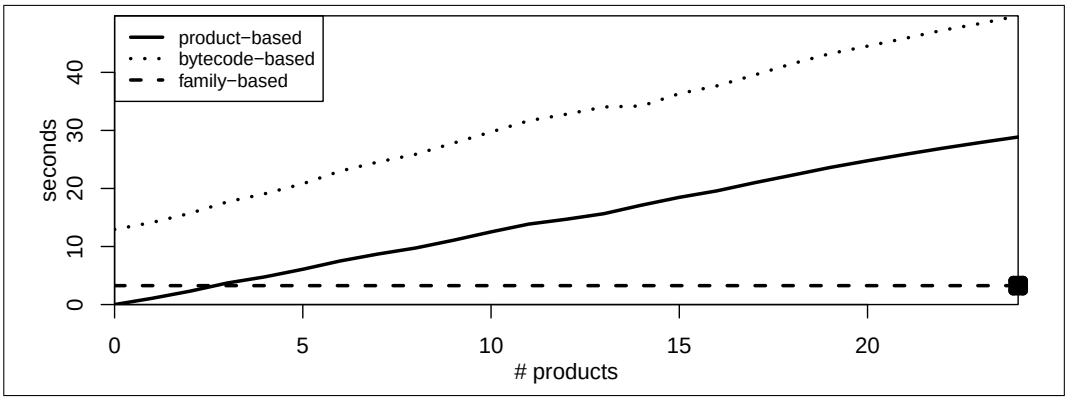

Figure 4.2: Run-time (in seconds) of each strategy by the number of type-checked products for the GUIDSL product line (*cumul* diagram).

an extensive data-flow analysis in step 3 of the bytecode verification and a high cost of those additional checks it performs. Therefore, the type-check itself of the bytecode-based analysis for PREVAYLER takes more time (37.79 s) than all time intervals of the product-based analysis (36.04 s). Also, the type check takes around 76% of the total time. The *avg* diagram for PREVAYLER in Figure 4.3 confirms the huge upfront investment with the feature compilation which makes up approximately 70% of the time to analyze an average product.

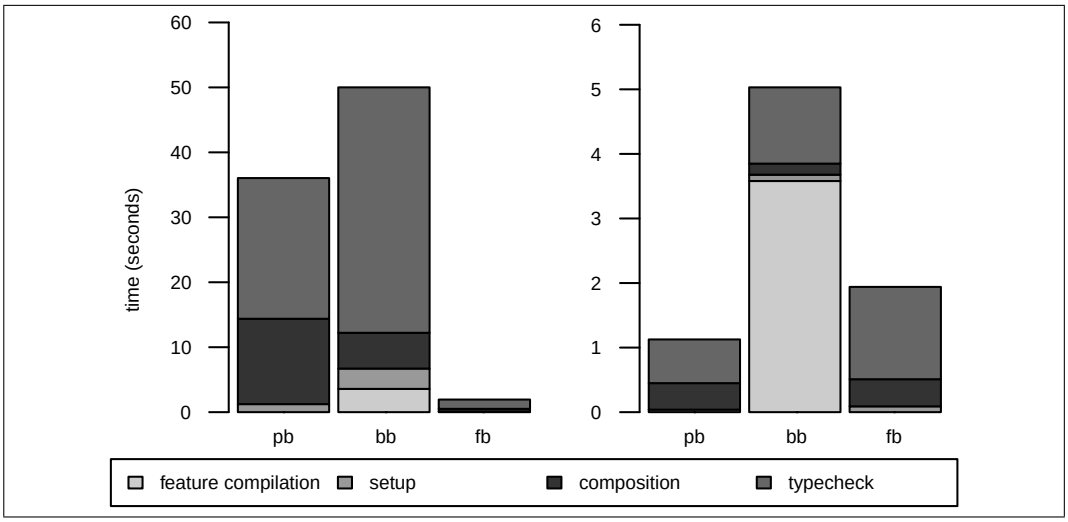

Figure 4.3: Plots for the run-time measurements results (in seconds) of PREVAYLER: (left) *sum* diagram, (right) *avg* diagram. (pb = product-based, bb = bytecode-based, fb = family-based)

Referring to the *cumul* plot for PREVAYLER (cf. Figure 4.4), the same conclusions can be drawn as from the GUIDSL product line. The bytecode-based strategy is not faster than both other strategies for any number of products, mostly due to the feature-compilation upfront investment. The break-even

for the Family-Based Type-Checker against pure Fuji is at the second product.

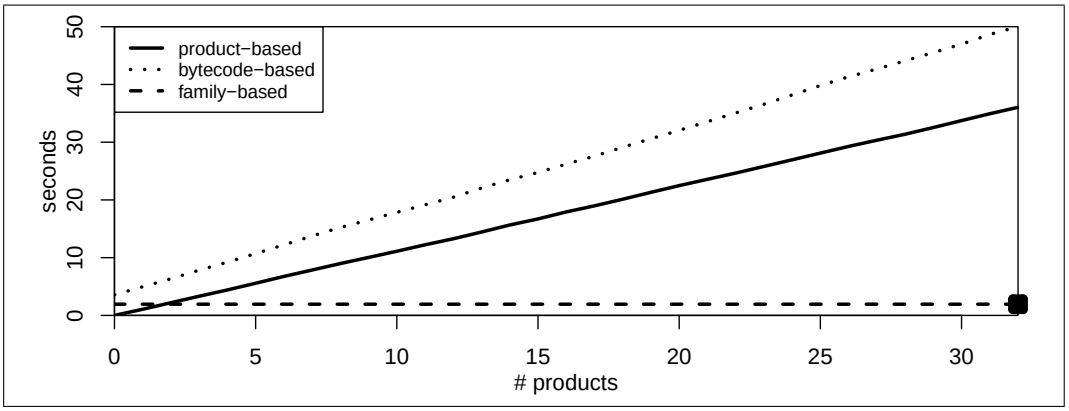

Figure 4.4: Run-time (in seconds) of each strategy by the number of type-checked products for the Prevayler product line (*cumul* diagram).

4.3 EPL

Thoughm, the *sum*-labeled diagram (cf. left plot of Figure 4.5) shows that the family-based strategy is the fastest approach, the bytecode-based strategy outperforms the product-based one for the EPL product line. Exactly, the bytecode-based strategy performs 1.27 times faster than the product-based one, the break-even point is at 52 of total 425 products. (cf. Figure 4.6) The bytecode-based strategy takes 88.52 seconds and the product-based one 112.47 seconds, although the family-based one takes just half a second (530 ms). Especially, the setup and composition of the bytecode-based strategy is by far faster (around 4 times) than the product-based equivalents. The setup and composition is so slow for the product-based strategy, because the amount of initialization is too high for this very small product line (126 LOC). Most of the needed time (69%) for bytecode-based analysis is needed for bytecode verification, instead.
The upfront investment in terms of feature compilation, however, is still apparent in the *avg*-plot of Figure 4.5. The single products are too small to justify bytecode-based analysis for only a small set of products.

4.4 Overall Picture

There are several observations that can be made by analysing the run-time measurements of all product lines of the benchmark set. Consulting Table 4.3 on page 36 and without comparison between the strategies, the bytecode-based and product-based strategies are fastest for Graph and slowest for

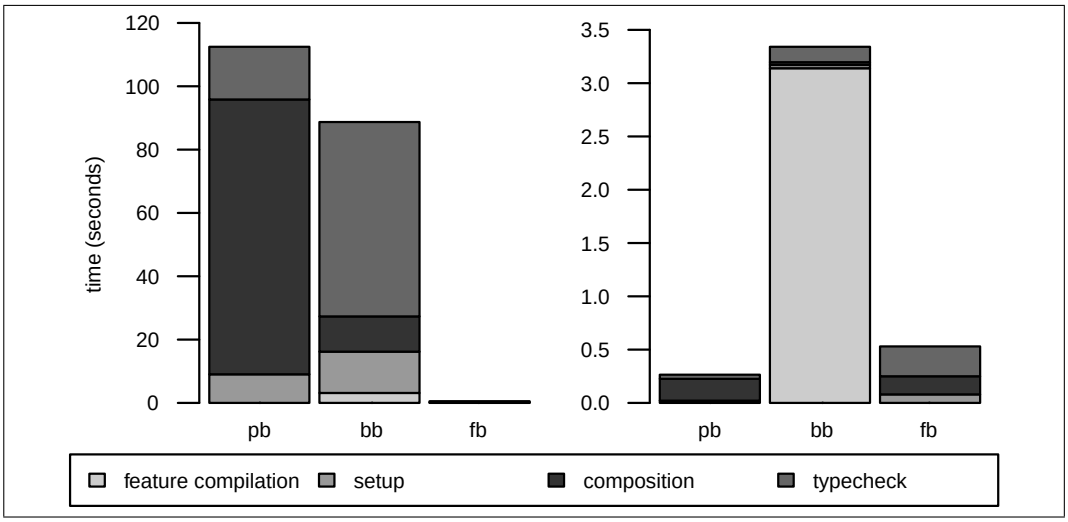

Figure 4.5: Plots for the run-time measurements results (in seconds) of EPL: (left) *sum* diagram, (right) *avg* diagram. (pb = product-based, bb = bytecode-based, fb = family-based)

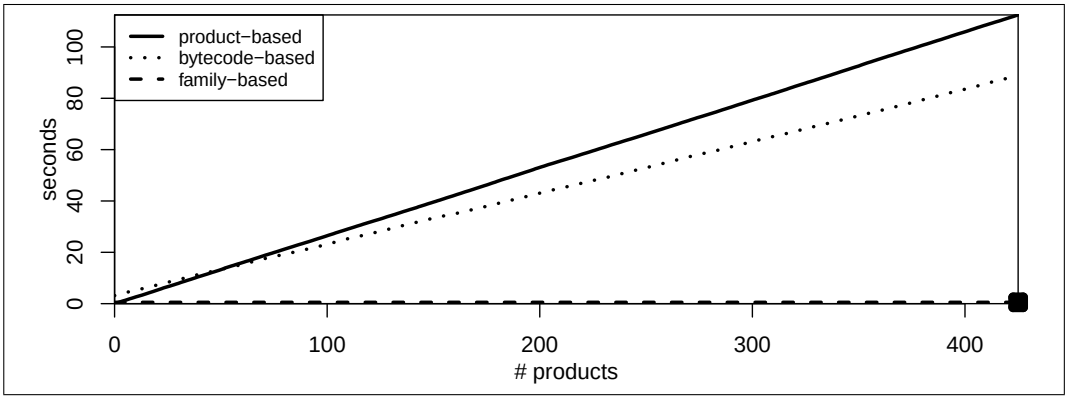

Figure 4.6: Run-time (in seconds) of each strategy by the number of type-checked products for the EPL product line (*cumul* diagram).

Notepad and TankWar. The family-based strategy is fastest for Raroscope and Graph instead and slowest for Violet and TankWar. When comparing the three strategies, the family-based strategy is clearly the winner. It outperforms both other strategies for each product line: the product-based strategy by factor 5 (Violet) to 714 (TankWar), and is about 11 times (Violet) up to 911 times (TankWar) faster than the bytecode-based strategy. (cf. Table 4.1)

Besides the family-based strategy clearly outruns the other two strategies as it is fastest for each product line, the significance is proved using statistical tests. The Shapiro-Wilk test implies rejection of the null hypothesis that the measurement data is normally distributed, so the significant difference of the

benchmark results can be justified with an two-sided Wilcoxon test for which the p-value for each data comparison is less than 0.05.

SPL	**product-based** w.r.t. bytecode-based	**family-based** w.r.t. bytecode-based	**family-based** w.r.t. product-based
EPL	0.79	167.38	212.21
GPL	1.15	51.63	45.02
Graph	1.11	13.09	11.84
GUIDSL	1.72	15.26	8.85
Notepad	1.06	215.99	202.84
PKJab	1.49	33.74	22.70
Prevayler	1.39	25.78	18.58
Raroscope	1.24	16.49	13.26
Sudoku	1.21	49.02	40.47
TankWar	1.28	911.14	714.10
Violet	2.06	11.07	5.38
ZipMe	1.67	25.00	14.97

Table 4.1: Speedups for the product-based strategy with respect to the bytecode-based strategy and speedups of the family-based strategy with respect to both other strategies.

Product-Based Strategy

A product-based type-checker has to parse, compose and type-check repeatedly. This leads to poor scalability. For example, the read-in of files of the code influences the performance badly, because the read-in is performed redundantly for some or even all files. This fact becomes obvious when comparing the corresponding measurement result for setup and composition of the product-based and family-based strategies: Although both type-checkers use Fuji as basis, the family-based time intervals are extremely shorter. Using these product-based—or just partly product-based—type-checkers induce too much redundant work to perform on huge product lines efficiently, so that the Violet product line could not be checked completely at all. Although, the number of products is no indicator for performance estimation. For example, GUIDSL and ZipMe have the same number of products (exactly 24), but performance of the product-based type-checker is slower for GUIDSL. The reason for the slow performance is that this product line is almost thrice as big in terms of lines of code as ZipMe and consists of more feature modules. For product lines with a similar number of features and products as well as a similiar size of the code base (such as Graph and Raroscope), the needed time for type-checking is nearly the same.

Bytecode-Based Strategy

The bytecode-based analysis actually incorporates even more redundancy than the plain product-based strategy, because the results from the feature-based type-check during feature compilation are ignored. As a result, the bytecode-based type-checking implementation is the slowest of all three, the product-based implemenation is 1.06 (Notepad) up to 2.06 times faster (Violet). (cf. Table 4.1)

The only exception here is the EPL product line for which the bytecode-based strategy outperforms the product-based one (cf. Section 4.3), because the bytecode verification is not that expensive for this very small product line (126 LOC). As recognizable for Raroscope (438 LOC) and Graph (596 LOC), this effect even amortizes for slightly bigger product lines, although the bytecode-based and the product-based strategy are nearly equally fast for these small product lines. In total, bytecode verification is the most expensive part of the bytecode-based implementation as it makes up 46% (Violet) up to 82% (TankWar) of the whole run-time. (cf. Table A.2)

In contrast to the bytecode verification, FeatureBite accomplishes composition and setup faster than Fuji. Nevertheless, feature compilation potentially takes longer with increasing number of features and bytecode verification involves additional checks such as data-flow sanity-checks. With higher numbers of products to be generated and type-checked, however, the relative amount of time needed for feature compilation in the bytecode-based type-checking process decreases. But even for SPLs with a small number of features (Graph, Prevayler, Raroscope, Sudoku), the feature compilation still takes more time than the whole family-based process to analyze the whole product line. For product lines with a similar size (number of features and products as well as size of code), the bytecode-based strategy needs a similar time for type-checking—such as the product-based strategy.

Family-Based Strategy

The family-based type-checker is the fastest, because it parses the code and the feature model only once and needs exactly one type-checking run. The advantage of no redundancy within the family-based analysis is clearly visible on the example of Violet which has 89 independent features and accordingly about 2^{89} valid products. The repeatedly performed type-checking analysis of the other strategies cannot be done in reasonable time. The overhead for repetitive parsing and composition grows with the number of valid products. Even for 40 random products, the family-strategy—which checks the whole product line and not just 40 products—is 5.38 times faster than the product-based one and 11.07 times faster than the bytecode-based. The feature compilation for 89 features alone takes 32.84 seconds and is 4.5 times

slower than the Family-Based Type-Checker needs for the whole type-checking process of the complete product line.
The needed time for the analysis part of type-checking makes up around 42% (Raroscope) up to 91% (Violet) of the whole run-time, but the real run-times are extremely shorter than for the bytecode-based analysis where the relative amount is similarly high, but type-checking times become critically long.

As Violet indicates, the variability-aware family-based analysis of large product lines clearly outperforms even very limited sampling approaches in terms of analysis time: the break-even for the family-based analysis (see Table 4.2) is at very low numbers of products for each product line.[15] Even, product lines with a small number of features and a small code base (Graph, Raroscope) as well do not doubt the outranking performance of the family-based strategy.

name	**#p**	**break-even**	**name**	**#p**	**break-even**
EPL	425	2	Prevayler	32	2
GPL	156	4	Raroscope	16	2
Graph	16	2	Sudoku	64	2
GUIDSL	24	3	TankWar	2,458	4
Notepad	512	3	Violet	$\sim 2^{89}$ (40)	8
PKJab	48	3	ZipMe	24	2

Table 4.2: Break-even points for the family-based type-checking strategy with respect to both other strategies. (#p = number of products)

4.5 Which Strategy to Choose

As the results of the benchmark show, the implementation of the *family-based type-checking strategy* is the fastest one for checking the complete product line and its derivated products. It is able to provide useful error messages that enable the user to track down errors even in the feature model. [KvHA13] The other two type-checker implementations do not provide such granularity as they are not able to blame the erroneous feature causing an error due to the missing compile-time variability. The only exception here are the feature-local errors that can be detected during feature compilation. As the break-even for each product line—when family-based type-checking becomes faster than both product- and bytecode-based strategies—is low with respect to the number of features per SPL, a family-based type-checking analysis is always preferable.

[15] That was also found in previous studies. [LvK+12]

When the developer of an SPL wants to check on certain feature interactions and resulting errors, application of a *sampling or product-based strategy* is appropriate on basis of a small set of products to be checked. This way, the very feature-interactions can be examined and evaluated. Also, if standard type-checkers have to be used, a product-based or sampling strategy is applicable.

Furthermore, during development of a new feature, a *feature-based type-check* is advisable, because it detects feature-local errors as soon as possible in the development process, although features may only consist of class refinements and no fully type-safe classes.

If all features and their corresponding stubs already exist in bytecode, a *bytecode-based analysis* is preferable, because it also checks for more than type-safety, especially for a sane data-flow and correctness of bytecode to reduce vulnerability of the run-time system. Other studies show that there are product-based type-checkers that are slower regarding performance than the chosen implementation and even type-checking using FeatureStubber and FeatureBite. [KvHA13] In this case, the bytecode-based analysis is preferable if the family-based one cannot be considered for some reason—such as sampling-based type-checking. Although, it may be a major drawback due to the amount of time that feature compilation and bytecode verification take for a higher number of features or huge and complex features, respectively. The identified challenges (cf. Section 3.3 and Section 3.3.2, especially) may also add to the amount of work for feature compilation.

Based on the tools used and evaluated in this thesis, the Family-Based Type-Checker is the most preferable due to its advantages of short run-time and accurate error detection. Both Fuji and FeatureBite are the runner-ups that are favored in context of sampling, depending on the requirements and goals of the particular analysis.

4.6 Threats to Validity

The choice of tools that were used as a representative of a particular strategy threatens the internal validity. Other tools may have needed shorter or respectively longer run-times for the same product line and its products. (cf. [KvHA13]) Particularly, the Family-Based Type-Checker does not implement all type checks that are necessary for the Java language, but only a representative part—in contrast to FeatureBite which performs more checks than necessary. It is safe to assume that the overall picture will not be influenced much by new type-checking rules for the Family-Based Type-Checker as this tool is so much faster than the other implementations in this thesis and has several more advantages, such as direct blaming of an

erroneous feature.
The external validity is threatened by the choice of product lines. The samples, all implemented using FOP, may only consist of language constructs and patterns or large feature modules that are not beneficial for certain strategies. Also, no conclusions about other product line implementation techniques such as C preprocessor or aspect-oriented programming (AOP) can be made.

SPL	product-based				bytecode-based					family-based			
	setup	**comp**	**check**	**sum**	**fc**	**setup**	**comp**	**check**	**sum**	**setup**	**comp**	**check**	**sum**
EPL	9.03	86.80	16.64	112.47	3.14	13.05	11.13	61.39	88.71	0.08	0.17	0.28	0.53
GPL	4.20	45.61	19.07	68.88	6.82	7.17	14.54	50.47	79.00	0.09	0.29	1.15	1.53
Graph	0.30	3.62	1.17	5.09	1.46	0.41	0.52	3.24	5.63	0.06	0.19	0.18	0.43
GUIDSL	0.94	13.15	14.77	28.86	12.94	3.33	5.07	28.40	49.74	0.10	0.49	2.67	3.26
Notepad	10.82	166.03	135.52	312.37	6.68	18.11	39.45	268.39	332.63	0.08	0.26	1.20	1.54
PKJab	1.86	21.11	23.33	46.30	4.31	3.11	5.97	55.44	68.83	0.09	0.35	1.60	2.04
Prevayler	1.24	13.14	21.66	36.04	3.58	3.13	5.51	37.79	50.01	0.09	0.42	1.43	1.94
Raroscope	0.31	4.04	1.35	5.70	1.55	0.51	0.63	4.40	7.09	0.05	0.20	0.18	0.43
Sudoku	1.25	24.34	17.71	43.30	3.73	2.67	5.34	40.71	52.45	0.05	0.29	0.73	1.07
TankWar	56.44	853.11	790.01	1,699.56	11.93	107.96	276.20	1,772.42	2,168.51	0.07	0.30	2.01	2.38
Violet*	2.03	16.42	20.25	38.70	33.40	3.46	5.87	36.85	79.58	0.22	0.42	6.55	7.19
ZipMe	0.64	8.12	7.86	16.62	4.48	1.22	2.80	19.25	27.75	0.09	0.29	0.73	1.11

Table 4.3: Total run-time in seconds for all products of each benchmark product lines by type-checking strategy. (measurement intervals: setup = setup, comp = composition, check = type check, fc = feature compilation, sum = sum of all intervals; * only 40 products checked for VIOLET, see Section 2.3 for details on this.)

5. Related Work

The classification of the product-line analysis strategies that is used in this thesis was published in a survey by Thüm et al. [TAK+12]. The authors already discussed advantages and disadvantages of each strategy as presented in Section 2.1. The authors also presented a set of mixed strategies such as the feature-product-based one which is the idea behind the bytecode-based analysis evaluated in this thesis. Von Rhein et al. presented the Product-Line-Analysis model [vRAK+13] on top of this classification of analysis strategies. This model is able to represent the full set of possible combinations of product-line analysis strategies and helps searching for an optimal strategy.
The single strategies were applied in many studies with different analysis techniques before, such as type checking, static analysis and model checking.

The *family-based* strategy was applied by Kästner et al. [KGR+11] on parsing preprocessor-annotated C and Java programs using the TypeChef framework. The authors parsed the Java-ME-based MobileMedia product line and the entire X86 architecture of the Linux kernel (6065 features). They generated a variability-aware AST—much like Fuji and the Family-Based Type-Checker are able to do for FOP product lines.
Safe composition was proven for Lightweight Feature Java by Delaware et al. [DCB09] They presented a type system to ensure safe composition for all combinations of features that satisfy the typing rules. Kästner et al. [KATS12] extended the Featherweight Java calculus with feature annotations. They proved formally that all program variants produced from a well-typed product line are well-typed, too. On top of their formal achievements, they implemented CIDE, a tool to virtually separate features within an IDE. Thaker et al. [TBKC07] showed on the example of AHEAD product lines how to guarantee safe composition using feature models and SAT solvers. They first compiled the feature modules by adding the union of all fields, methods and

declarations that can appear in a class as stubs. Then, they inferred composition constraints for each feature module that are implied by the module's "requires-and-provides interface". This is generally similar to the way FeatureStubber works on top of the introduction- and references-files provided by Fuji. (cf. Section 3.1)
Bodden et al. [BMB+13] contributed a tool to statically analyze product lines by supplying traditional program analyses that were converted to proper product-line-aware analyses automatically in the process. They used a tool named SPL$^{\text{LIFT}}$ to accomplish this. So that developers do not have to generate and analyze products individually, Braband et al. [BRT+13] presented three ways to take any standard intraprocedural data-flow analysis and converted it into a feature-aware data-flow analysis. Liebig et al. [LvK+12] compared a variability-aware type-checking and data-flow analysis with a sampling approach for product lines written in C and annotated with preprocessor directives. They found the variability-aware analysis outranks the sampling heuristics with respect to analysis time. (cf. Section 4.4)
There is also related work for family-based analysis in terms of model checking [LTP09][CHS+10][ASW+11][AvRW+13] and deductive verification [TSAH12].

In the context of *product-based* analyses with sampling, Johansen et al. suggested an algorithm called ICPL which generates covering arrays from feature models. [JHF12] With this non-deterministic algorithm, they were able to quickly generate small covering arrays. Oster et al. [OMR10] suggested a way to apply combinatorial testing to a feature model of a SPL.
Siegmund et al. [SRK+13] approximated non-functional properties by measuring a small set of products and predict the actual property for a given product configuration. They evaluated their approach for the non-functional property "footprint" and accomplished an accuracy of 98% for their prediction.

The *feature-product-based* strategy was used by Apel and Hutchins [AH10] regarding type checking. The authors propose with *g*Deep a core calculus that enables the generation of interfaces after a feature-based type-check and performs product-based linking analyses on valid compositions of these interfaces. Schaefer et al. [BDS13] provided a core calculus for delta-oriented programming that is the foundation for compositional type-checking of delta-oriented product lines. The delta modules are type-checked in isolation and the combined results can be used to reason about all products and their well-typedness regarding the Java type system.
Li et al. [LKF02] described requirements for verifying feature modules through model checking by providing a new methodology for verification. They validated their work by application to feature modules inheriting feature interaction problems. The authors also presented a model of interfaces to verify features independently and to support automated feature-based model checking. [LKF05]

Delaware et al. [DCB11] proved type-safety of a product-line of languages (written in Featherweight Java) in a feature-product-based manner. Theorems were created and proved for each feature first; then, these theorems helped proving preservation and progress for the product line.

Regarding feature composition, the AHEAD tool suite [BSR04] and FeatureHouse [AKL09] have to be referenced apart from Fuji [Kol11]. The authors of AHEAD presented a model for FOP and showed how both code and non-code fragments of programs can be composed using algebraic models by retaining hierarchical structure. They also provided bytecode tools[16] that are capable of composing AHEAD-based feature modules as is FeatureBite. These tools use BCEL as underlying bytecode framework for composition instead of ASM like FeatureBite[17]. A feature-stub generator utility is also provided that works similar to FeatureStubber. FeatureHouse is another approach to the composition of software artifacts. It is language-independent, so that, for example, source code, models and even documentation can be composed.

The differences of this thesis to the work of Kolesnikov et al. [KvHA13] primarily lie in the choice of tools while the same sample product lines were used. The researchers integrated all type-checkers in one tool that was based on Fuji. The Family-Based Type-Checker is integrated in the same version as it was used in this thesis. The product-based type-checker was reimplemented to conform with the family-based type-checking algorithm. Also, a pure feature-based type-checker was implemented that does not depends on FeatureStubber. By using these tools, the product-based analysis took longest for type-checking most of the product lines. Also the feature-product-based analysis that was applied (also using FeatureBite) is mostly faster than the pure product-based.

[16] http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/ByteCodeTools.html
[17] BCEL is only used for verification purposes. See Section 3.4 for details.

6. Conclusion

In this thesis, we compared three type-checking strategies in terms of performance: the product-based, the bytecode-based and the family-based strategy. For evaluation, we conducted one type-checker tool per strategy on a set of 12 Java-based, feature-oriented product lines in a controlled setting in order to measure the needed run-time of each tool. The type-checkers were Fuji for the product-based strategy, FeatureBite for the bytecode-based one and Family-Based Type-Checker for the latter strategy.
The comparison emphasized the family-based strategy as the fastest one in terms of performance of type-checking, not even sampling is able to outperform this strategy. (cf. Section 4.4) This strategy is up to 714 times faster than the product-based one and up to even 911 times faster than the bytecode-based approach. The bytecode-based strategy implemented by FeatureBite is the slowest approach.

During the benchmarking, several advantages and drawbacks were identified for each strategy:
Both the product-based strategy as well as the bytecode-based strategy scale poorly, because the feature modules have to be parsed and composed repeatedly, in contrast to the family-based strategy which performs the needed actions only once. However, this strategy has to repeat the whole analysis when the features' code base is changed, the other strategies are able to repeat the analysis only for affected products. Apart from that, the bytecode-based strategy is yet for product lines with a small number of features slower than the product-based strategy because of the expensive bytecode verification, although the setup and the composition analysis parts are faster than for the product-based strategy. The bytecode-based strategy also exhibits critical problems for feature composition. Nevertheless, this strategy may be appropriate if special sanity checks for security reasons are necessary such as those

performed during bytecode verification. In the end, the family-based strategy is not only the fastest one, it also supplies the user with the most comprehensive error messages [KvHA13] that enables the developer to track down the error accordingly, because it is able to incorporate the variability into the analysis.

The bytecode-based strategy could be expanded in future in order to transform it to an actual feature-product-based strategy. Accordingly, the bytecode verification could be optimized in this way because results from the feature-based part of the analysis were handed over to the product-based part to save effort for double checks and, potentially, unnecessary security routines. Moreover, a more detailed statistical evaluation could support the results of this thesis.

List of Abbreviations

AOP aspect-oriented programming

API application programming interface

AST abstract syntax tree

FOP feature-oriented programming

GB gigabyte

GHz gigahertz (10^9 Hertz)

IDE integrated development environment

JVM Java Virtual Machine

LOC lines of code

SPL software product line

TB terabyte

List of Figures

List of Tables

A. Appendix

A.1 Run-Time Measurements

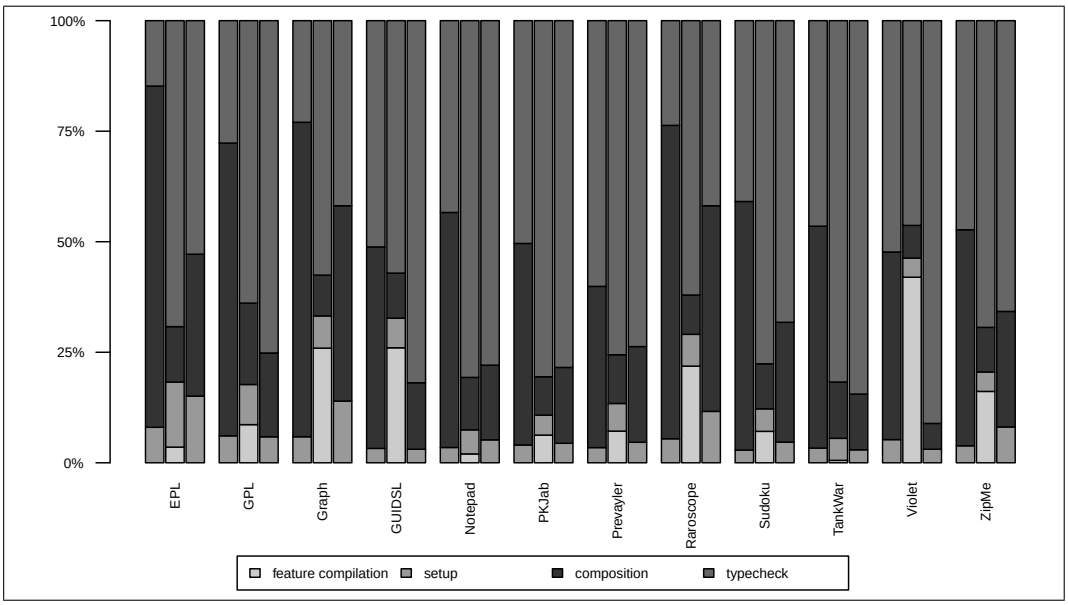

Figure A.1: Visualization of Table A.2: Relative run-time for each measured time interval by type-checking strategy.

SPL	product-based				bytecode-based					family-based			
	setup	**comp**	**check**	**sum**	**fc**	**setup**	**comp**	**check**	**sum**	**setup**	**comp**	**check**	**sum**
EPL	0.02	0.20	0.04	0.26	3.14	0.03	0.03	0.14	3.34	0.08	0.17	0.28	0.53
GPL	0.03	0.29	0.12	0.44	6.82	0.05	0.09	0.32	7.28	0.09	0.29	1.15	1.53
Graph	0.02	0.23	0.07	0.32	1.46	0.03	0.03	0.20	1.72	0.06	0.19	0.18	0.43
GUIDSL	0.04	0.55	0.62	1.20	12.94	0.14	0.21	1.18	14.47	0.10	0.49	2.67	3.26
Notepad	0.02	0.32	0.26	0.61	6.68	0.04	0.08	0.52	7.32	0.08	0.26	1.20	1.54
PKJab	0.04	0.44	0.49	0.96	4.31	0.06	0.12	1.16	5.65	0.09	0.35	1.60	2.04
Prevayler	0.04	0.41	0.68	1.13	3.58	0.10	0.17	1.18	5.03	0.09	0.42	1.43	1.94
Raroscope	0.02	0.25	0.08	0.36	1.55	0.03	0.04	0.28	1.90	0.05	0.20	0.18	0.43
Sudoku	0.02	0.38	0.28	0.68	3.73	0.04	0.08	0.64	4.49	0.05	0.29	0.73	1.07
TankWar	0.02	0.35	0.32	0.69	11.93	0.04	0.11	0.72	12.81	0.07	0.30	2.01	2.38
Violet*	0.05	0.41	0.51	0.97	33.40	0.09	0.15	0.92	34.55	0.22	0.42	6.55	7.19
ZipMe	0.03	0.34	0.33	0.69	4.48	0.05	0.12	0.80	5.45	0.09	0.29	0.73	1.11

Table A.1: Run-time in seconds for an average product of each benchmark product lines by type-checking strategy. (measurement intervals: setup = setup, comp = composition, check = type check, fc = feature compilation, sum = sum of all intervals; * only 40 products checked for VIOLET, see Section 2.3 for details on this.)

SPL	product-based			bytecode-based				family-based		
	setup	**comp**	**check**	**fc**	**setup**	**comp**	**check**	**setup**	**comp**	**check**
EPL	8%	77%	15%	4%	15%	13%	69%	15%	32%	53%
GPL	6%	66%	28%	9%	9%	18%	64%	6%	19%	75%
Graph	6%	71%	23%	26%	7%	9%	58%	14%	44%	42%
GUIDSL	3%	46%	51%	26%	7%	10%	57%	3%	15%	82%
Notepad	3%	53%	43%	2%	5%	12%	81%	5%	17%	78%
PKJab	4%	46%	50%	6%	5%	9%	81%	4%	17%	78%
Prevayler	3%	36%	60%	7%	6%	11%	76%	5%	22%	74%
Raroscope	5%	71%	24%	22%	7%	9%	62%	12%	47%	42%
Sudoku	3%	56%	41%	7%	5%	10%	78%	5%	27%	68%
TankWar	3%	50%	46%	1%	5%	13%	82%	3%	13%	84%
Violet*	5%	42%	52%	42%	4%	7%	46%	3%	6%	91%
ZipMe	4%	49%	47%	16%	4%	10%	69%	8%	26%	66%

Table A.2: Relative run-time for each measured time interval by type-checking strategy.
(measurement intervals: setup = setup, comp = composition, check = type check, fc = feature compilation, sum = sum of all intervals; * only 40 products checked for VIOLET, see Section 2.3 for details on this.)

A.2 Plots for the Other Product Lines

A.2.1 GPL

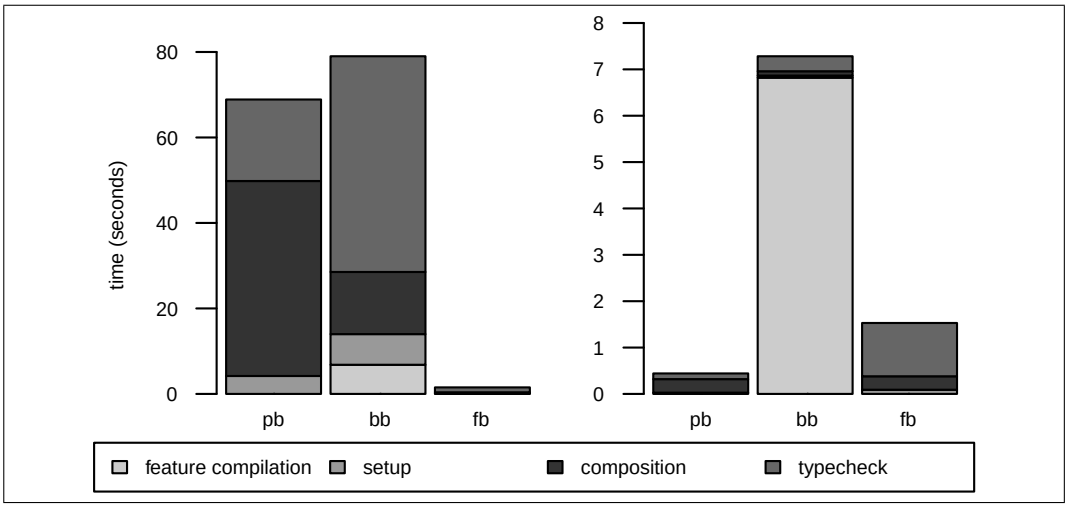

Figure A.2: Plots for the run-time measurements results (in seconds) of GPL: (left) *sum* diagram, (right) *avg* diagram. (pb = product-based, bb = bytecode-based, fb = family-based)

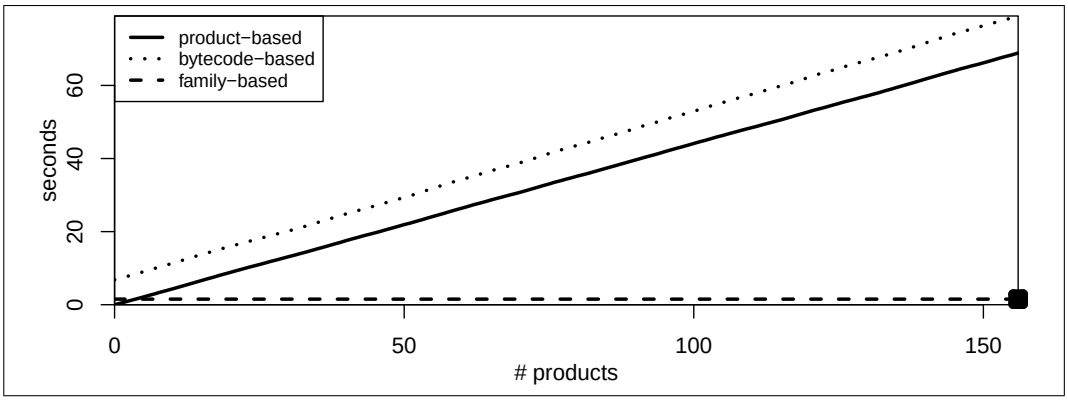

Figure A.3: Run-time (in seconds) of each strategy by the number of type-checked products for the GPL product line (*cumul* diagram).

A.2.2 Graph

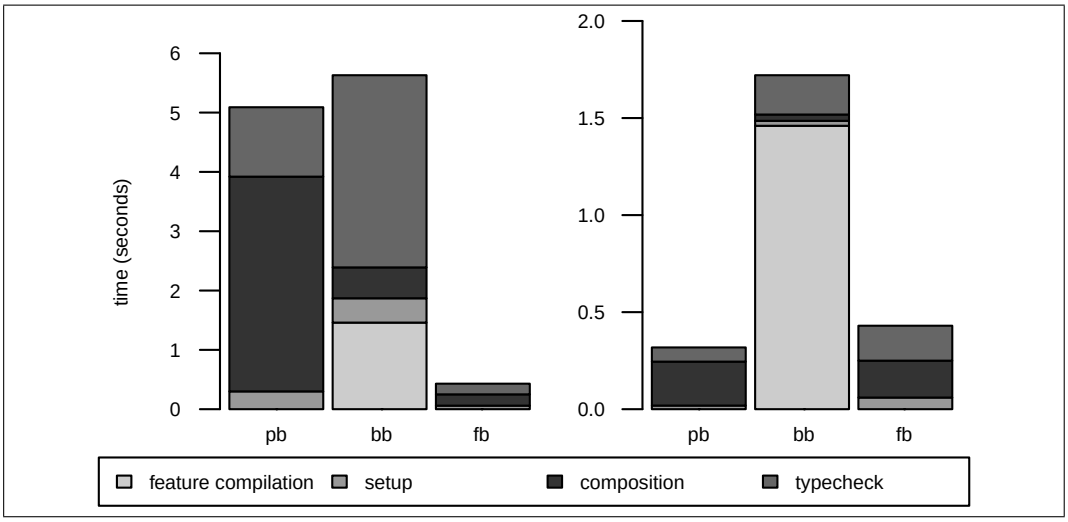

Figure A.4: Plots for the run-time measurements results (in seconds) of GRAPH: (left) *sum* diagram, (right) *avg* diagram. (pb = product-based, bb = bytecode-based, fb = family-based)

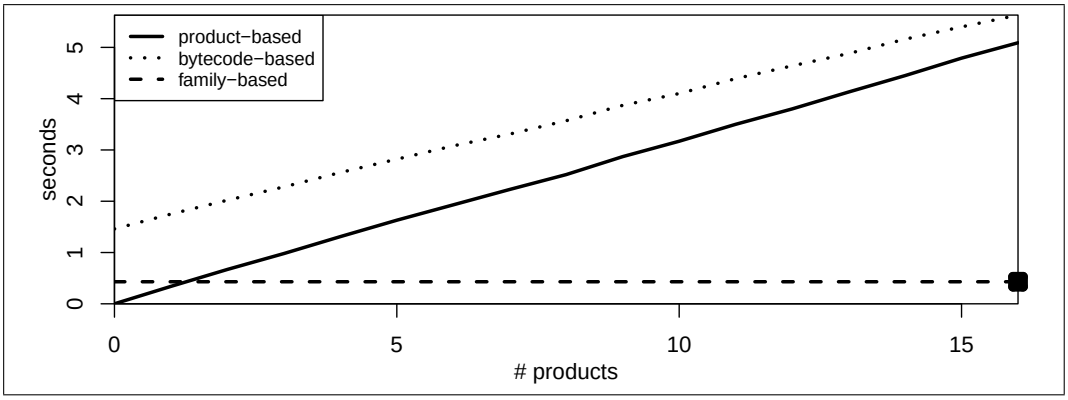

Figure A.5: Run-time (in seconds) of each strategy by the number of type-checked products for the GRAPH product line (*cumul* diagram).

A.2.3 Notepad

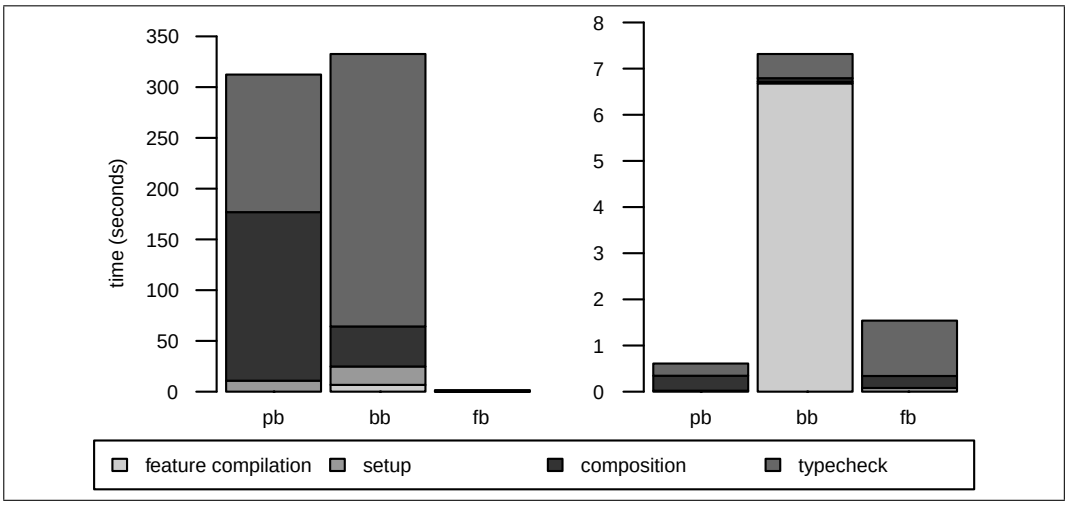

Figure A.6: Plots for the run-time measurements results (in seconds) of NOTEPAD: (left) *sum* diagram, (right) *avg* diagram. (pb = product-based, bb = bytecode-based, fb = family-based)

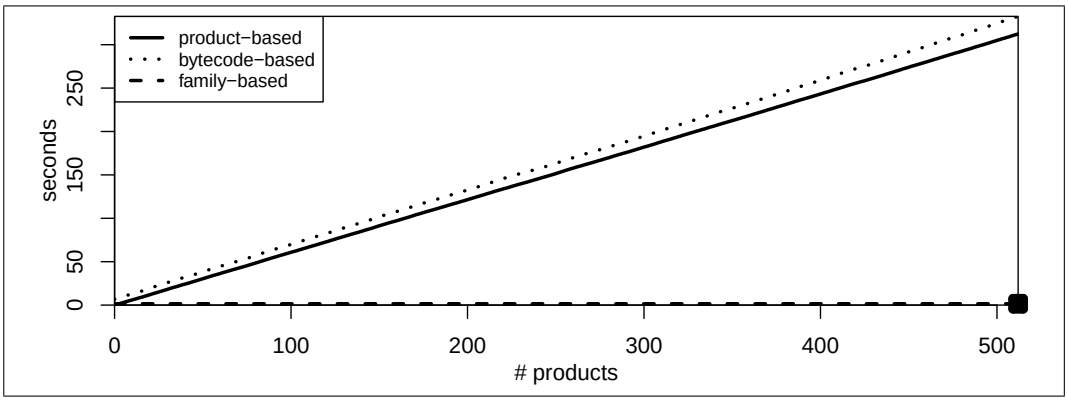

Figure A.7: Run-time (in seconds) of each strategy by the number of type-checked products for the NOTEPAD product line (*cumul* diagram).

A.2.4 PKJab

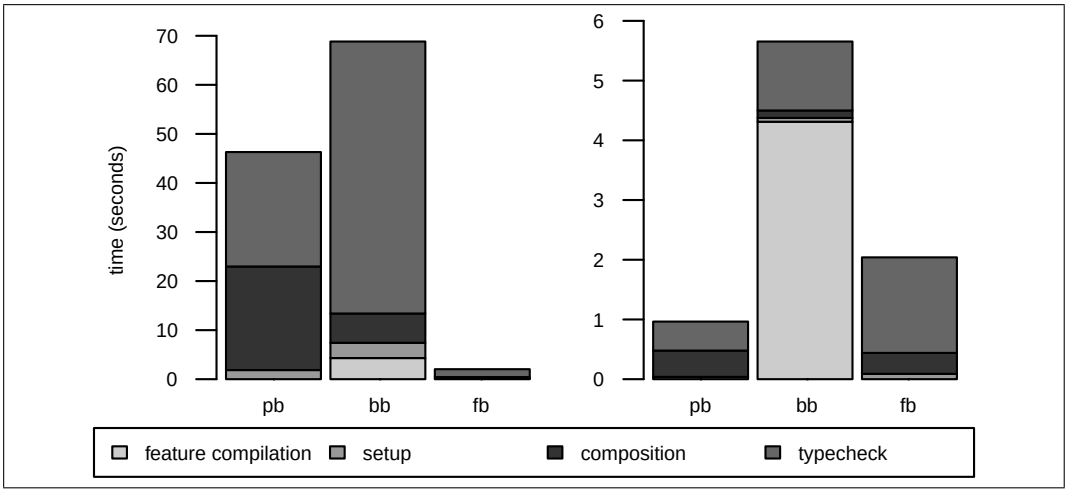

Figure A.8: Plots for the run-time measurements results (in seconds) of PKJab: (left) *sum* diagram, (right) *avg* diagram. (pb = product-based, bb = bytecode-based, fb = family-based)

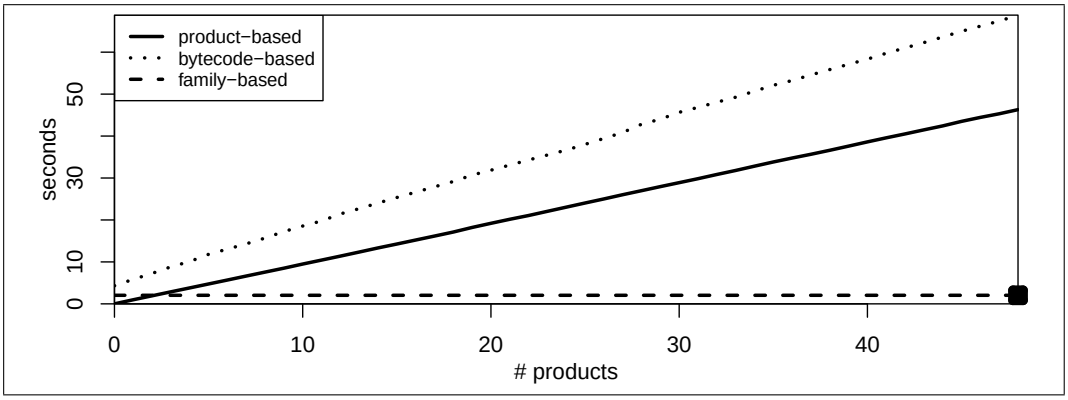

Figure A.9: Run-time (in seconds) of each strategy by the number of type-checked products for the PKJab product line (*cumul* diagram).

A.2.5 Raroscope

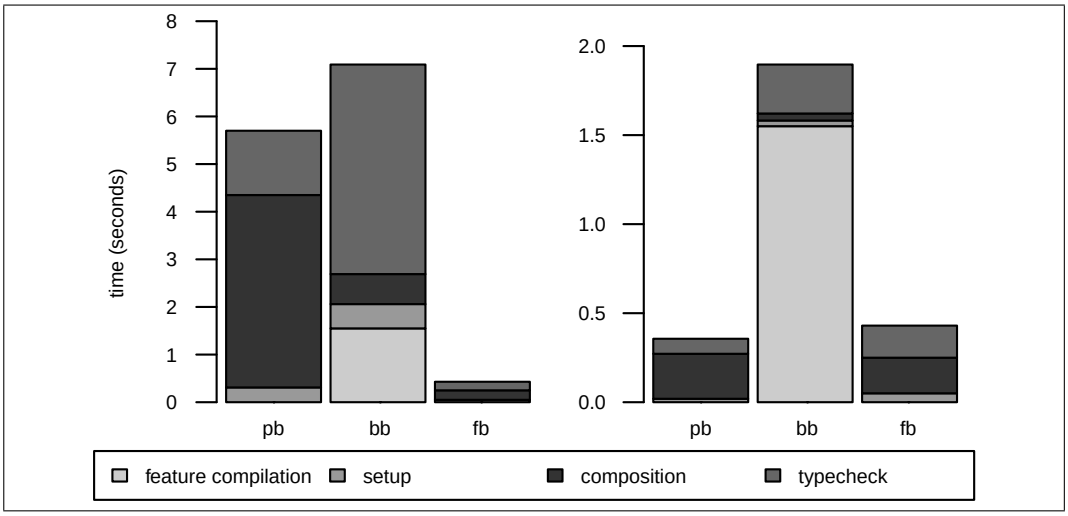

Figure A.10: Plots for the run-time measurements results (in seconds) of Raroscope: (left) *sum* diagram, (right) *avg* diagram. (pb = product-based, bb = bytecode-based, fb = family-based)

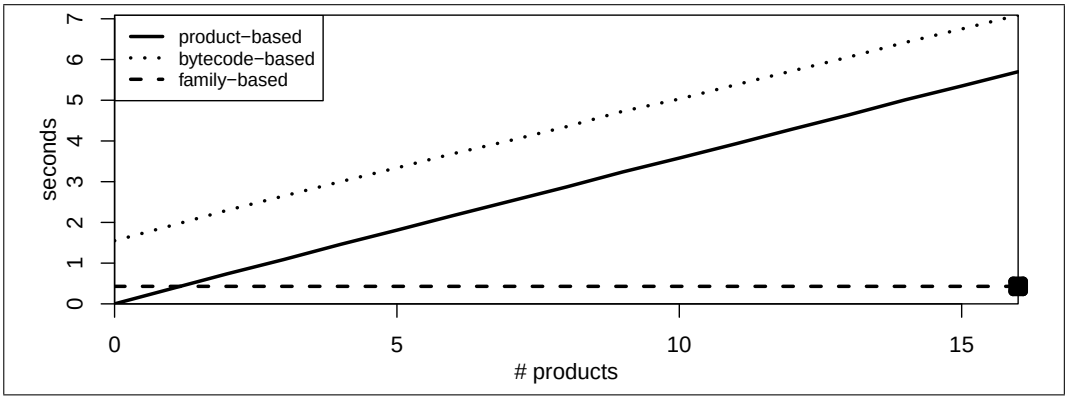

Figure A.11: Run-time (in seconds) of each strategy by the number of type-checked products for the Raroscope product line (*cumul* diagram).

A.2.6 Sudoku

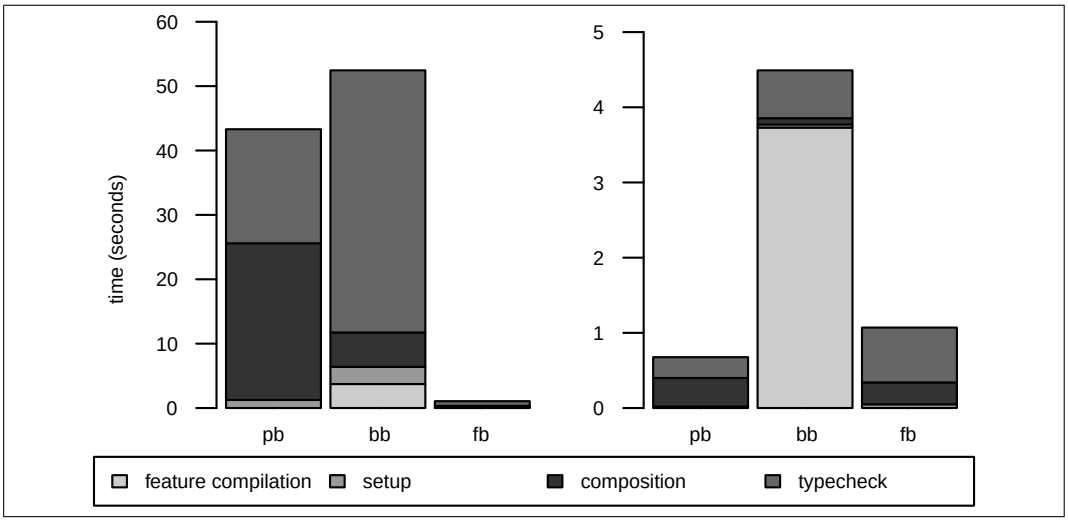

Figure A.12: Plots for the run-time measurements results (in seconds) of SUDOKU: (left) *sum* diagram, (right) *avg* diagram. (pb = product-based, bb = bytecode-based, fb = family-based)

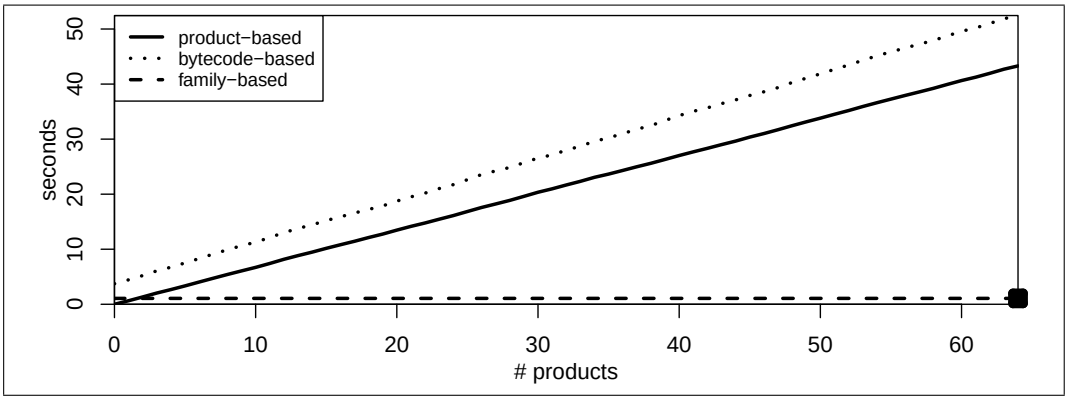

Figure A.13: Run-time (in seconds) of each strategy by the number of type-checked products for the SUDOKU product line (*cumul* diagram).

A.2.7 TankWar

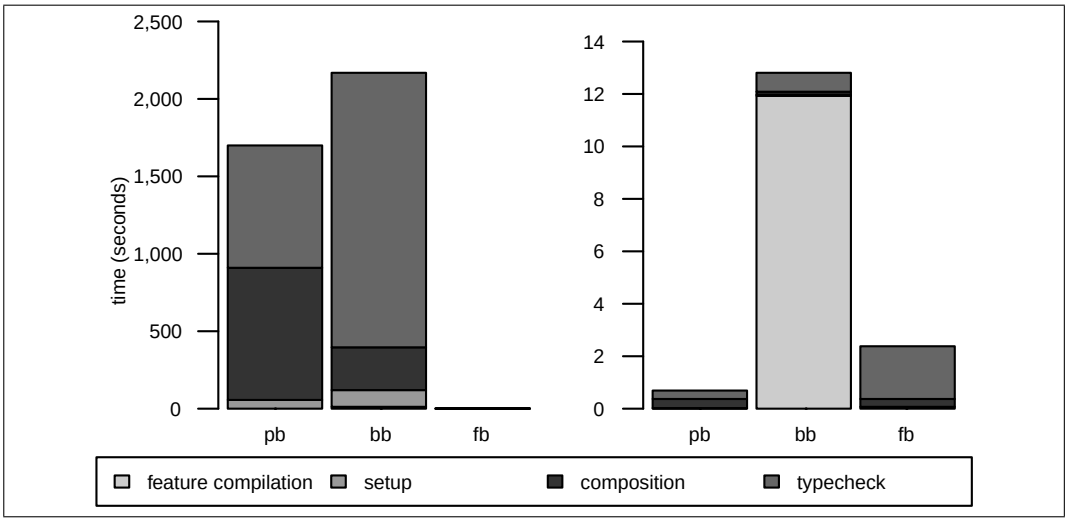

Figure A.14: Plots for the run-time measurements results (in seconds) of TankWar: (left) *sum* diagram, (right) *avg* diagram. (pb = product-based, bb = bytecode-based, fb = family-based)

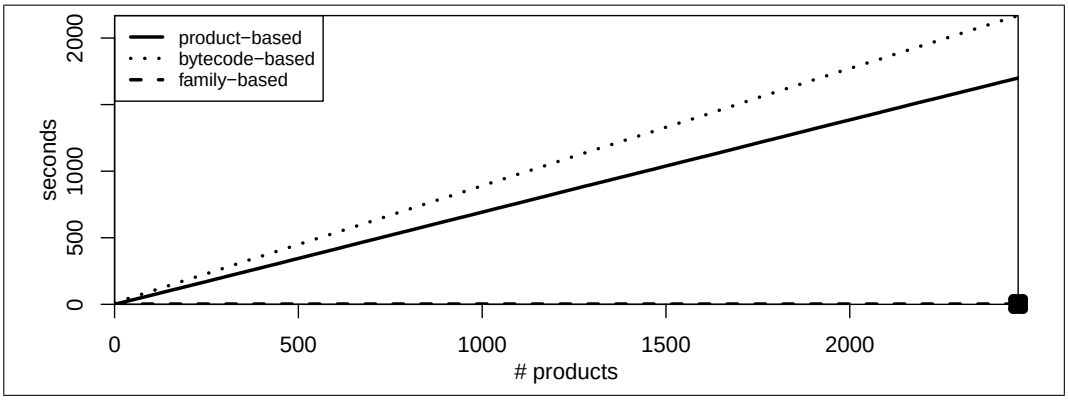

Figure A.15: Run-time (in seconds) of each strategy by the number of type-checked products for the TankWar product line (*cumul* diagram).

A.2.8 Violet

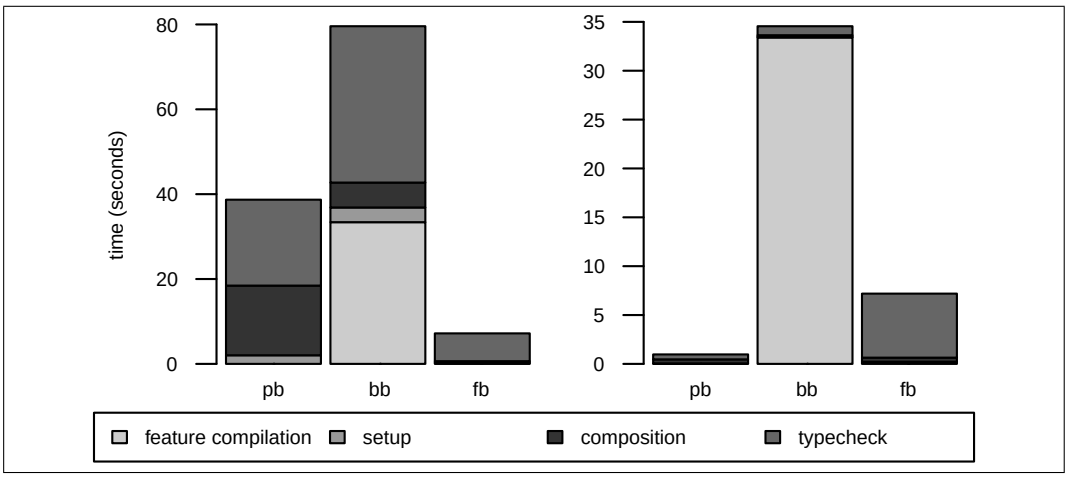

Figure A.16: Plots for the run-time measurements results (in seconds) of VIOLET: (left) *sum* diagram, (right) *avg* diagram. (pb = product-based, bb = bytecode-based, fb = family-based)

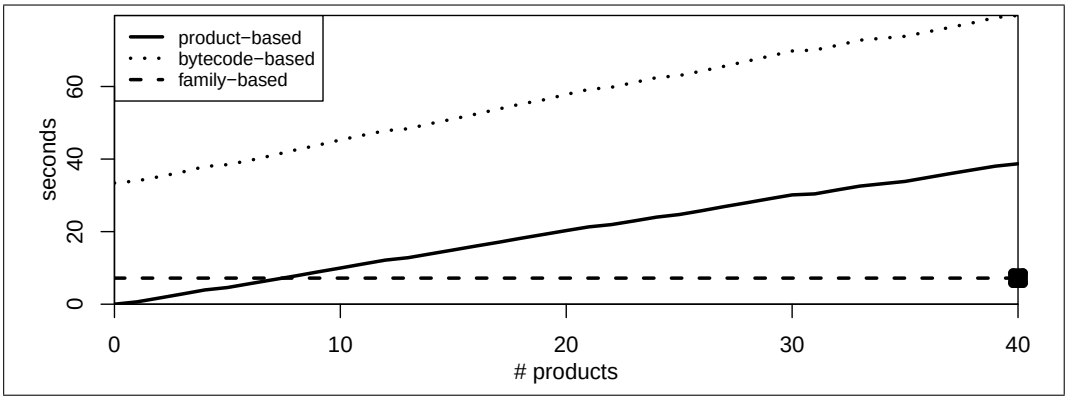

Figure A.17: Run-time (in seconds) of each strategy by the number of type-checked products for the VIOLET product line (*cumul* diagram).

A.2.9 ZipMe

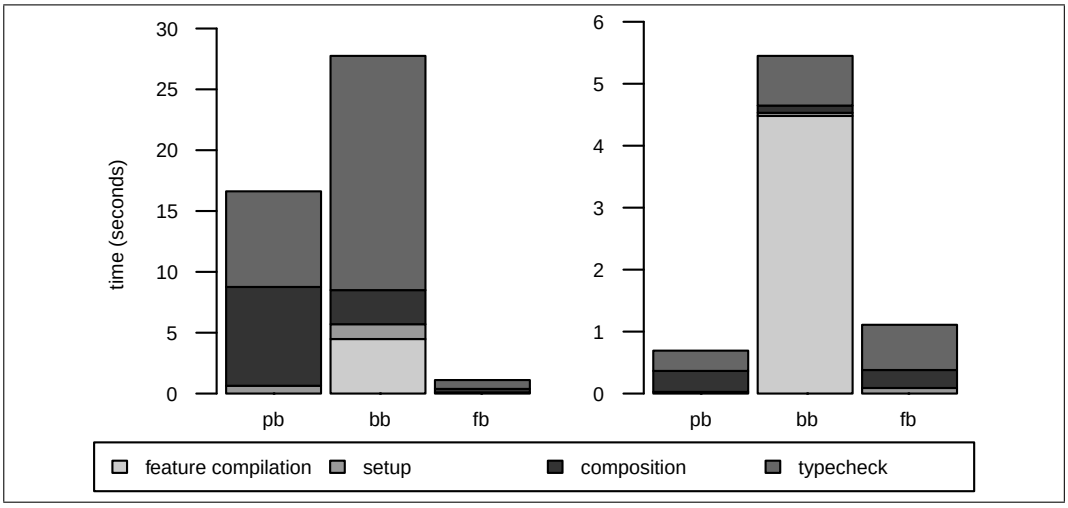

Figure A.18: Plots for the run-time measurements results (in seconds) of ZipMe: (left) *sum* diagram, (right) *avg* diagram. (pb = product-based, bb = bytecode-based, fb = family-based)

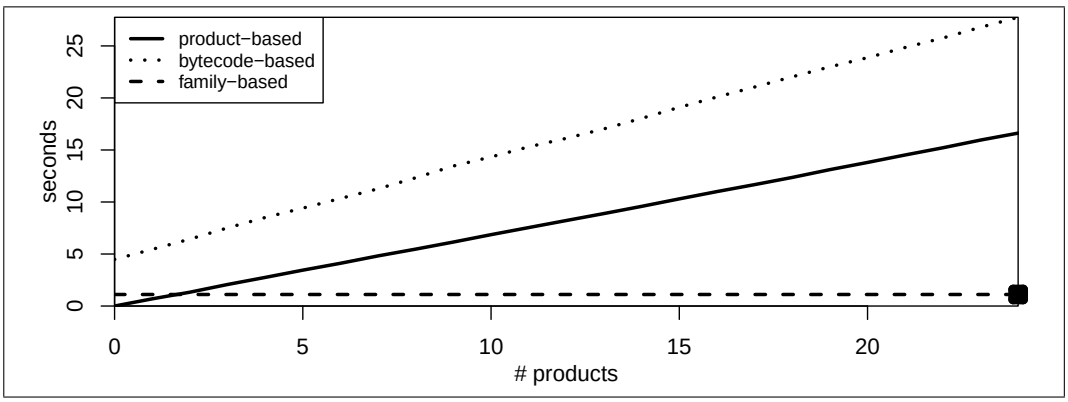

Figure A.19: Run-time (in seconds) of each strategy by the number of type-checked products for the ZipMe product line (*cumul* diagram).

Bibliography

[AB11] S. Apel and D. Beyer. Feature cohesion in software product lines: An exploratory study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 421–430. ACM, 2011.

[AH10] S. Apel and D. Hutchins. A calculus for uniform feature composition. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(5):19:1–19:33, 2010.

[AKGL10] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[AKL09] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE, 2009.

[AKL+12] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich. Access control in feature-oriented programming. *Science of Computer Programming (SCP)*, 77(3):174–187, 2012.

[AKL13] S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering (TSE)*, 39(1):63–79, 2013.

[ASW+11] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 372–375. IEEE, 2011.

[AvRW+13] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 482–491. IEEE, 2013.

[BDS13] L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013.

[BMB+13] E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, and P. Borba. SPLlift - Statically analyzing software product lines in minutes instead of years. In *Proceedings of the International Conference on Programming Languages Design and Implementation (PLDI)*, 2013.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[BRT+13] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, and P. Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development (TAOSD)*, 10:73–108, 2013.

[BSR04] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

[CHS+10] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 335–344. ACM, 2010.

[DCB09] B. Delaware, W. Cook, and D. Batory. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 243–252. ACM, 2009.

[DCB11] DB. Delaware, W. Cook, and D. Batory. Product lines of theorems. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 595–608. ACM, 2011.

[GJSB13] J. Gosling, B. Joy, G. .Steele, and G. Bracha. The Java Language Specification, Third Edition. http://docs.oracle.com/javase/specs/jls/se5.0/html/j3TOC.html, March 2013.

[JHF12] M. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Pro-*

ceedings of the International Software Product Line Conference (SPLC), pages 46–55. ACM, 2012.

[KATS12] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(3):14:1–14:29, 2012.

[KBK09] M. Kuhlemann, D. Batory, and C. Kästner. Safe composition of non-monotonic features. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 177–186. ACM, 2009.

[KGR+11] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 805–824. ACM, 2011.

[Kol11] Sergiy Kolesnikov. An extensible compiler for feature-oriented programming in java. Master's Thesis, Department of Informatics and Mathematics, Passau University, 2011.

[KvHA13] S. Kolesnikov, A. von Rhein, C. Hunsen, and S. Apel. A comparison of product-based, feature-based, and family-based type checking. submitted, June 2013.

[LHB01] R. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, LNCS 2186, pages 10–24. Springer, 2001.

[LKF02] H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 89–98. ACM, 2002.

[LKF05] H. Li, S. Krishnamurthi, and K. Fisler. Modular verification of open features using three-valued model checking. *Automated Software Engineering*, 12(3):349–382, 2005.

[LTP09] K. Lauenroth, S. Toehning, and K. Pohl. Model checking of domain artifacts in product line engineering. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 269–280. IEEE, 2009.

[LvK+12] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Large-scale variability-aware type checking and dataflow analyis. Technical Report MIP-1212, Department of Informatics and Mathematics, University of Passau, November 2012.

[LvRK+13] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proceedings of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013.

[LY13] T. Lindholm and F. Yellin. The Java(TM) Virtual Machine Specification - second edition. http://docs.oracle.com/javase/specs/jvms/se5.0/html/VMSpecTOC.doc.html, March 2013.

[OMR10] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, LNCS 6287, pages 196–210. Springer, 2010.

[Pie02] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[SRK+13] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology*, 55(3):491–507, 2013.

[TAK+12] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis strategies for software product lines. Technical Report FIN-004-2012, University of Magdeburg, 2012.

[TBKC07] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM, 2007.

[TKB+13] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An extensible framework for feature-oriented software development. In *Science of Computer Programming*, 2013. To appear; accepted 2012-06-07.

[TOS02] P. Tarr, H. Ossher, and S. Sutton Jr. Hyper/J: Multi-dimensional separation of concerns for Java. In *Proceedings of*

the International Conference on Software Engineering (ICSE), pages 689–690. ACM, 2002.

[TSAH12] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based deductive verification of software product lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 11–20. ACM, 2012.

[vRAK+13] A. von Rhein, S. Apel, C. Kästner, Thomas Thüm, and I. Schaefer. The PLA model: On the combination of product-line analyses. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 73–80. ACM, 2013.

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Masterarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Claus Hunsen

Passau, den 05. Juli 2013