A Comparison of Product-based, Feature-based, and Family-based Type Checking

Sergiy Kolesnikov Alexander von Rhein Claus Hunsen Sven Apel

University of Passau
Germany

Abstract

Analyzing software product lines is difficult, due to their inherent variability. In the past, several strategies for product-line analysis have been proposed, in particular, *product-based*, *feature-based*, and *family-based* strategies. Despite recent attempts to conceptually and empirically compare different strategies, there is no work that empirically compares all of the three strategies in a controlled setting. We close this gap by extending a compiler for feature-oriented programming with support for product-based, feature-based, and family-based type checking. We present and discuss the results of a comparative performance evaluation that we conducted on a set of 12 feature-oriented, JAVA-based product lines. Most notably, we found that the family-based strategy is superior for all subject product lines: it is substantially faster, it detects all kinds of errors, and provides the most detailed information about them.

Categories and Subject Descriptors D.2.13 [*SOFTWARE ENGINEERING*]: Reusable Software; D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features

Keywords Feature-oriented programming; product-line analysis; type checking; Fuji

1. Introduction

A *feature* is an end-user-visible behavior or characteristic of a product that satisfies a stakeholder's requirement [17]. A *software product line* is a family of related software products that share common features and differ in other features [12]. The product-line approach introduces a further dimension of complexity to software engineering: *variability*. It is this additional dimension that renders existing analysis tools impractical, for example, model checkers, static analyzers, type checkers, and so on [30]. Sure, to analyze a product line, we can use an off-the-shelf analysis tool and apply it to all of its products, which, however, requires exponential analysis effort, due to feature combinatorics (in the worst case, the number of products grows exponentially with the number of features). Alternatively, applying off-the-shelf tools to the variable code base of a product line itself (e.g., consisting of unprocessed C code, feature modules, or aspects) is often impossible. However, a solution is to make the tools *variability-aware* [30].

Variability-aware analysis techniques that can be applied to the variable code base of a product line are called *family-based*. They take advantage of the inherent variability of a product line and can deliver sound and complete analysis results. However, they are often computationally expensive, compared to standard analyses.

To handle computational complexity, *feature-based* analyses operate on the implementation of individual features, without considering interactions across feature boundaries. While feature-based analyses are fast, they are incomplete, for example, in that they cannot catch bugs that arise from feature interactions.

Of course, using a *product-based* analysis, we could analyze each product of a product line individually. This way, we do not need to adapt existing analysis tools, but would face severe scalability problems for larger sets of products. However, a developer could analyze only a subset of all products, which is again incomplete.

Each of these three product-line analysis strategies has different strengths and weaknesses, as has been discussed conceptually in the literature (see a recent survey on product-line analysis by Thüm et al. [30]). To gain more empirical evidence, different researchers began to compare product-line analysis strategies quantitatively, for example, in terms of scalability and coverage. For example, Apel et al. compared family-based and product-based strategies in the context of model checking [7], and Liebig at al. compared family-based and product-based strategies in the context of type checking and data-flow analysis [21].

Despite existing empirical work on product-line analysis, there is no any study that compared all three strategies in a controlled setting (i.e., by means of a common set of subject systems and the same analysis tool that implements all three strategies). Our goal is to close this gap. As a concrete analysis technique, we chose type checking, as it has been used before in several studies on product-line analysis [3, 14, 18, 21]. As an implementation technique, we use feature-oriented programming [6], because we already have a proper tool infrastructure at our disposal, and we have access to a repository of subject systems for our experiments.

For our experiments, we implemented the three type-checking strategies—family-based, feature-based, and product-based—as an extension of FUJI, an extensible compiler for feature-oriented programming in JAVA [5]. Using FUJI, we compared the three strategies by applying them to 12 feature-oriented, JAVA-based product lines, from different application domains and of different sizes. Overall, we found that the family-based strategy is superior in that it is complete and takes substantially less time for type checking than the other strategies. The feature-based strategy is also quite fast, compared to the product-based strategy, but incomplete.

GPCE '13, October 27–28, 2013, Indianapolis, IN, USA.

http://dx.doi.org/10.1145/2517208.2517213

Based on the experimental results, we discuss a number of issues regarding the ability to detect and report errors, the role of optimization for family-based strategies, the influence of factors such as the size of a product line, and the trade-off between analysis coverage and time.

To summarize, we make the following contributions:

- We implemented a type checker for feature-oriented, JAVA-based product lines that supports family-based, feature-based, and product-based type checking. This is the first time that all three analysis strategies have been integrated within a single tool.
- We compare the three type-checking strategies regarding different aspects, such as the ability to detect different kinds of type errors and the quality of the provided information about errors.
- We present and discuss the results of a comparative performance evaluation that we conducted on a set of 12 subject product lines. Most notably, we found that the family-based strategy is the most efficient strategy for all of them. It is substantially faster, detects all kinds of errors, and provides the most detailed information about the errors found, but it requires adaption of the standard type-checking tools.

The implementation of the strategies (in the form of a FUJI compiler extension), the subject product lines, and the experimental data are available online: `http://fosd.de/fuji/`.

2. Product-Line Type Checking

By means of a running example and type checking as a concrete analysis technique, we illustrate product-based, feature-based, and family-based strategies for product-line analysis.

2.1 Running Example

The example in Figure 1 is a very simple product line of list data structures. It consists of two features: *Base* and *Batch*. The mandatory feature *Base* provides two basic implementations of lists: SingleList for singly-linked lists, and DoubleList for doubly-linked lists. It also provides a test class TestCase. The optional feature *Batch* provides a special list implementation BatchList for scheduling batch jobs that uses class SingleList of feature *Base*.

In practice, not every combination of features is valid. Feature models are commonly used to describe the conditions of the absence and presence of features, including dependencies among features [17]. In Figure 2, we show the feature model of our example. It states that feature *Base* is mandatory (i.e., it must be present in every product) and feature *Batch* is optional (i.e., it may be present in a product). Consequently, our running example consists of two valid products: {*Base*} and {*Base*, *Batch*}.

If we take a closer look at the code of feature *Base*, we see that it refers to class BatchList in Line 9 of Figure 1. If we attempt to compile a product with *Base* and without *Batch*, we get a type error, because BatchList is declared only in *Batch*. This dangling reference is a simple example of an error that involves multiple features. The cause of this type error is an inconsistency between the feature model and the implementation of the product line: The feature model suggests that feature *Base* is independent from feature *Batch*, but the implementation requires these features to be selected together.

To resolve the inconsistency, we can make *Batch* mandatory or move the test case from *Base* to *Batch*. The key point is that, in large-scale product lines, such inconsistencies may go unnoticed for a long time and show up only late in the development cycle [28].

Another kind of type error is illustrated in Line 17 of Figure 1. There, the undeclared variable resutl is accessed. This type error is caused by a simple typo. It is an example of a *feature-local* error

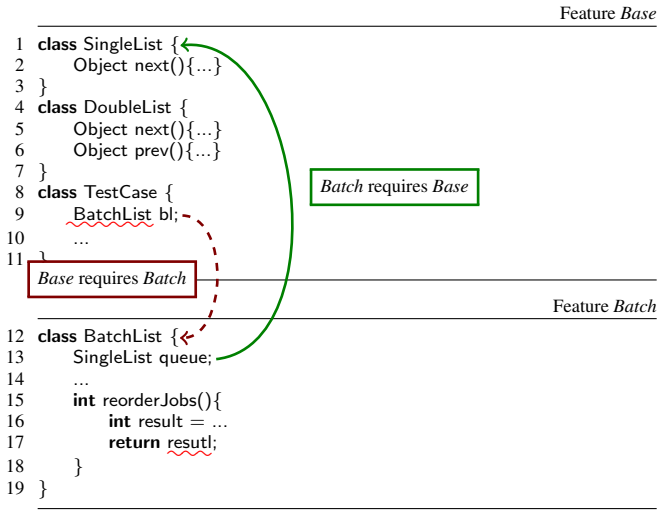

Figure 1. Running example: two basic list implementations (feature *Base*) and a list for batch jobs (feature *Batch*); type errors are underlined; arrows denote references; the dashed arrow denotes a possibly dangling reference.

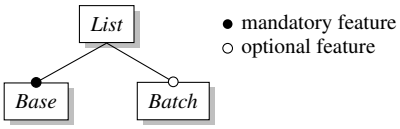

Figure 2. Feature model of the running example.

that can be discovered as soon as the affected feature is used in a product.

Next, we discuss which type errors can be detected by different type-checking strategies. We consider what information can be provided to a developer by a type checker to help to fix these errors. We also take a look at certain properties of the strategies that can influence type-checking performance.

2.2 Type Checking Product Lines

Our procedure for type checking of product lines consists of two steps, as illustrated in Figure 3. In the first step, *setup*, we parse the code of the considered features and compose it accordingly. In the second step, *checking*, we perform the actual type checks on the result of the first step. If a particular type-checking strategy cannot check the whole product line in a single run, the type-checking procedure is repeated. For example, the product-based strategy repeats the procedure for each product, the feature-based strategy repeats it for each feature, and only the family-based strategy checks all products simultaneously in a single run.

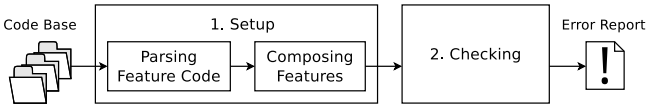

Figure 3. Steps of product-line type checking.

The performance of a type-checking strategy is the total time t required to check the entire product line:

$$t = \sum_{i=1}^{r} t_i^{\text{setup}} + t_i^{\text{checking}}$$

The value r is the number of type-checker runs needed to verify the complete product line; t_i^{setup} is the time used by the setup in run i, and t_i^{checking} is the time used by the checking step in run i. Based on this equation, we can derive the following possibilities for optimizing the type-checking procedure, of which the tree type-checking strategies make use to different extents:

- Minimize the number of type-checker runs r
- Minimize the setup time t_i^{setup}
- Minimize the checking time t_i^{checking}

2.3 Product-based Strategy

To ensure that every product of a product line is well typed, we can apply the product-based strategy. That is, we generate and check each product individually. This way, we find every type error in all products of the product line.

We can even use an off-the-shelf type checker (or compiler) for this task, because the individual products do not contain any compile-time variability. In our setting, the products are normal JAVA programs.

However, by generating and checking individual products, we lose information about the features the products are made of as well as about their dependencies. This makes it difficult to create meaningful error messages. The error messages for our running example will only tell us that one product accesses an unknown type BatchList. A product-based type checker fails to identify the primary reason, namely, the false optionality of feature *Batch*. That is, the type checker cannot blame the feature that is responsible for the error, which is left to the user. This also applies to feature-local errors, such as for the undeclared variable resutl.

A major weakness of the product-based strategy is its poor scalability. With n optional and independent features, we have to repeat the type-checking procedure for each of the 2^n products. Therefore, the *upper bound* for the performance is

$$t = \sum_{i=1}^{2^n} t_i^{\text{setup}} + t_i^{\text{checking}} \qquad \text{(Product-based)}$$

The reason for the poor scalability are redundant analyses made in every step of the type-checking procedure (Figure 3). During the setup, we repeatedly parse and compose the same features again and again. During type checking, we repeat type checks that are similar among different products.

To avoid this redundancy, it is possible to parse the code of feature *Base* only once, because the corresponding parse tree is the same per product. Likewise, it is sufficient to perform type checks that concern code inside *Base*, such as type checking the body of method next, only once.

The remaining two type-checking strategies exploit this optimization potential, which we explain next.

2.4 Feature-based Strategy

Using the feature-based strategy, we check every feature of a product line individually. We assume that all types, declarations, and so on that a feature requires are available in all valid products. For example, if we check feature *Base*, then the feature-based type checker assumes that the required type BatchList, provided by feature *Batch*, is always available. Type BatchList becomes part of the feature's required interface. While the feature-based strategy may seem naive at first glance, it is motivated by open-world systems, in which not all features are known at development time [20, 22].

Technically, we implement the required interface of a feature module using stubs. A *stub* is a bundle of JAVA interfaces and classes, possibly with member prototypes, that represent the types and members a feature requires from other features. Either stubs are provided by the developer to define the required interface, such as in HYPER/J [27], or they are generated using tools, such as AHEAD [29] or FEATURESTUBBER.[1] To type check feature *Base* of our example, a stub containing an empty class named BatchList is needed, possibly with proper member declarations.

While checking a feature, the feature-based strategy does not know anything about other features. Consequently, a feature-based type checker cannot detect type errors that arise between features. In our example, it cannot detect the erroneous access to the missing type BatchList, because the type is provided by the stub (i.e., the type checker simply assumes that it will be provided by another feature). Only errors that are local to a feature can be detected by the feature-based type checker, as the undeclared variable resutl.

To guarantee that all products are well typed, feature-based type checking has to be supplemented with additional type checks during byte-code feature composition (which corresponds to linking in C). The result is a mixed *feature-product-based* strategy [30].

Much like for the product-based strategy, we can use an off-the-shelf type checker for feature-based type checking, because a single feature complemented with stubs does not contain any compile-time variability.

In contrast to a product-based type checker, a feature-based type checker can provide sufficient information about feature-local errors, but it misses errors that arise from combinations of features.

Furthermore, the feature-based strategy requires one type-checker run for each feature. Thus, every feature is parsed and checked only once, and the number of the unnecessarily repeated actions is reduced compared to the product-based strategy. The strategy also completely avoids the feature-composition part of the setup (cf. Figure 3). To summarize, it utilizes the following optimization possibilities:

- The number of type-checker runs r is reduced from 2^n to n, where n is the number of features.
- The setup time t_i^{setup} is reduced by omitting feature composition.

Therefore, the performance of the feature-based strategy is

$$t = \sum_{i=1}^{n} t_i^{\text{setup}} + t_i^{\text{checking}} \qquad \text{(Feature-based)}$$

2.5 Family-based Strategy

The family-based strategy analyses the code base of a product line as a whole. Hence, it can detect all type errors and guarantee that all products of a product line are well typed.

The variability of a product line has to be incorporated into the type-checking procedure, so that the type checker can take it properly into account. The key idea is to compose all features of the product line (even mutually exclusive ones), and to keep variability information in the syntax tree (i.e., which program element belongs to which feature and depends on which other features) [29]. This way, the syntax tree does not represent only a single product or feature, but the whole product line, as illustrated in Figure 4. A family-based type checker works on these enriched syntax trees and must be able to cope with variability. For this reason, we cannot use an off-the-shelf type checker for this task.

[1] `http://fosd.de/featurebite/`

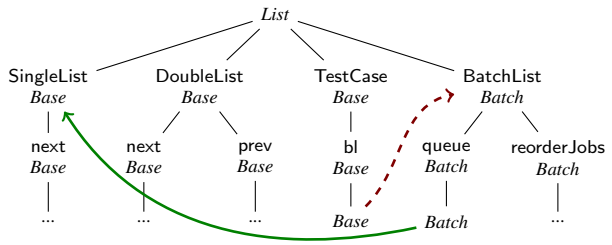

Figure 4. Syntax tree of the running example. Every node knows the feature to which it belongs. Arrows denote references; the dashed arrow denotes a possibly dangling reference (cf. Figure 1).

```
Base/TestCase.java:9:
  Type Error: 1 optional target:
    Feature Base accesses the type
    (default package).BatchList of feature Batch
```

Figure 5. Error message of the family-based type checker.

Figure 5 shows an example of an error message produced by our family-based type checker. The error message identifies the features participating in the type error. It shows in which feature and where exactly in its code the error occurs. Moreover, the error message describes the exact cause of the error, namely that feature *Base* requires feature *Batch* (because *Base* uses type BatchList introduced by *Batch*), but feature *Batch* is not present in all products in which *Base* is present; all these products contain the type error, which is useful information for debugging.

The family-based strategy parses and composes the code of all features in one run. Thus, no repetitive parsing or composition of the same source code is necessary. The resulting syntax tree represents the whole product line. Therefore, only one type-checking run is needed to cover the whole product line. To summarize, the strategy utilizes the following optimization possibilities:

- The number of type-checker runs r is reduced to the minimum of one.
- Furthermore, the time t^{checking} for type checking can be reduced by using caching, as we will explain in Section 3.6.

Therefore, the performance of the family-based strategy is

$$t = t^{\text{setup}} + t^{\text{checking}} \qquad \text{(Family-based)}$$

2.6 Summary

For a better overview, Table 1 summarizes the properties of the three strategies, regarding performance, optimization, the ability to find errors and to blame features, and the possibility to reuse off-the-shelf tools.

3. Empirical Evaluation

The main goal of our evaluation is to compare the three strategies quantitatively in terms of their performance. We believe that the family-based strategy outperforms the other two, because it reduces the number of type-checker runs to one. This way, the strategy avoids unnecessarily repeated analysis operations during the setup and checking steps. Nevertheless, a family-based type checker has to take the whole variability of a product line into account. Therefore, the respective problem is more complex and may require more time for analysis. Moreover, family-based type checkers rely on SAT solvers to determine dependencies between features [3, 18]. The corresponding SAT solver calls are expensive and may reduce

Feature *BatchSingle*

```
20 class BatchList extends SingleList {
21   ...
22 }
```

Feature *BatchDouble*

```
23 class BatchList extends DoubleList {
24   ...
25 }
```

Figure 6. Example of a variable type hierarchy: The choice between *BatchSingle* and *BatchDouble* defines the superclass of class BatchList.

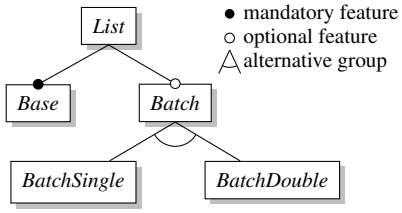

Figure 7. Feature model for the running example, extended with the two new features *BatchDouble* and *BatchSingle*.

the overall performance. Therefore, product-based type checking may be faster than family-based, especially, on product lines with a small number of products.

As for the feature-based strategy, it processes the same amount of code as the family-based strategy and completely avoids the composition part and ignores feature combinatorics. Thus, it is an open question whether it outperforms the family-based strategy when the analyzed product line has a small number of features. Of course, the potential win of the feature-based strategy would be at the expense of the number of type errors found, as it would detect only feature-local errors (see Section 2.4). To be able to detect errors occurring between features, we supplemented the strategy with additional type checks at the byte-code level, as explained in Section 2.4. These type checks run on the per-product basis when composing separately compiled feature modules, which corresponds, in fact, to a mixed feature-product-based strategy. For illustration, we also present the performance measurements for this mixed strategy.

3.1 Hypotheses

Based on our previous considerations, we state the following three hypotheses that we address in the evaluation:

- **H.1:** The family-based strategy is superior to the feature-based and product-based strategies in terms of performance.
- **H.2:** As an exception to H.1, the product-based strategy is superior to the family-based one, if the analyzed product line has a relatively small number of products.
- **H.3:** As an exception to H.1, the feature-based strategy is superior to the family-based one, if the analyzed product line has a relatively small number of features.

3.2 Implementation

We implemented the three type-checking strategies in a single type checker. The type checker is an extension of a feature-oriented JAVA compiler FUJI.[2] It operates on the abstract syntax tree, built

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

Table 1. Conceptual comparison of the three type-checking strategies.

Strategy	Performance	Optimization	Error detection	Feature blaming	Tool reuse
Product-based	$\sum_{i=1}^{\mathbf{2^n}} t_i^{\text{setup}} + t_i^{\text{checking}}$	—	●	○	●
Feature-based	$\sum_{i=1}^{\mathbf{n}} t_i^{\text{setup}} + t_i^{\text{checking}}$	# runs, setup	◐	●	●
Family-based	$t^{\text{setup}} + t^{\text{checking}}$	# runs, checking	●	●	○

● possible
◐ partly possible
○ impossible

by the FUJI's parser, and extends the underlying JAVA type system to implement variability-aware type checks.

To calculate dependencies between features, our type checker uses a corresponding library from the FEATUREIDE project.[3] In FEATUREIDE, the problem of determining a dependency between features is reduced to a SAT problem and solved by querying an off-the-shelf SAT solver (SAT4J). FEATUREIDE implements a caching mechanism to reduce the response time in the case of multiple identical SAT solver queries (see Section 3.6, Caching).

It is important to note that our type checker covers many but not all JAVA type rules. In a nutshell, it checks all accesses to fields, methods, constructors, and types, and verifies that the accessed elements are present in all corresponding products. The possibly variable type hierarchy of the corresponding product line is considered too, because it can influence the presence or absence of program elements, such as fields or methods. For illustration, let us assume that we decided to add two *alternative* features to our running example, as illustrated in the Figures 6 and 7. The new features refine feature *Batch*, and specify a new superclass for class BatchList. Consequently, the two alternative features define which methods are inherited by BatchList. If feature *BatchSingle* is selected, the superclass of BatchList is SingleList, and BatchList inherits method next. If feature *BatchDouble* is selected, the superclass of BatchList is DoubleList, and BatchList inherits the methods next and prev. Now, if prev is called on a BatchList object, the type checker has to determine in which feature combination method prev is inherited by BatchList (in our case, the required combination is {*Base*, *BatchDouble*}).

Further type rules cover explicit and implicit casts, which also take a possibly variable type hierarchy into account. The rules for implicit casts cover assignment expressions involving two variables, assignments of a return value of a method call, parameter passing, and so on.

3.3 Subject Systems

We conducted the evaluation of the type-checking strategies on a set of 12 feature-oriented, JAVA-based product lines. The set has been collected and prepared before for benchmarking purposes and was used in several studies [1, 5]. The subject systems belong to different application domains, and they are of different sizes: in terms of lines of code, number of features, and number of products. In Section 3.6, we use the three size categories to discuss the relation between the size of a product line and the corresponding performance of type checking. Table 2 summarizes relevant information about the systems.

3.4 Measurement Procedure

To compare the performance of the three type-checking strategies, we applied each strategy to each subject system, and we measured the time required by every step of the type-checking procedure (i.e., t^{setup} and t^{checking}). We repeated each measurement 10 times and took the average value to reduce measurement bias. The maximum relative standard error was 3.1 %, which we observed for family-based type checking of ZIPME. For the product-based strategy, we did not include the time needed to generate the configuration of each product, because this time was negligible compared to the time required for type checking. Likewise, for the feature-based strategy, we did not include the time used to generate stubs (Section 2.4), because stubs represent required interfaces and are part of the corresponding feature modules. We measured the performance of the family-based type checker twice, with and without caching, to investigate the influence of the caching (Section 3.2) on the overall performance.

Table 2. Overview of the subject systems (LOC: number of lines of code; # F: number of features; # P: number of products).

System	Domain	LOC	# F	# P
EPL	Expression evaluation	304	12	425
GPL	Graph library	2 855	25	156
GRAPHLIB	Graph library	401	5	16
GUIDSL	Configuration tool	14 318	26	24
NOTEPAD	Text editor	2 193	10	512
PKJAB	Chat client	4 109	8	48
PREVAYLER	Persistence library	6 185	6	32
RAROSCOPE	Compression library	415	4	16
SUDOKU	Game	1 926	6	64
TANKWAR	Game	4 845	38	2 458
VIOLET	Model editor	10 866	88	$\approx 2^{88}$
ZIPME	compression library	5 076	13	24

We instrumented the code of the FUJI compiler with calls to the timer of ThreadMXBean,[4] such that we measure only the CPU time consumed by the type-checker thread. This approach eliminates the influence of other concurrent tasks (e.g., garbage collection) on the measurement results. Furthermore, we did not measure the JVM startup time for each type-checking run, because the overhead can be avoided using special tools.[5]

We conducted all measurements on a workstation equipped with an Intel Xeon CPU (2.9 GHz) and 8 GB RAM, running Ubuntu 12.04 (64-Bit) and OpenJDK 7 (u21).

3.5 Results

In Table 3 (page 10), we present the results of our measurements (in seconds). For each subject system and each strategy from our comparison, we show the setup time (t^{setup}), the checking time (t^{checking}), and the total time (i.e., the performance of the strategy, t). We also provide total times (t) for the feature-product-based strategy and the family-based strategy with caching disabled. For the feature-based strategy, we computed the speedups relative to the product-based strategy. For the family-based strategy, we computed the speedups relative to the product-based strategy and the feature-based strategy. Note that we aborted the product-based measurements for VIOLET after checking 40 random products, because it was impossible to check all of the approximately 2^{88} products in

[3] http://fosd.de/featureide/

[4] java.lang.management.ThreadMXBean is part of the JAVA 7 API.

[5] http://martiansoftware.com/nailgun/

reasonable time. We mark the corresponding values in the table with "X."

For comparison, we visualize the results in Figure 8 by means of bar plots. There is one bar plot per subject system, consisting of five stacked bars, divided into two groups (with different axes to compensate the considerable differences between the measured times). Each bar denotes the amount of time (in seconds) used by the corresponding type-checking strategy. The light gray part of each bar denotes the time required by the setup step; the white part denotes the time required by the check step. The crosses over the bars for VIOLET indicate that we aborted measurements at this point.

Note that, for the family-based strategy, there are two bars: the first, FM, denotes the performance of the strategy with SAT-solver caching *enabled*; the second, FM*, denotes the performance with SAT-solver caching *disabled.* FT* denotes the performance of the feature-product-based strategy; its dark gray part denotes the time required by the byte-code feature composition (Section 2.4).

As we can see, the family-based strategy is the fastest for all subject systems. Compared to the product-based strategy, the minimum *speedup* of this strategy has been measured for GUIDSL, where it is 8.8 times faster. The maximum speedup of 745.3 has been measured for TANKWAR. As we could not check all products of VIOLET in reasonable time, we do not consider the corresponding speedups (they are likely to be much higher). Compared to the feature-based strategy, the speedup of the family-based strategy lies in between 1.7 and 6.5. The feature-based strategy is the second fastest. Its speedup compared to the product-based strategy lies in between 2.2 and 129.7.

Recall that the feature-based strategy finds only feature-local errors (Section 2.4). In our evaluation, it found no errors at all. The reason is that our subject systems have been used in many previous studies. Every single feature of the product lines was type checked as part of a product at least once. Therefore, all feature-local errors have already been detected and fixed. The other two strategies detected the same 556 unique type errors. These errors occurred between features and stayed undetected, because the corresponding feature combinations have been never considered by the developers and users of the systems.

The results support our first hypothesis H.1 (Section 3.1): the family-based strategy is superior to the other two strategies in terms of performance. Although quite apparent from Table 3, we still conducted statistical tests to test all hypotheses. We used the Wilcoxon test, because the data are not normally distributed (according to a Shapiro-Wilk test). Though, we could use the non-parametric ANOVA, we decided to use the conservative double-test variant with Bonferroni correction, because it is more rigorous. For all performance comparisons, the p value is much smaller than 0.01.

We found no supporting evidence for hypothesis H.2 (i.e., product-based is superior on product lines with few products) and H.3 (i.e., feature-based is superior on product lines with few features), because, for none of our subject systems, the product-based or the feature-based strategy is superior to the family-based strategy, not even for very small product lines with few products and features (e.g., GRAPHLIB and RAROSCOPE).

A comparison of the results for the two variants of the family-based type checker shows a substantial influence of SAT-solver caching on performance.

Finally, the feature-product-based strategy is always slower than the family-based strategy (with caching) and the feature-based strategy. More interestingly, it is only in several cases slower than the product-based strategy (e.g., GUIDSL, PKJAB, ZIPME), which indicates the benefits of separate feature compilation and byte-code feature composition.

Table 4. Break-even points of the superiority of the family-based strategy for the subject systems. Total number of products #P and break-even points ↯B (i.e., the number of products whose cumulative analysis time exceeds the time needed by the family-based strategy to check *all* products of the product line).

System	#P	↯B	System	#P	↯B
EPL	425	2	PREVAYLER	32	2
GPL	156	4	RAROSCOPE	16	2
GRAPHLIB	16	2	SUDOKU	64	2
GUIDSL	24	3	TANKWAR	2458	4
NOTEPAD	512	3	VIOLET	$\approx 2^{88}$	8
PKJAB	48	2	ZIPME	24	2

3.6 Discussion

Next, we discuss the results of our measurements based on the size categories of Table 2 as well as regarding the implementation of our type checker. We use the product-based strategy as the base line, and compare it to the other strategies. We subdivided this section in three parts, one part for each strategy.

Product-based strategy. The measurements of the product-based strategy support our expectations about its poor scalability. As discussed in Section 2.3, this strategy induces considerable redundant work in every step of the type-checking procedure, while the number of the unneeded repetitions increases with the number of products. An extreme example is VIOLET, which we could not even check completely, because of the sheer amount of time required to generate and check all of the approximately 2^{88} products. Nevertheless, if developers have to use a standard (non-variability-aware) type checker, product lines with few products and relatively small code bases (e.g., RAROSCOPE) can be checked in reasonable time.

Comparing the results for GUIDSL and ZIPME makes it apparent that it is insufficient to consider only the number of products when estimating the performance of the product-based strategy. Both product lines have the same number of products, but type checking GUIDSL lasts almost twice as long as type checking ZIPME. The cause is the larger code base of GUIDSL, which is almost three times larger than the code base of ZIPME. Systems with a similar number of products and a similar size of the code base (e.g., GRAPHLIB and RAROSCOPE) have similar times.

Although not in the scope of our study, one could use sampling to speed up the product-based strategy [15, 16, 23–25]. This would render the analysis incomplete, but tractable, at least. To give an impression of how the family-based strategy performs in comparison to sampling, we computed the (average) number of products one has to check with a sample-based strategy to exceed the time needed for the family-based strategy. This number marks the break-even point, at which the family-based strategy is superior without question (recall sampling is incomplete). In Table 4, we list the break-even points for the subject systems in terms of this number (and the overall number of products checked by the product-based strategy). The results are clear: Only when checking a very small number of products (less than 5 %, on average), a sample-based strategy is faster. But these small numbers also mean that the coverage will be very low and does not satisfy state-of-the-art coverage criteria (e.g., pair-wise coverage [23]). This observation is in line with previous results [21].

Feature-based strategy. A feature-based type checker parses and checks the code of each feature only once (cf. Section 2.4). The result of this optimization becomes apparent when we compare the performance of the product-based and feature-based strategies in the setup step. With a growing number of products the advantage of the feature-based strategy becomes more evident. Nevertheless,

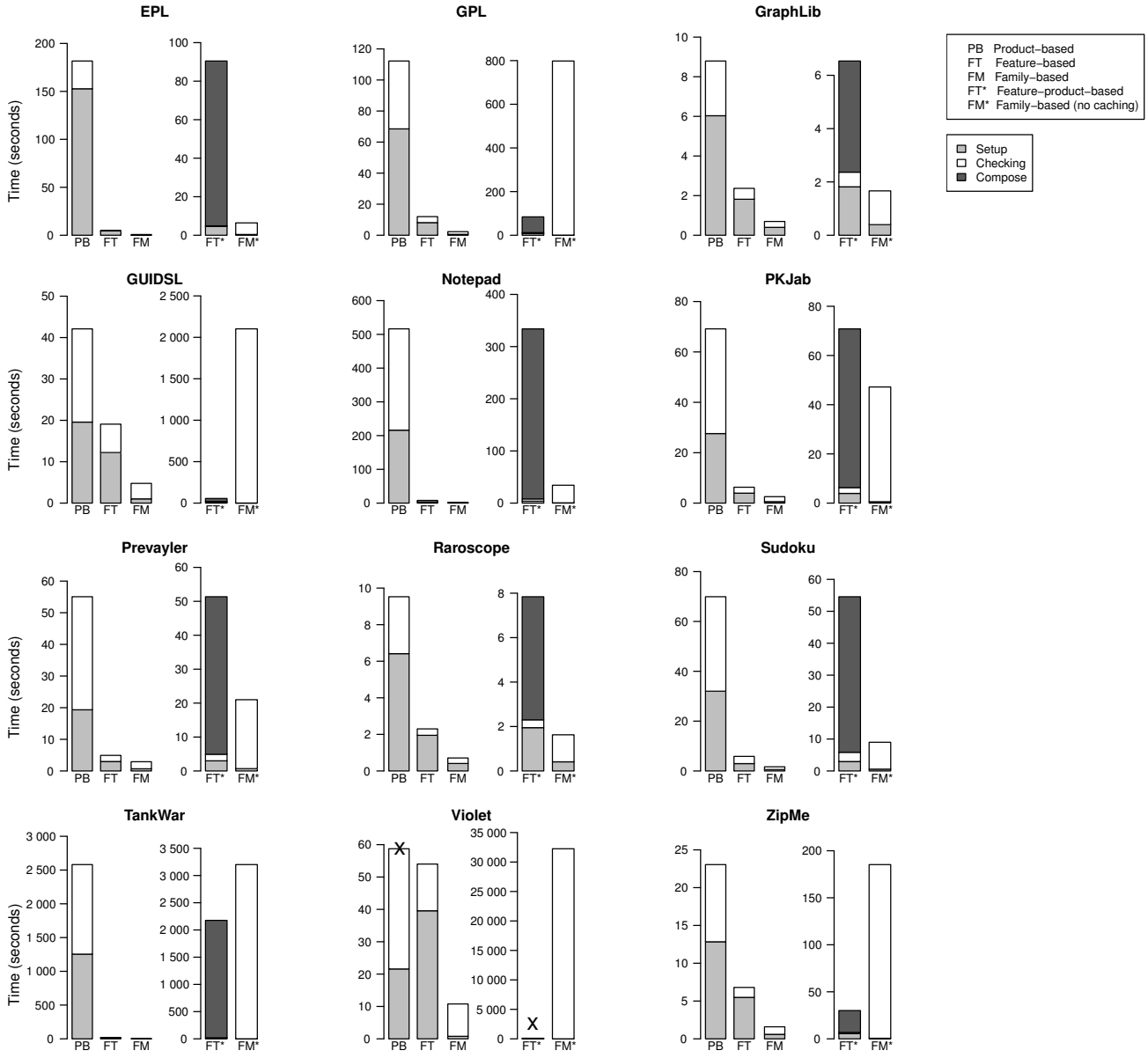

Figure 8. Type-checking times for each subject system—five bars per system. A bar denotes the time used by the corresponding type-checking strategy. Each step of the type-checking procedure (Section 2.2) is denoted by a different color inside a bar. The crosses over the bars for VIOLET indicate that we aborted the product-based measurement after checking 40 products (cf. Section 3.5).

the feature-based strategy induces an overhead for every type-checker run that is caused by instantiating internal data structures and loading classes from the JAVA run-time library. This overhead explains why setting up type checking for GUIDSL is only slightly faster using the feature-based type strategy, then using the product-based strategy. A peculiarity of GUIDSL that is responsible for this effect is that it has a relatively small number of products and more features than products.

The checking step of the feature-based strategy also consumes less time than that of the product-based strategy. Still, we have to keep in mind that the feature-based strategy is able to detect only feature-local errors (cf. Section 2.4). Our subject systems have been used in many previous studies and all eventual feature-local errors have already been fixed. Therefore, the feature-based strategy found no errors. The inability to detect the full range of errors is the main weakness of this strategy.

We used FEATUREBITE[6]—a tool developed by us—to perform supplementary type checks when composing individually compiled feature modules to products. These additional checks at the byte-code level find type errors that arise between features (cf. Section 2.4). This way, we can achieve the same level of type safety as with the other two type-checking strategies (all 556 errors are found). However, our evaluation demonstrates that attaining type safety by combining the feature-based and product-based strategy requires considerably more effort than using the feature-based strategy alone, which was to be expected. The interesting finding is that,

[6] `http://fosd.de/featurebite/`

in all subject product lines except GUIDSL, PKJAB, and ZIPME, the feature-product-based strategy outperforms the product-based strategy.[7] The reason is that the number of products of these three product lines is relatively low compared to the number of their features, which outweighs the benefit of separate feature compilation. This result demonstrates that the intermediate steps of checking and compiling feature modules and composing them at the byte-code level can positively influence analysis performance.

Family-based strategy. The family-based strategy is the clear winner among the three strategies. It requires only one run to check all products of a product line. Consequently, it does not induce the overhead of feature-based type checking (i.e., repeated instantiation of data structures in each run) in the setup step. It also avoids the overhead of the product-based strategy (i.e., repeated type checks) in the checking step (cf. Section 2.5).

Furthermore, our results show that the family-based strategy outperforms also the feature-based strategy in the checking step, even though the feature-based strategy considers only features in isolation. We attribute this phenomenon to the same kind of overhead that the feature-based strategy induces in the setup step (i.e., repeated instantiation of data structures in each run). From Table 3 (page 10), we can see that the advantage of the family-based strategy in the checking step increases with the number of features. For product lines with a small number of features (e.g., GRAPHLIB, PREVAYLER, RAROSCOPE, SUDOKU), the family-based strategy is 1.7 to 3.5 times faster than the feature-based strategy. For product lines with larger numbers of features (e.g., GPL, GUIDSL, TANKWAR), the family-based strategy is 4.1 to 5.8 times faster than the feature-based strategy.

Caching. One property of our family-based type checker poses a principal boundary on its performance. The type checker reduces the problem of determining dependencies between features to a SAT problem (cf. Section 2.5). SAT is NP-complete, which renders family-based type checking NP-complete, as well (w.r.t. the number of features). Luckily, today's SAT solvers mitigate this theoretical boundary for practical problems. Nevertheless, the calls to a SAT solver are still expensive enough, so minimizing the number of such calls is always a good idea.

Our family-based type checker uses a caching mechanism. All queries of the type checker to the SAT solver are cached, and none of the queries is performed twice. As we can see from the measurements (the FM and FM* bars in each plot, Figure 8), the caching mechanism leads to a substantial speedup. This is due to the fact that the family-based type checker makes a considerable number of repeated, identical calls that involve the SAT solver.

A large number of features often means a more complex feature model and, consequently, more expensive SAT solver calls. A small number of products keeps the time needed for the product-based type checking relatively low. ZIPME and GUIDSL are such product-lines, and, as we can see in Figure 8, the family-based type checker without caching is slower than the product-based type checker.

The reasons for the success of caching is that the feature modules in our subject systems are relatively coarse-grained units, and checking them involves checking a large number of identical type, method, and field accesses. This may not be the case if a product line consists of many fine-grained features containing no or few identical accesses (e.g., as may be the case for preprocessor-based variability).

[7] We do not consider VIOLET, as we could not check all its products.

3.7 Threats to Validity

We implemented a substantial subset of type rules in FUJI, but not all type rules specified for the JAVA language. This threatens the internal validity of our study. However, the implemented rules cover a considerable number of language constructs and involve complex analyses of the possibly variable type hierarchy. We can safely assume that adding new type checking rules (e.g., checking access modifiers) will not change the overall picture substantially.

As often the case, the external validity of our study is affected by the choice of the subject product lines. In our evaluation, we used only product lines built with AHEAD/FUJI-style feature modules. The coarse-grained nature of these features is beneficial for the caching mechanism used in the family-based type checker (cf. Section 3.6, Caching). Although, we cannot draw sound conclusions for other kinds of feature implementations (e.g., based on the C preprocessor), previous work shows a similar picture, at least regarding the performance of the family-based strategy compared to the product-based strategy [21].

4. Related Work

Our classification of product-line analysis strategies is based on a recent survey by Thüm et al. [30]. Beside the classification, the authors discuss the conceptual strengths and weaknesses of the individual strategies. Based on this survey, von Rhein et al. propose the Product-Line-Analysis model [32] that describes a whole spectrum of possible combinations of product-line analysis strategies.

The family-based strategy has been applied to several analysis techniques, including type checking [3, 14, 18, 29], static analysis [9, 10, 21], model checking [4, 7, 11, 19], performance measurement [26], and deductive verification [31]. The feature-based strategy has been used before for type checking [2, 8] and verification [20] of product lines. Product-based analyses with sampling have been used in the context of product-line testing [16, 23] and performance prediction [15, 24, 25].

There are only few studies that compare product-line analysis strategies empirically. Two studies evaluated the performance of the family-based and product-based strategy in the context of product-line verification [7, 13]. Brabrand et al. compares the performance of the family-based and product-based strategy for static analysis [10]. For type checking, Liebig at al. evaluated the efficiency of several sample-based strategies, compared to the family-based strategy [21]. While their results are in line with ours, our work is the first that implements all three strategies in one tool and evaluates them using the same subject systems and measurement procedure, so that all comparisons are made in a controlled setting.

5. Conclusion

For the first time, we compared the three product-line analysis strategies—product-based, feature-based, and family-based—in a controlled setting. In our evaluation, we used feature-oriented programming as an implementation technique and type checking as an analysis technique, although the big picture of our results may be transferable to other techniques. In particular, we compared the analysis performance, but we also addressed the ability to detect different kinds of errors, and the quality of the provided information about errors. Our evaluation is based on a feature-oriented compiler that we extended with the three type-checking strategies for this purpose, and a subject set of 12 feature-oriented, JAVA-based product lines.

A main result of our study is that the family-based strategy outperforms the other strategies for all subject systems in terms of analysis time. We identified its caching mechanism as the key factor for the success, as it substantially reduces the number of

SAT-solver queries. At the same time, the family-based strategy is complete: it finds errors that are feature-local and that occur between features (556 in total), which is not the case for the feature-based strategy. Furthermore, the family-based strategy provides the most comprehensive error messages, as it has all information on features and variability at its disposal, which is not the case for the other two strategies.

Although not being in the focus of our study, we found that pursuing a sampling-based strategy (checking only a tractable subset of products) would not change the big picture. For our subject systems, the break-even point, at which the family-based strategy becomes faster, is at very low numbers of products, which means that the corresponding analysis coverage of sampling is likely to be very small, compared to the family-based strategy, which achieves full coverage.

Surprisingly, the feature-based strategy is often slower than the family-based strategy, although it ignores feature interactions and is, consequently, incomplete. Combining the feature-based with the product-based strategy makes it complete, but is substantially slower. Interestingly, such a combined strategy outperforms the plain product-based strategy in most cases in our experiments, which indicates that separate feature compilation and byte-code composition can have a positive effect on analysis performance.

An interesting avenue of further work is to combine the individual strategies and to explore trade-offs in the search for an optimal strategy [32].

Acknowledgments

We thank Peter Lutz for the implementation of the family-based strategy in FUJI. This work was supported by the DFG grants AP 206/2, AP 206/4, and AP 206/5.

References

[1] S. Apel and D. Beyer. Feature Cohesion in Software Product Lines: An Exploratory Study. In *Proc. ICSE*, pages 421–430. ACM, 2011.

[2] S. Apel and D. Hutchins. A Calculus for Uniform Feature Composition. *ACM TOPLAS*, 32(5):19:1–19:33, 2010.

[3] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17 (3):251–300, 2010.

[4] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of Feature Interactions using Feature-Aware Verification. In *Proc. ASE*, pages 372–375. IEEE, 2011.

[5] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich. Access Control in Feature-Oriented Programming. *Science of Computer Programming*, 77(3):174–187, 2012.

[6] S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FEATUREHOUSE Experience. *IEEE Trans. Software Engineering*, 39(1):63–79, 2013.

[7] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proc. ICSE*, pages 482–491. IEEE, 2013.

[8] L. Bettini, F. Damiani, and I. Schaefer. Compositional Type Checking of Delta-oriented Software Product Lines. *Acta Informatica*, 50(2): 77–122, 2013.

[9] E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, and P. Borba. SPL$^{\text{LIFT}}$: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proc. PLDI*, pages 355–364. ACM, 2013.

[10] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. *Trans. on Aspect-Oriented Software Development*, 10:73–108, 2013.

[11] 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 *Proc. ICSE*, pages 335–344. ACM, 2010.

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[13] M. Cordy, A. Classen, G. Perrouin, P.-Y. Schobbens, P. Heymans, and A. Legay. Simulation-Based Abstractions for Software Product-Line Model Checking. In *Proc. ICSE*, pages 672–682. ACM, 2012.

[14] B. Delaware, W. Cook, and D. Batory. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *Proc. FSE*, pages 243–252. ACM, 2009.

[15] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *Proc. ASE*. IEEE, 2013.

[16] M. Johansen, Ø. Haugen, and F. Fleurey. An Algorithm for Generating t-wise Covering Arrays from Large Feature Models. In *Proc. SPLC*, pages 46–55. ACM, 2012.

[17] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, 1990.

[18] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM Trans. on Software Engineering and Methodology*, 21(3):14:1–14:29, 2012.

[19] K. Lauenroth, S. Toehning, and K. Pohl. Model Checking of Domain Artifacts in Product Line Engineering. In *Proc. ASE*, pages 269–280. IEEE, 2009.

[20] H. Li, S. Krishnamurthi, and K. Fisler. Verifying Cross-cutting Features as Open Systems. In *Proc. FSE*, pages 89–98. ACM, 2002.

[21] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *Proc. ESEC/FSE*, pages 81–91. ACM, 2013.

[22] J. Liu, S. Basu, and R. Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering*, 18(1):39–76, Mar. 2011.

[23] S. Oster, F. Markert, and P. Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. SPLC*, LNCS 6287, pages 196–210. Springer, 2010.

[24] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting Performance via Automated Feature-Interaction Detection. In *Proc. ICSE*, pages 167–177. IEEE, 2012.

[25] 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.

[26] N. Siegmund, A. von Rhein, and S. Apel. Family-Based Performance Measurement. In *Proc. GPCE*. ACM, 2013.

[27] P. Tarr, H. Ossher, and S. Sutton Jr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proc. ICSE*, pages 689–690. ACM, 2002.

[28] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature Consistency in Compile-time-configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. EuroSys*, pages 47–60. ACM, 2011.

[29] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. GPCE*, pages 95–104. ACM, 2007.

[30] 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.

[31] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based Deductive Verification of Software Product Lines. In *Proc. GPCE*, pages 11–20. ACM, 2012.

[32] A. von Rhein, S. Apel, C. Kästner, T. Thüm, and I. Schaefer. The PLA Model: On the Combination of Product-Line Analyses. In *Proc. VaMoS*, pages 73–80. ACM, 2013.

Table 3. Measurement results for each subject system and type-checking strategy (in seconds). For each system and each strategy from our comparison, the setup time (t^{setup}), checking time (t^{checking}), and total time (t) are provided. We also provide total times (t) for the feature-product-based strategy and the family-based strategy with caching disabled. For the feature-based strategy, the speedups relative to the product-based strategy are provided. For the family-based strategy, the speedups relative to the product-based strategy and the feature-based strategy are provided. We rounded all values to one decimal place. $^{\text{X}}$ We aborted the product-based measurements for VIOLET after checking 40 random products (cf. Section 3.5).

	Product			Feature				Family					Feature-product	Family (no caching)
	Time (seconds)			Time (seconds)			Speedup w.r.t.	Time (seconds)			Speedup w.r.t.		Time (seconds)	Time (seconds)
System	t^{setup}	t^{checking}	t	t^{setup}	t^{checking}	t	Product	t^{setup}	t^{checking}	t	Product	Feature	t	t
EPL	152.7	28.9	181.7	4.6	0.3	4.9	37.2	0.4	0.4	0.8	240.3	6.5	90.5	6.4
GPL	68.5	43.7	112.2	8.1	3.8	12	9.4	0.6	1.7	2.3	48.9	5.2	84.2	798.2
GRAPHLIB	6	2.8	8.8	1.8	0.5	2.4	3.7	0.4	0.3	0.7	12.7	3.4	6.5	1.7
GUIDSL	19.6	22.5	42.1	12.2	6.9	19.1	2.2	1	3.7	4.8	8.8	4	55.9	2102.2
NOTEPAD	216.1	300.1	516.3	3.7	4.3	8	64.5	0.5	1.6	2.1	242.2	3.8	334	34.2
PKJAB	27.6	41.6	69.2	3.9	2.4	6.3	11	0.6	2	2.6	27.1	2.5	70.8	47.2
PREVAYLER	19.3	35.7	55.1	3	1.9	4.9	11.2	0.7	2.2	2.9	18.9	1.7	51.4	21
RAROSCOPE	6.4	3.1	9.5	1.9	0.4	2.3	4.1	0.4	0.3	0.7	13.4	3.2	7.8	1.6
SUDOKU	32	37.9	69.9	3	2.9	5.8	12	0.6	1.1	1.7	41.7	3.5	54.6	9
TANKWAR	1254.4	1326.5	2580.8	12.7	7.2	19.9	129.7	0.6	2.9	3.5	745.3	5.7	2176.5	3200.9
VIOLET	21.6^{X}	37.1^{X}	58.7^{X}	39.6	14.4	54	1.1^{X}	0.8	10	10.8	5.4^{X}	5	100.2^{X}	32251.1
ZIPME	12.8	10.2	23	5.5	1.3	6.8	3.4	0.6	1	1.6	14.5	4.3	30.1	185.3