Strategies for Product-Line Verification: Case Studies and Experiments

Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer
University of Passau, Germany

***Abstract*—Product-line technology is increasingly used in mission-critical and safety-critical applications. Hence, researchers are developing verification approaches that follow different strategies to cope with the specific properties of product lines. While the research community is discussing the mutual strengths and weaknesses of the different strategies—mostly at a conceptual level—there is a lack of evidence in terms of case studies, tool implementations, and experiments. We have collected and prepared six product lines as subject systems for experimentation. Furthermore, we have developed a model-checking tool chain for C-based and Java-based product lines, called SPLVERIFIER, which we use to compare sample-based and family-based strategies with regard to verification performance and the ability to find defects. Based on the experimental results and an analytical model, we revisit the discussion of the strengths and weaknesses of product-line–verification strategies.**

I. INTRODUCTION

Software product lines (a.k.a. product families) are gaining momentum in academia and industry. A product line is a family of software systems that are distinguished in terms of features (i.e., end-user–visible units of behavior). The goal of systematic product-line development is to facilitate reuse, manage variability, and support automated product generation [12]. Companies and institutions such as General Motors, NASA, HP, Boeing, and Nokia are applying product-line technology to decrease time to market, improve software quality, and diversify their product portfolio (cf. Product-Line Hall of Fame: http://splc.net/fame.html). As product-line technology is increasingly applied to safety-critical and mission-critical software projects (e.g., in the domain of automotive and health-care systems), analysis and verification techniques become important means to ensure correctness, reliability, and security.

Recently, researchers began to apply model-checking technology to analyze and verify software product lines [2], [3], [11], [18]. As a product line may comprise a multitude of products—a number of products that is, in the worst case, exponential in the number of features—it is imperative to tailor existing model-checking technology to the specifics of product lines. The brute-force approach to verify an entire product line, called *product-based strategy*, is to create and analyze every possible product individually. An alternative approach, called *sample-based strategy*, is to concentrate on a subset of possible products (i.e., for some selected feature combinations) to reduce the verification problem and to identify defects faster. A third approach, called *family-based strategy*, is to analyze the design and implementation artifacts of a whole product line (i.e., product family) in one single pass [2], [11], [18], for example, by creating a simulator that simulates the behavior of all individual products [2], [25].

Conceptually, the three strategies of product-line verification have different strengths and weaknesses. The product-based strategy is a brute-force approach: as the number of products increases, this strategy quickly becomes infeasible; it is the base line for our investigation. The sample-based strategy is likely to find defects quickly, depending on how large the sample is, but it may miss defects due to the incompleteness of the approach. The family-based strategy is beneficial for product lines that comprise many products with substantial similarities [2], [11], [18], because similar product behaviors are not re-checked for every product. However, verification tasks may become more complex than for individual products, which may exceed resource limits.

The research community began to discuss and weigh the strengths and weaknesses of sample-based and family-based strategies, each in comparison to the product-based strategy [2], [10], [11], [18], [21], [28]. However, there is no work that compares the sample-based strategy with the family-based strategy systematically in a controlled setting. In general, there is a lack of case studies and experiments in this field, which was the motivation for us to collect and prepare existing and implement further case-study product lines, as well as to develop a model-checking tool chain, called SPLVERIFIER, for the verification of product lines written in C and Java.

Specifically, we conducted a series of experiments based on six case studies to compare the sample-based strategy and the family-based strategy (with the product-based strategy as a base line) with regard to their verification performance and their ability to identify defects. We have concentrated on undesired *feature interactions*, a specific class of defects that is especially challenging for verification, because they emerge between several features and not within individual features [8].

In summary, we make the following contributions:

- We provide the tool chain SPLVERIFIER for conducting experiments with product-based, sample-based, and family-based model checking of product lines written in C and Java.
- We collected and prepared six case studies, written in the general-purpose languages C and Java, to be used as benchmarks for product-line verification.
- Based on the case studies, we conducted experiments comparing the three verification strategies (including three different sampling heuristics for feature-interaction

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

detection) in terms of the verification performance and the ability to identify defects.
- Based on the experiments, we revisit the discussion of the strengths and weaknesses of sample-based and family-based strategies and put them into perspective, and we provide an analytic model that describes the trade-offs of the individual verification strategies. A key result is that, in our experiments, the family-based strategy outperforms the sample-based strategy in terms of defect-detection efficiency.

SPLVERIFIER, all case studies, and the experimental results are available on the project's web site: http://fosd.net/FAV.

II. Product-Line Verification

We introduce product-based, sample-based, and family-based verification strategies by means of an example.

A. Running Example and Setting

For illustration, we use a simple e-mail system as running example. We used the example before in our work on feature-interaction detection [2], but its roots lie in Hall's work on modularity of e-mail systems [14].

We implemented the features of the e-mail system by feature modules in C and Java, and we compose them by superimposition [1]. Basically, superimposition merges the code of all feature modules recursively based on nominal and structural similarity. Feature composition is generally not commutative [1]. The composition tool expects a certain order over the features of a product line that determines the composition order (for a particular feature selection, the composition tool constructs exactly one corresponding product, because the tool is bound to the specific order).

In Fig. 1, we depict excerpts of four feature modules of the e-mail system. Feature *EMailClient* implements a basic e-mail client, feature *Encrypt* encrypts outgoing e-mails, feature *Decrypt* decrypts incoming e-mails, and feature *Forward* forwards incoming e-mails to another host. Note that encryption and decryption rely on the availability of proper keys—a circumstance that gives rise to a feature interaction, as we will explain shortly.

EMailClient is the base feature in our example. It introduces a structure email for representing e-mails and the two functions outgoing and incoming for handling outgoing and incoming e-mails. Composing it with feature *Encrypt*, the existing structure email is extended by the two new fields isEncrypted and encryptionKey, function encrypt is added, and the existing function outgoing is overridden to intercept outgoing e-mails and to encrypt them using function encrypt; keyword original invokes the overridden function. Feature *Decrypt* introduces a function decrypt and overrides the existing function incoming to intercept and decrypt incoming e-mails. Feature *Forward* introduces a function forward and overrides the existing function incoming to forward incoming e-mails to another host.

For the purpose of the example, let us assume that all features are optional except *EMailClient*, which is present in all products. This flexibility gives rise to feature interactions, a class of defects that are difficult to detect. A feature interaction is a situation in which new behavior emerges from the composition of two or more features that cannot easily be deduced from the behavior of the individual involved features. The emergent behavior can be undesired and associated with unexpected program states [8].

While the features *Encrypt* and *Decrypt* are designed to cooperate, feature *Forward* has been developed independently of the two, only based on feature *EMailClient*. The composition of all four features leads to an undesired feature interaction. The interaction occurs if one host sends an encrypted e-mail to a second host that forwards the e-mail automatically to a third host. If the second host does not have the public key of the third host, it forwards the e-mail in plain text (*Forward* has been developed independently and thus does not take encryption into account). This situation contradicts the requirement that encrypted e-mails must never be sent in plain text over the network [14].

Feature *EMailClient*
```
1  // representation of e-mail
2  struct email {
3    int id; char *from; char *to; char *subject; char *body;
4  };
5
6  // outgoing e-mails are processed before they leave the system
7  void outgoing (struct client *client, struct email *msg) { ... }
8
9  // incoming e-mails enter here and are stored in a mailbox
10 void incoming (struct client *client, struct email *msg) { ... }
```

Feature *Encrypt*
```
11 // extending the e-mail structure by information on encryption
12 struct email {
13   int isEncrypted;
14   char *encryptionKey;
15 };
16
17 // encrypt an e-mail, if the public key of the receiver is known
18 void encrypt (struct client *client, struct email *msg) { ... }
19
20 // override 'outgoing' to encrypt e-mails before they are sent
21 void outgoing (struct client *client, struct email *msg) {
22   encrypt (client, msg);
23   original (client, msg); // invoke the overridden function
24 }
```

Feature *Decrypt*
```
25 // decrypt a given e-mail
26 void decrypt (struct client *client, struct email *msg) { ... }
27
28 // override 'incoming' to decrypt encrypted incoming e-mails
29 void incoming (struct client *client, struct email *msg) {
30   decrypt (client, msg);
31   original (client, msg); // invoke the overridden function
32 }
```

Feature *Forward*
```
33 // forward an e-mail to another host
34 void forward (struct client *client, struct email *msg) { ... }
35
36 // override 'incoming' to forward e-mails automatically
37 void incoming (struct client *client, struct email *msg) {
38   forward (client, msg);
39   original (client, msg); // invoke the overridden function
40 }
```

Fig. 1. Implementation of four features of our e-mail client in C [2]

Note that, even if there is a feature model that describes the domain dependencies between features [12], it typically does not cover (hidden) implementation-level dependencies that may lead to inadvertent feature interactions at run time [2]. Hence, we need analysis and verification techniques that check whether a feature composition satisfies the specifications of the involved features.

In our case studies, individual features come with their own specification(s), expressed in the form of assertions that indicate erroneous executions, or by automata that are woven into the code in the form of assertions [2]. In Fig. 2, we show the specification of feature *Encrypt* expressed as an automaton: When the client receives an encrypted e-mail (Lines 7–9), the status (encrypted or not) of the message is stored into a field (Line 8) that has been attached as a shadow to the email structure (Line 4). When an e-mail that was encrypted leaves the system (Lines 11–13), it must still be encrypted; if not, the e-mail client reaches an error state flagged by fail (Line 12).

```
1  automaton EncryptSpec {
2    // introduce an auxiliary field to store the state of an e-mail
3    introduction {
4      shadow struct email { int in_encrypted; };
5    }
6    // if an e-mail is encrypted when entering the system...
7    before void incoming(_: struct client *, msg: struct email *) {
8      msg−>in_encrypted = isEncrypted(msg);
9    }
10   // ...it must be encrypted as well when leaving the system
11   after void outgoing(_: struct client *, msg: struct email *) {
12     if (msg−>in_encrypted && ! isEncrypted(msg)) { fail; }
13   }
14 }
```

Fig. 2. Automaton-based specification of feature *Encrypt* [2]

B. Verification Strategies

Product-Based Strategy: Pursuing a product-based strategy, all products of a product line are generated and analyzed, each using a standard model checker. In our example, we compose the set $\mathcal{P}$ of all eight valid products: *EMailClient* is mandatory, so it is present in all products; of the remaining three features, we form all possible combinations respecting the predefined composition order. Then, we verify the implementation of each product, based on the specifications of the involved features:

$$\forall p \in \mathcal{P} : \forall f \in p : impl(p) \models spec(f)$$

where $impl(p)$ is the implementation of product p, and $spec(f)$ the specification of feature f.

In our setting, the implementation is made up of real C or Java code; specifications are mostly local to individual features and comprise safety properties that must hold if the corresponding features are selected. If there is a global specification, all products are checked against it.

Sample-Based Strategy: Generating and analyzing all products in a brute-force fashion is feasible only for product lines with a small number of products. Hence, the sample-based strategy selects a restricted number of products, usually based on some coverage criterion:

$$\forall p \in sample(\mathcal{P}) : \forall f \in p : impl(p) \models spec(f)$$

where $sample(\mathcal{P})$ selects a subset of all valid products according to a sampling heuristic.

In our experiments, we use the following heuristic for n-wise sampling of feature combinations, where n is the (minimal) number of features in the sample:

$$\text{for each subset } \{f_1, \dots, f_n\} \subseteq F \text{ of } n \text{ features,}\\ \text{select a small product } p \in \mathcal{P} \text{ with } f_1 \in p \wedge \dots \wedge f_n \in p$$

where F is the set of features of the product line and 'small product' refers to a valid product with a small number of features ($\geq$ n).

For $n = 2$ (pair-wise), all binary feature interactions can be detected. While this heuristic reduces the number of products to be generated and analyzed significantly (from an exponential number, in the worst case, to a polynomial number), interactions between more than two features cannot be detected—so the analysis is incomplete. Using this heuristic, we can reduce the number of products in our example from eight to three: as *EMailClient* is mandatory, we form three pairs of the remaining three optional features. However, this way, we do not detect interactions that occur only between all of the three optional features. In our experiments, we also use two other sampling heuristics: single-wise ($n = 1$) and triple-wise ($n = 3$).

Our sampling heuristics are inspired by previous work on feature-interaction detection [9], [23], prediction of non-functional properties [26], and analysis of model-weaving interference [15]. As they aim at small products that cover certain feature combinations, they facilitate the process of detecting, isolating, and understanding individual feature interactions. An alternative would be to aim at large products, which decreases the sample size, but makes identification of feature interactions and the set of all defective features difficult (cf. Sect. IV).

Family-Based Strategy: Typically, there are many similarities between the products of a product line [12]. Thus, checking products individually leads to redundant analyses of execution paths that are similar among products. To minimize effort, the family-based strategy analyzes the entire code base of a product line in one single pass. This can be achieved by combining all code of all features in a single *product simulator*. In a nutshell, a product simulator simulates the behavior of all products of the corresponding product line depending on the values of *feature variables* that represent the presence or absence of individual features [2], [25]. All necessary information on valid feature combinations and feature-dependent execution paths is encoded in the code of the product simulator. The model checker initializes the boolean feature variables within the product simulator using a non-deterministic choice, such that it must assume that all feature combinations that are allowed by the feature model may occur. This way, it checks *all* valid execution paths of *all* products without the need of generating and checking any individual product.

Technically, the approach relies on the concept of *variability encoding* [2], [25], which is a modification of the regular

feature-composition process. Basically, the variability induced by different combinations of features is encoded in the form of conditional program executions using `if` statements:

1) *All* features are composed according to the total composition order of the product line.
2) For each feature, there is a global boolean feature variable defined (initialized at program load time) that models the presence or absence of the feature.
3) For each function refinement, a dispatcher function is introduced that dispatches between the refined and the refining function, depending on whether the feature that contains the refinement is selected.
4) Dependencies between features (i.e., the feature model) are encoded by means of a boolean formula over the feature variables that models the corresponding constraints.
5) The entire program execution is enclosed in a conditional block that is executed only if the constraints imposed by the feature model are satisfied; this way, execution paths that are associated with invalid feature combinations are not considered by the model checker.

The resulting product simulator can simulate the behavior of any product of the product line, depending on the values of the feature variables. More details about variability encoding (including a formal model and a discussion of correctness) are provided elsewhere [2].

Using a family-based strategy, a product line can be checked in a single pass:

$$\forall f \in F : \mathbb{P}(F) \models spec(f)$$

where F is the set of all features of the product line and $\mathbb{P}$ is the product simulator that results from variability encoding, incorporating all valid combinations of features in F.

Figure 3 shows the product simulator for the composition of $EMailClient$ and $Forward$, as produced by the variability encoding of our tool chain (cf. Sect. III). Function `incoming` (Lines 10–13) dispatches between its variants with and without feature *Forward*. The feature model is encoded (Lines 2–7) and the execution is guarded (Line 28). In Fig. 4, we show the effect of variability encoding on the state graph. States that are associated with invalid feature combinations are not considered by the analysis (left sub-tree). All other states are checked (right sub-tree). Hence, it can be verified if none of the valid feature combinations exhibits an unsafe feature interaction (or other defects). Also, one can see how both alternative execution paths—for products with and without feature *Forward*—are encoded in the state graph.

Late Splitting and Early Joining: Largely implicit in previous work, there are two principles that allow the model checker to exploit similarities between products using a family-based strategy: late splitting and early joining.

Late splitting means that, as long as the execution paths of different products are equal (starting from the common program entry point), they are explored by the model checker only once. Only if execution paths diverge (i.e., the values of feature variables differ in the subsequent program state), the state-space exploration is split.

```
1  // one boolean variable per feature
2  int EMailClient, Forward;
3
4  // encoding the feature model
5  int feature_model() {
6    return EMailClient; // EMailClient && (Forward || !Forward);
7  }
8
9  // dispatch between 'Forward' and '!Forward'
10 void incoming (struct client *client, struct email *msg) {
11   if (Forward) { incoming_Forward (client, msg); }
12   else { incoming_EMailClient (client, msg); }
13 }
14
15 // refinement of method 'incoming' by 'Forward'
16 void incoming_Forward (struct client *client, struct email *msg) {
17   forward(client, msg);
18   incoming_EMailClient(client, msg);
19 }
20
21 // base implementation of method 'incoming' by 'EMailClient'
22 void incoming_EMailClient(struct client *client, struct email *msg) { ... }
23
24 // base implementation of method 'forward' by 'Forward'
25 void forward (struct client *client, struct email *msg) { ... }
26
27 int main(int argc, char **argv) {
28   if (feature_model()) { /* start the e-mail client */ }
29   return 0;
30 }
```

Fig. 3. Variability encoding of the features *EMailClient* and *Forward* [2]

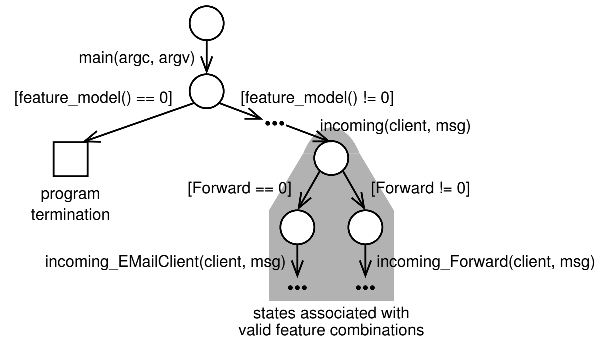

Fig. 4. State graph of the product simulator of Fig. 3 [2]

Early joining means that, if two program states differ *only* in the values of their feature variables, the execution paths are joined and explored from there only once (until a further split). Because feature variables are boolean, they can be efficiently checked for equivalence, and joined using disjunction, by binary decision diagrams (BDDs). We refer to previous work for details on encoding feature-variables in BDDs [10], [30] and on BDD-based software model checking [6].

III. Case Studies and Experiments

Both sample-based and family-based strategies promise to significantly decrease verification time, either by analyzing only a subset of products, or by sharing analysis results among products. To learn about the trade-offs of the two strategies, we compare them (as defined in Sect. II-B) in terms of their verification performance and their ability to identify defects; we use the product-based strategy as a base line.

A key hypothesis is that sample-based strategies are faster than the family-based strategy (the fewer products are checked, the less time is needed for verification), but may miss defective products. Conversely, the family-based strategy consumes more verification time than sampling, but is complete.

With regard to the sample-based strategy, we are interested in the tension between sample, defect, and detection rates and their influence on verification time. For the family-based strategy, we are interested in the factors that influence verification performance. Especially, we want to explore whether late splitting and early joining are the driving factors for the speedups observed by us and others using family-based strategies. We quantify the number of verification steps that are saved due to sharing analysis results among products.

For the experiments, we collected and prepared six case studies, which exceed the case studies used in previous work on product-line verification substantially, in terms of volume and complexity (Sect. IV). This corpus of case studies is also meant to serve as a benchmark suite in further work, which is in itself a valuable contribution to the community.

A. Tool Chain

We developed a tool chain for product-line verification, called SPLVERIFIER, which consists of a number of tools: For feature composition, we use FEATUREHOUSE [1]. For model checking, we use the tools CPACHECKER [4] (revision 5540; branch "explicit") for C, and JAVA PATHFINDER [29] (revision 635) for Java. Both support the verification of safety properties by means of explicit-state and symbolic model checking. Specifically, we use the explicit analyses of CPACHECKER (-explicitAnalysis [5]) and JAVA PATHFINDER (jpf-core); for product simulators, we use additionally a BDD-based treatment (cf. Sect. II-B) of feature variables (-explicitAnalysis-featureVars for CPACHECKER and the jpf-bdd extension [30] for JAVA PATHFINDER).

All specifications are woven into the target code in the form of assertions, by means of aspect weaving (cf. Sect. II-A); we use ACC [1] for C code, and ASPECTJ [2] for Java code. Variability encoding is implemented using FEATUREHOUSE's composition facilities. More details on specification weaving and variability encoding in SPLVERIFIER are available on the project's web site (http://fosd.net/FAV).

B. Subject Systems

As subject systems, we selected three product lines that have been used before to assess product-line verification, and developed implementations in C and Java:

- The **e-mail** system of Hall [14] models an e-mail communication suite. It provides several features, such as encryption, automatic forwarding, and e-mail signatures, which can be activated or deactivated.
- The **elevator** system has been designed by Plath and Ryan [24]. It is an elevator model that is extensible by various features such as stopping if the elevator is empty or priority service for a special floor.
- The **mine-pump** system is based on work in the CONIC project [17]. It simulates a water pump in a mining operation, including several features that vary the pump's behavior. The pump keeps the bottom of the mine shaft dry, but must be deactivated if the mine contains combustible methane gas.

Based on the respective original systems, we created for each system a C and a Java implementation, obtaining six implementations in sum (we used the C implementation of the e-mail system in previous work [2]). Note that the respective Java and C implementations may differ in details. Although we aimed at comparability, the differences of the languages as well as the corresponding support of the model-checking tools forced us to diverge from a common implementation schema (e.g., the explicit-value analysis of CPACHECKER did not yet support arrays and structures).

Additionally, we selected three existing Java product lines from the FEATUREHOUSE repository. [3] All of them have been developed for other purposes. The primary selection criterion was that the Java code could be processed properly by JAVA PATHFINDER.

- **AJStats** is a product line of source-code–analysis tools. It has been developed by the first author to explore the use of AspectJ. It provides several features to tailor the analysis process, for example, recognizing various syntactic program structures.
- **GPL** is a product line of graph libraries developed by Lopez-Herrejon and Batory, as a standard problem for the evaluation of product-line techniques. It allows a programmer to tailor graph data structures, including optional support for weighted and directed edges as well as different traversal strategies and algorithms.
- **ZipMe** is an open-source zip compression library for Java ME. It has been refactored into a product line by Kuhlemann. It includes features for computing check-sums and different compression techniques.

For the product lines e-mail, elevator, and mine pump, we adapted the original specifications, which have been distributed with their models. Mostly, the specifications concern domain-specific safety properties such as that encrypted e-mails are never transferred in plain text, the elevator must refuse to operate if the maximum weight is exceeded, or the mine pump must be deactivated when methane gas is detected. Furthermore, all three case studies contain defects (documented by the original authors) violating at least one specification.

For the product lines AJStats, GPL, and ZipMe, we included proper specifications based on domain knowledge (two for AJStats, two for GPL, and one for ZipMe). The systems AJStats and ZipMe do not contain defects; for GPL, we used defects introduced by others [7]. In Table I, we summarize relevant information on all case studies.

Note that, although the subject systems are implemented in C and Java, the implementations comprise only the key functionalities of the product lines. In this sense, the implemen-

[1] http://research.msrg.utoronto.ca/ACC

[2] http://eclipse.org/aspectj/

[3] http://fosd.net/fh/

TABLE I
OVERVIEW OF SUBJECT SYSTEMS

System	Lang.	LOC	Features	Specs	Products
E-Mail	Java	1233	9	9	40
	C	258	9	9	40
Elevator	Java	1046	6	9	20
	C	877	6	6	20
Mine pump	Java	580	7	5	64
	C	279	7	5	64
AJStats	Java	13393	20	2	200
GPL	Java	1405	18	2	42
ZipMe	Java	3636	8	1	10

tations are models of the respective product lines. Nevertheless, we use software model-checking technology to verify the code, *without* the need of extracting intermediate models manually—the software model checkers extract the models from the code automatically (cf. Sect. III-A).

C. Experiments

We performed all experiments on a Ubuntu 11.10 system that has an Intel i7-2600 CPU with 3.4 GHz, 8 cores, and 16 GB RAM. For each subject system, we created:

1) all products (product-based strategy),
2) product samples that cover all (a) single-wise, (b) pair-wise, and (c) triple-wise combinations, and
3) a corresponding product simulator using variability encoding (family-based strategy).

The overall goal of the verification tasks is to identify all defective products (which violate the specification of one feature, at least) of the given product line. That is, for each specification, we have to run a sequence of verification tasks: for the product-based and sample-based strategies, one verification task per (selected) product; for the family-based strategy, one verification task for the product simulator.

For the product-based and sample-based strategies, we terminated the verification task after detecting a violation, and continued with the next product to be checked. For the family-based strategy, we determined which products contributed to the detected violation, and proceeded with the exploration of the remaining state space that was not associated with these products. [4]

The composition time for our subject systems is negligible compared to the verification time, thus we compare the verification times only (including the generation of counterexamples). That is, we measure five values for each specification to be checked (product-based, single-wise, pair-wise, triple-wise, family-based). Additionally, for the sample-based strategies, we determine what percentage of defective products has been identified (i.e., the *detection rate*), and we log what percentage of products has been checked (i.e., the *sample rate*).

D. Results

In Fig. 5, we illustrate the relative verification times of using the sample-based and the family-based strategies for

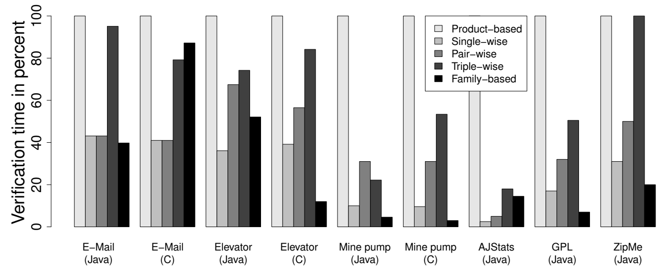

Fig. 5. Comparison of average verification times; product-based strategy defines the 100 %

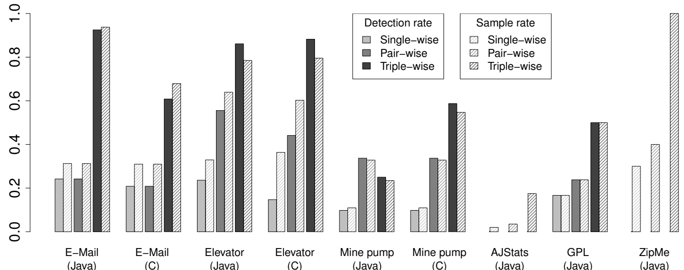

Fig. 6. Detection and sample rates of the sample-based strategies (AJStats and ZipMe do not contain any defect)

each subject system, compared to the product-based strategy (whose time defines the 100 %). As expected, sample-based and family-based strategies can improve verification performance significantly compared to a product-based strategy (except for triple-wise sampling in ZipMe, which effectively selects all possible products): single-wise by 75 %, pair-wise by 60 %, triple-wise by 36 %, and family-based by 73 %, on average.

The family-based strategy is in many cases faster than most of our sampling heuristics; for six subject systems, it outperforms even single-wise sampling. Note that, using a sample-based strategy, we may find only a fraction of all defective products. In Fig. 6, we provide sample and detection rates (side by side, for later comparison) for the different sampling heuristics. The detection rates for AJStats and ZipMe are omitted because they do not contain defects. In Table II and III (cf. Appendix), we provide all raw data for replication.

E. Discussion

We divide our discussion into three parts, regarding (1) the sample-based strategy, (2) the family-based strategy, and (3) a comparison of the two.

Sample-Based Strategy: First, we consider the probability of sample-based strategies to identify defective products. The detection rate depends on the defect and sample rates (which are based on the set $\mathcal{P}$ of valid products of a product line, the non-empty subset $\mathcal{P}_d \subseteq \mathcal{P}$ of defective products, and the subset $\mathcal{P}_s \subseteq \mathcal{P}$ of products that are selected by a sampling heuristic), as well as the subset $\mathcal{P}_f = \mathcal{P}_d \cap \mathcal{P}_s$ of selected products that are actually defect (detected defects). The defect rate r_d is defined as $|\mathcal{P}_d|/|\mathcal{P}|$, the sample rate r_s as $|\mathcal{P}_s|/|\mathcal{P}|$, and the detection rate r_f as $|\mathcal{P}_f|/|\mathcal{P}_d|$. There are no special assumptions about $\mathcal{P}$, $\mathcal{P}_d$, $\mathcal{P}_s$, and $\mathcal{P}_f$.

[4] We could have stopped the verification once a violation was identified, but then we could not have been certain that we identified *all* defective products.

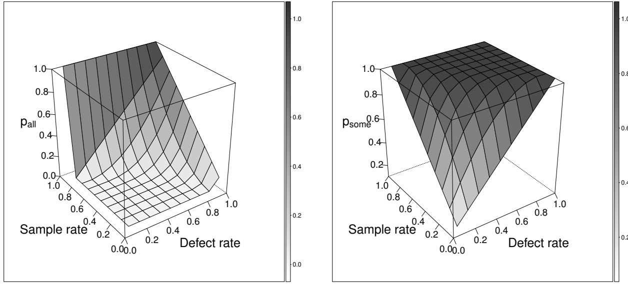

Fig. 7. Probability of detecting *all* defective products (left) and *some* defective products (right)

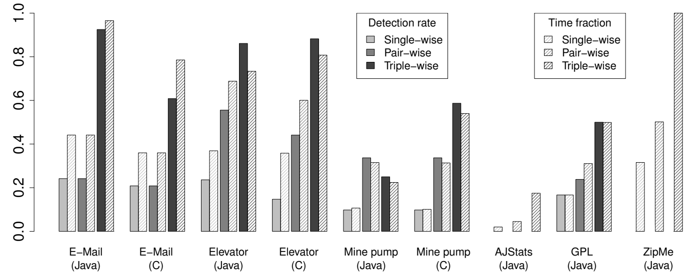

Fig. 8. Detection rates versus fractions of verification time (AJStats and ZipMe do not contain any defect)

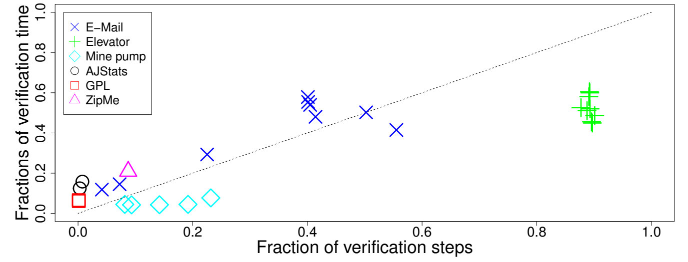

Fig. 9. Fractions of verification steps and verification time using late splitting and early joining; correlation coefficient: 0.84 ($p \ll 0.05$)

The probability q_i to identify exactly i defective products (derived from combinatorics [5]) is given as:

$$q_i = \begin{cases} \dfrac{\dbinom{|\mathcal{P}_d|}{i}\dbinom{|\mathcal{P}|-|\mathcal{P}_d|}{|\mathcal{P}_s|-i}}{\dbinom{|\mathcal{P}|}{|\mathcal{P}_s|}} & \text{if } i \leq |\mathcal{P}_d|,\ i \leq |\mathcal{P}_s|, \\ & \text{and } |\mathcal{P}_s| - i \ \leq\ |\mathcal{P}| - |\mathcal{P}_d| \\ 0 & \text{otherwise} \end{cases}$$

The probability q_{all} to identify all defective products ($r_f = 1$) is then $q_{|\mathcal{P}_d|}$, and the probability q_{some} to identify at least one defective product ($r_f > 0$) is $1 - q_0$. In Fig. 7, we illustrate the corresponding probabilities of detecting all and some defective products depending on varying defect and sample rates.

Note that a single product may contain several defects, and a single defect may be contained in several products. So, identifying a defective product and removing the defect from the product line's code base, may remove the defect also from other products. While this is an advantage for sample-based strategies (which is more likely to detect some defective products quickly; useful in early development stages), it still does not help with verification (which is about providing guarantees that all products are correct; typically, the goal in later development stages).

Next, we consider how the detection rate is related to the sample rate, as illustrated in Fig. 6. On the one hand, the higher the detection rate is for some given fixed defect and sample rates, the better is a sampling heuristic. On the other hand, the lower the sample rate is for some given fixed defect and detection rates, the better is a sampling heuristic. In this light, triple-wise sampling performs best, because the ratio between detection rate and sample rate is large in many cases (cf. Fig. 6).

Finally, we consider how the detection rate is related to the time that is actually needed for verification. Ideally, the goal is to achieve a high detection rate and a low fraction of verification time, compared to the product-based strategy. In Fig. 8, we show for each sample-based verification of our experiments the detection rate and the corresponding time fraction. The higher the detection rate is for a given time fraction that is needed for verification, the better is the sampling heuristic. In our experiments, the triple-wise sampling heuristic has high detection rates, as compared to the time spent for verification; the ratio between detection rate and fraction of verification time is large in many cases (cf. Fig. 6). We get back to this ratio when we compare sample-based and family-based strategies.

Family-Based Strategy: For the family-based strategy, it is more difficult to explain the observed verification times. In our experiments, we observed speedups of up to thirty times (mine pump), compared to a product-based strategy.

A key question—which has not been answered in previous work—is whether late splitting and early joining are the driving factors for the observed speedups, or whether other effects such as internal optimizations in the model checker or technical issues play a dominant role. Hence, we instrumented jpf-bdd to quantify the verification steps saved due to late splitting and early joining. Specifically, for each transition $t \in T$ (where T is the set of transitions in the product line), we computed the number of instructions that it contains (cost C_t) and in how many products P_t it would have been executed. Then $\sum_{t \in T} C_t * (P_t - 1)$ is the number of verification steps (i.e., executed instructions) that are saved due to late splitting and early joining.

In Fig. 9, we illustrate the fraction of verification steps (in relation to the verification steps needed without late splitting and early joining) and the fraction of verification time that the family-based strategy needs (in relation to the product-based strategy). Although the data points are not on the dotted line, a statistical analysis reveals that the fraction of

[5] In the numerator, we count all possible samples that contain exactly i defective products: the first term counts all possible selections of i defective products, the second term counts all possible selections of the remaining products. In the denominator, we count all possible samples. The probability is 0 for all situations that cannot exist.

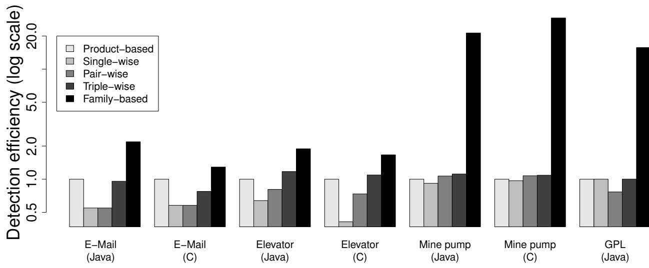

Fig. 10. Detection efficiencies of different verification strategies (AJStats and ZipMe omitted because they do not contain any defect)

verification steps and the fraction of verification time correlate (Pearson's product-moment correlation: $cor = 0.84$, $p \ll 0.05$). This correlation suggests that the principles of late splitting and early joining can explain similarity within product lines (i.e., the amount of shared instructions could be used as similarity degree). However, the time consumed by join operations depends on the size of the BDDs—a factor that causes the deviations in Fig. 9.

Family-Based vs. Sample-Based Strategies: Our experimental data suggest that both the family-based strategy and the sample-based strategies outperform the product-based strategy in terms of verification performance. But which strategy is superior? We cannot compare solely their verification times, but we have to take into account that, for sample-based strategies, the detection rate decreases with the sample rate. Hence, we compare the strategies with regard to *detection efficiency*, which we define as the ratio between detection rate and the time fraction (both in relation to the product-based strategy). A verification strategy with a detection efficiency of one is similarly efficient as the product-based strategy.

In Fig. 10, we show the detection efficiencies for all our experiments, grouped by subject systems. It reveals that the family-based strategy is the most detection-efficient strategy. (The notion of detection efficiency can of course not be applied to AJStats and ZipMe, which do not contain any defect.) Of the sample-based strategies, only triple-wise sampling exceeds in some cases the detection efficiency of product-based verification. The fact that the family-based strategy is mostly superior—in terms of detection efficiency—over our sample-based heuristics is one of the main results of our experiments.

F. Threats to Validity

The kind and distribution of defects threatens internal validity, because they affect the detection rate. If defects occur only if many features interact, then sampling heuristics such as pair-wise are only of limited use. However, the subject systems under investigation contained only defects that occur within single features or among pairs of features, which reflects what is known about the distribution and probability of feature interactions [9], [16], [22].

Much like the kind and distribution of defects, several other characteristics of a product line influence the benefits of the individual strategies. For example, the number and distribution of dependencies among features (as documented in the feature model) can have an influence on the sample and detection rates; the degree of code sharing among products can influence the potential for late splitting and early joining; the granularity of variability may have an effect on the efficiency of join operations (based on BDDs). Further work should develop proper feature-model or code measures to predict the benefits of sample-based and family-based strategies and possibly combinations thereof.

The choice of the subject systems threatens external validity. Hence, we selected as many subject systems as we were able to locate, including standard benchmarks that condense the state-of-the-art in the field. But the tool chain that we used, as well as the availability of product lines that contain specifications and that are amenable to model checking, were limiting factors. Nevertheless, for the first time, a substantial set of different subject systems written in different languages has been considered for evaluating strategies of product-line verification.

IV. Related Work

So far, sample-based and family-based strategies have not been compared systematically in a controlled setting. We discuss related work that focused on either sample-based or family-based strategies. For a comprehensive overview, we recommend a survey report [28].

The sampling heuristics that we considered in this paper are inspired by previous work on feature-interaction detection [9], [15], [23], [26]; they are tailored to pin down execution paths of individual feature interactions, without being distracted by other features (i.e., they aim at small sample products). Alternative sampling heuristics that are used in product-line testing [21] and bug finding [27] have different characteristics and tradeoffs.

By means of case studies [21], [26], [27], it has been shown that a sample-based strategy can have significant performance benefits, while still being able to make reasonable statements about the products of a product line (e.g., non-functional properties and defects). For example, using a pair-wise sampling strategy like ours, the database product-line SQLite could be analyzed in 276 hours [26], with a prediction accuracy of 99.9 %, which was not feasible using a product-based strategy.

The family-based strategy has been used in several model-checking approaches [2], [3], [10], [11], [13], [18], [25], but there is less experience—compared to sampling—with respect to performance gains over using a product-based strategy. Still, it has been shown that substantial performance gains are possible [2], [10], [11]: for example, an average speedup of two was observed [10], [11], by using a family-based strategy, compared to the product-based strategy (and a speedup of sometimes two orders of magnitude if using BDDs for feature variables).

Feature-based verification, which aims at verifying features as far as possible in isolation to minimize the verification effort upon feature composition [19], [20], was not considered in our

study because practical verification tools and corresponding case studies were not available.

Our work is based on state-of-the-art software model-checking technology. We used the explicit-state verification algorithms in CPACHECKER [5] and JAVA PATHFINDER [29], and we encode feature variables in BDDs. Encoding of feature variables in BDDs [10], [30] and BDD-based software model checking of event-condition-action systems [6] have been described before.

V. CONCLUSION

Approaches of product-line verification use different strategies to cope with the specific properties of product lines. Because they all promise benefits, we conducted experiments to compare them with regard to verification performance and the ability to identify defects. Our experiments are based on the SPLVERIFIER tool chain for product-line model checking and six case studies that are implemented in C and Java. We found that the family-based strategy is in almost all case studies superior: the corresponding verification runs produced results faster and the analysis is exhaustive, compared to the considered sample-based strategies (aiming at small sample products). By means of our experimental data and an analytical model, we have discussed the merits of the individual strategies. The success of a sample-based strategy depends on the defect and sample rates, whereas the success of the family-based strategy depends on the similarity between products, represented by the potential for late splitting and early joining. While sampling can reduce the verification time significantly, this does not necessarily increase the effectiveness of verification, because many defective products may be missed. Key results of our experiments are that triple-wise outperformed pair-wise sampling, the family-based strategy outperformed all sample-based strategies in terms of detection efficiency, and that late splitting and early joining are the driving factors for the success of the family-based strategy.

ACKNOWLEDGEMENTS

We thank J. Atlee, A. Classen, C. Kästner, S. Kolesnikov, and T. Thüm for fruitful discussions. This work was supported by the DFG grants AP 206/2 and AP 206/4.

REFERENCES

[1] S. Apel, C. Kästner, and C. Lengauer. FEATUREHOUSE: Language-Independent, Automated Software Composition. In *Proc. ICSE*, pages 221–231. IEEE, 2009.

[2] 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.

[3] P. Asirelli, M. ter Beek, A. Fantechi, and S. Gnesi. A Model-Checking Tool for Families of Services. In *Proc. FMOODS/FORTE*, LNCS 6722, pages 44–58. Springer, 2011.

[4] D. Beyer and M. Keremoglu. CPACHECKER: A Tool for Configurable Software Verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.

[5] D. Beyer and S. Löwe. Explicit-State Software Model Checking Based on CEGAR and Interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.

[6] D. Beyer and A. Stahlbauer. BDD-Based Software Model Checking with CPACHECKER. In *Proc. MEMICS*, LNCS 7721, pages 1–11. Springer, 2013.

[7] I. Cabral, M. Cohen, and G. Rothermel. Improving the Testing and Testability of Software Product Lines. In *Proc. SPLC*, LNCS 6287, pages 241–255. Springer, 2010.

[8] M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003.

[9] M. Calder and A. Miller. Feature Interaction Detection by Pairwise Analysis of LTL Properties: A Case Study. *Formal Methods in System Design*, 28(3):213–261, 2006.

[10] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic Model Checking of Software Product Lines. In *Proc. ICSE*, pages 321–330. ACM, 2011.

[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] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[13] A. Gruler. *A Formal Approach to Software Product Lines*. PhD thesis, Technical University of Munich, 2010.

[14] R. Hall. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering*, 12(1):41–79, 2005.

[15] P. Jayaraman, J. Whittle, A. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *Proc. MoDELS*, LNCS 4735, pages 151–165. Springer, 2007.

[16] M. Kolberg, E. Magill, D. Marples, and S. Reiff. Results of the Second Feature Interaction Contest. In *Feature Interactions in Telecommunications and Software Systems VI*, pages 311–325. IOS Press, 2000.

[17] J. Kramer, J. Magee, M. Sloman, and A. Lister. CONIC: An Integrated Approach to Distributed Computer Control Systems. *Computers and Digital Techniques, IEE Proceedings E*, 130(1):1–10, 1983.

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

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

[20] 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, 2011.

[21] 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.

[22] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal*, 20(3–4):605–643, 2012.

[23] M. Plath and M. Ryan. Plug-and-Play Features. In *Feature Interactions in Telecommunications and Software Systems V*, pages 150–164. IOS Press, 1998.

[24] M. Plath and M. Ryan. Feature Integration using a Feature Construct. *Science of Computer Programming*, 41(1):53–84, 2001.

[25] H. Post and C. Sinz. Configuration Lifting: Verification meets Software Configuration. In *Proc. ASE*, pages 347–350. IEEE, 2008.

[26] 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.

[27] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration Coverage in the Analysis of Large-scale System Software. *Operating Systems Review*, 45(3):10–14, 2011.

[28] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and Gunter Saake. Analysis Strategies for Software Product Lines. Technical Report FIN-004-2012, University of Magdeburg, 2012.

[29] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[30] A. von Rhein, S. Apel, and F. Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. http://www.infosun.fim.uni-passau.de/cl/publications/docs/JPF2011.pdf, 2011.

APPENDIX

TABLE II
PERFORMANCE OF THE VERIFICATION TASKS (GIVEN IN SECONDS OF CPU TIME, THREE SIGNIFICANT DIGITS); ALL VERIFICATION TASKS SUCCEEDED AND PRODUCED CORRECT RESULTS; AN ENTRY '–' INDICATES THAT THE SPECIFICATION WAS NOT APPLICABLE TO THE RESPECTIVE PRODUCT

System	Spec.	Verification time									
		Product		Single-wise		Pair-wise		Triple-wise		Family	
		Java	C	Java	C	Java	C	Java	C	Java	C
E-Mail	1	–	431	–	91.2	–	91.2	–	295	–	378
	2	28.3	98.4	17.0	66.4	17.0	66.4	27.3	83.9	16.4	96.3
	3	11.4	40.7	3.46	12.5	3.46	12.4	10.7	27.7	1.35	43.1
	4	11.6	202	3.52	101	3.52	101	10.8	189	1.68	170
	5	16.7	38.6	4.87	11.9	4.87	11.9	15.6	26.3	4.89	44.5
	6	77.2	426	30.6	137	30.6	137	76.1	329	32.0	229
	7	28.9	207	17.6	98.7	17.6	98.7	27.9	190	16.0	203
	8	31.2	209	17.6	103	17.6	103	30.1	192	16.9	152
	9	81.2	422	30.7	92.2	30.7	92.2	79.0	289	40.8	264
	10	40.4	119	18.8	75.2	18.8	75.2	38.2	101	19.4	120
Elevator	1	81.4	50.2	36.5	14.8	62.5	32.4	52.2	37.7	49.2	4.24
	2	80.2	48.2	35.8	14.1	61.8	31.1	51.4	36.5	48.1	3.68
	3	81.7	52.2	36.3	15.7	62.1	33.9	58.0	39.2	39.8	6.50
	4	156	–	46.6	–	101	–	117	–	79.7	–
	5	83.1	24.1	38.0	11.8	71.6	11.8	52.5	21.8	48.3	3.72
	6	59.6	–	8.90	–	30.1	–	51.8	–	26.8	–
	7	59.4	–	9.03	–	30.1	–	51.7	–	27.0	–
	8	50.0	18.3	17.5	9.11	36.8	9.11	45.8	18.3	26.3	2.85
	9	63.5	25.1	35.3	12.6	35.4	12.6	43.8	22.7	33.0	3.94
Mine pump	1	172	99.6	21.3	10.3	56.8	31.3	38.8	53.4	7.69	3.25
	2	208	106	20.8	10.5	61.8	33.1	44.3	57.6	8.96	3.67
	3	89.5	103	7.63	10.8	28.5	33.0	21.6	55.8	6.89	3.12
	4	183	103	20.8	10.3	58.7	32.0	41.2	55.4	8.20	4.05
	5	235	109	24.3	10.7	74.4	33.6	53.4	58.7	9.98	3.74
AJStats	1	172	–	3.44	–	7.77	–	30.1	–	27.2	–
	2	256	–	5.08	–	11.5	–	44.8	–	32.0	–
GPL	1	38.4	–	6.39	–	11.9	–	19.2	–	2.35	–
	2	40.8	–	6.81	–	12.7	–	20.3	–	2.69	–
ZipMe	1	21.5	–	6.79	–	10.8	–	21.5	–	4.50	–

TABLE III
DETECTION RATES (x / y MEANS 'x DEFECTS FOUND OUT OF A TOTAL OF y DEFECTS') AND SAMPLE RATES (x / y MEANS 'x PRODUCTS CHECKED OUT OF A TOTAL OF y PRODUCTS') OF THE SAMPLE-BASED VERIFICATION EXPERIMENTS; AJSTATS AND ZIPME DO NOT CONTAIN ANY DEFECTS

System	Spec.	Single-wise				Pair-wise				Triple-wise			
		Detection rate		Sample rate		Detection rate		Sample rate		Detection rate		Sample rate	
		Java	C	Java	C	Java	C	Java	C	Java	C	Java	C
E-Mail	1	–	1 / 8	–	6 / 20	–	1 / 8	–	6 / 20	–	4 / 8	–	13 / 20
	2	3 / 14	3 / 14	5 / 16	5 / 16	3 / 14	3 / 14	5 / 16	5 / 16	13 / 14	9 / 14	15 / 16	11 / 16
	3	5 / 16	5 / 16	5 / 16	5 / 16	5 / 16	5 / 16	5 / 16	5 / 16	15 / 16	11 / 16	15 / 16	11 / 16
	4	5 / 16	2 / 12	5 / 16	5 / 16	5 / 16	2 / 12	5 / 16	5 / 16	15 / 16	7 / 12	15 / 16	11 / 16
	5	5 / 16	5 / 16	5 / 16	5 / 16	5 / 16	5 / 16	5 / 16	5 / 16	15 / 16	11 / 16	15 / 16	11 / 16
	6	1 / 8	1 / 8	5 / 16	5 / 16	1 / 8	1 / 8	5 / 16	5 / 16	7 / 8	4 / 8	15 / 16	11 / 16
	7	3 / 14	2 / 12	5 / 16	5 / 16	3 / 14	2 / 12	5 / 16	5 / 16	13 / 14	7 / 12	15 / 16	11 / 16
	8	3 / 14	2 / 12	5 / 16	5 / 16	3 / 14	2 / 12	5 / 16	5 / 16	13 / 14	7 / 12	15 / 16	11 / 16
	9	1 / 8	1 / 8	5 / 16	6 / 20	1 / 8	1 / 8	5 / 16	6 / 20	7 / 8	4 / 8	15 / 16	13 / 20
	10	3 / 14	3 / 14	5 / 16	5 / 16	3 / 14	3 / 14	5 / 16	5 / 16	13 / 14	9 / 14	15 / 16	11 / 16
Elevator	1	1 / 10	1 / 8	6 / 20	6 / 20	5 / 10	4 / 8	13 / 20	13 / 20	9 / 10	7 / 8	15 / 20	15 / 20
	2	1 / 10	1 / 8	6 / 20	6 / 20	5 / 10	4 / 8	13 / 20	13 / 20	9 / 10	7 / 8	15 / 20	15 / 20
	3	1 / 10	1 / 10	6 / 20	6 / 20	5 / 10	5 / 10	13 / 20	13 / 20	8 / 10	8 / 10	15 / 20	15 / 20
	4	0 / 0	–	6 / 20	–	0 / 0	–	13 / 20	–	0 / 0	–	15 / 20	–
	5	1 / 10	1 / 4	6 / 20	5 / 10	5 / 10	1 / 4	14 / 20	5 / 10	9 / 10	4 / 4	15 / 20	9 / 10
	6	4 / 8	–	6 / 16	–	6 / 8	–	10 / 16	–	7 / 8	–	14 / 16	–
	7	4 / 8	–	5 / 16	–	6 / 8	–	10 / 16	–	7 / 8	–	14 / 16	–
	8	4 / 12	0 / 0	5 / 16	4 / 8	7 / 12	0 / 0	10 / 16	4 / 8	10 / 12	0 / 0	14 / 16	7 / 8
	9	1 / 4	1 / 4	6 / 10	5 / 10	1 / 4	1 / 4	5 / 10	5 / 10	3 / 4	4 / 4	7 / 10	9 / 10
Mine pump	1	1 / 20	1 / 20	7 / 64	7 / 64	6 / 20	6 / 20	21 / 64	21 / 64	5 / 20	13 / 20	15 / 64	35 / 64
	2	1 / 8	1 / 8	7 / 64	7 / 64	4 / 8	4 / 8	21 / 64	21 / 64	3 / 8	6 / 8	15 / 64	35 / 64
	3	6 / 48	6 / 48	7 / 64	7 / 64	16 / 48	16 / 48	21 / 64	21 / 64	11 / 48	25 / 48	15 / 64	35 / 64
	4	1 / 16	1 / 16	7 / 64	7 / 64	5 / 16	5 / 16	21 / 64	21 / 64	4 / 16	10 / 16	15 / 64	35 / 64
	5	0 / 0	0 / 0	7 / 64	7 / 64	0 / 0	0 / 0	21 / 64	21 / 64	0 / 0	0 / 0	15 / 64	35 / 64
AJStats	1	–	–	4 / 200	–	–	–	7 / 200	–	–	–	35 / 200	–
	2	–	–	4 / 200	–	–	–	7 / 200	–	–	–	35 / 200	–
GPL	1	7 / 42	–	7 / 42	–	10 / 42	–	10 / 42	–	21 / 42	–	21 / 42	–
	2	7 / 42	–	7 / 42	–	10 / 42	–	10 / 42	–	21 / 42	–	21 / 42	–
ZipMe	1	–	–	3 / 10	–	–	–	4 / 10	–	–	–	10 / 10	–