Product-Line Maintenance with Emergent Contract Interfaces

Thomas Thüm
TU Braunschweig
Germany

Márcio Ribeiro
Federal University of Alagoas
Maceió, Brazil

Reimar Schröter
University of Magdeburg
Germany

Janet Siegmund
University of Passau
Germany

Francisco Dalton
Federal University of Alagoas
Maceió, Brazil

ABSTRACT

A software product line evolves whenever one of its products need to evolve. Maintenance of preprocessor-based product lines is a difficult task, as changes to the code base may unintentionally influence the behavior of uninvolved products. Hence, developers should be supported during maintenance. We present emergent contract interfaces to make product-line development more efficient and less error-prone. The key idea is that for a given maintenance point (i.e., an assignment), we calculate (a) features in the source code that may be affected and (b) assertions based on contracts defined in the code base. By means of a controlled experiment, we provide empirical evidence regarding efficiency and error-avoidance with emergent contract interfaces.

CCS Concepts

•**Software and its engineering** → **Software maintenance tools; Software product lines;**

Keywords

Software product lines, preprocessor variability, evolution, maintenance, design by contract, weakest precondition

1. INTRODUCTION

A software product line is a family of systems developed based on reusable assets to improve productivity and time-to-market [12]. In practice, product lines are typically developed using preprocessors. Due to many products that developers have to deal with, engineering tasks become more challenging than for a single software system. For instance, the high number of feature combinations that need to be checked and tested after every change to unprocessed code makes product-line maintenance difficult [45, 10]. This scenario becomes even harder when product lines contain *cross-feature dependencies*; that is, when features share program elements, such as variables and methods [41]. Developers have to trace these cross-feature dependencies to ensure that a change in one feature does not cause problems to other features [40, 9]. In this context, not recognizing dependencies may lead to errors [11], such as division by zero, and the task of searching for them might occur frequently. In a study on 43 large-scale open-source implementations, we found that feature dependencies are common in practice [41].

In prior work [40], we proposed *emergent interfaces* to reduce errors caused by cross-feature dependencies. An emergent interface is an abstraction of the data-flow dependencies of a feature [39]. This interface contains a set of *provides clauses* and *requires clauses* describing cross-feature dependencies in terms of provided and required data. However, unlike regular interfaces, developers do not need to write emergent interfaces manually. Instead, emergent interfaces are inferred on demand and emerge automatically in the integrated development environment (IDE). When a developer is supposed to change a feature's code, an emergent interface is generated by selecting a set of maintenance points (i.e., code statements to be changed). When analyzing the interface, the developer becomes aware of dependencies between the maintained feature and others. Emergent interfaces help to avoid some errors, as we found in a previous study [39].

Despite the benefits of emergent interfaces, they are not enough for feature modularization, which aims at achieving independent feature comprehensibility and changeability [36]. For example, consider a developer who is supposed to change the value of a variable `x` in the common base code. An emergent interface points out features that use this variable, so that a developer can focus on those features and ignore others. While this is important to decrease the effort of analyzing many features, the developer still needs to analyze features that use `x`, which are likely to be features that are not under the developer's responsibility and expertise. This scenario hinders independent comprehensibility and changeability, and happens due to the lack of semantic information: emergent interfaces do not provide any additional information about `x`, such as the range of valid values for `x`. In fact, recent studies suggest that managing semantic dependencies challenges software development organizations [11].

To reduce this problem, we propose *emergent contract interfaces*, which extend emergent interfaces with *contracts*. In particular, we enrich emergent interfaces with information on the behavior of program elements, which can improve independent comprehensibility and changeability. For example, an emergent contract interface may state that a feature

SPLC '16, September 16-23, 2016, Beijing, China

DOI: http://dx.doi.org/10.1145/2934466.2934471

```
1. public void computeLevel() {
2.   int totalScore = perfectCurvesCounter * PERFECT_CURVE_BONUS + ...
3.                   + perfectStraightCounter * PERFECT_STRAIGHT_BONUS
4.                 + gc_levelManager.getCurrentCountryId();
     ...
5.   #ifdef PENALTIES
6.   totalScore = totalScore - totalLapTime * SRC_TIME_MULTIPLIER;
7.   #endif
     ...
8.   #ifdef ARENA
9.   NetworkFacade.setScore(totalScore);
10.  #endif
     ...
11.  #ifdef SKU
     ...
12.  #endif
     ...
13.}
```

```
public class NetworkFacade {
  ...
  public static void setScore(int totalScore) {
    this.score = totalScore;
  }
  ...
}
```

Figure 1: Extract of product line *Best Lap* that is subject to a maintenance task.

using `x` requires it to be negative. This way, the developer supposed to change the value of `x` knows the valid range of values for `x` without looking at other features. To capture such behavioral information, we rely on contracts as written in languages, such as Eiffel [33], JML [27], or Spec# [5]. As suggested by *design by contract* [33], we assume that such contracts have been previously specified during software development. While this is a strong assumption, contracts have been shown to advance other applications ranging from documentation over runtime assertion checking to formal verification [8, 20].

Compared to emergent interfaces, the goal of emergent contract interfaces is to (i) speed up program comprehension and (ii) reduce errors in the comprehension process. To evaluate these hypotheses, we conducted a controlled experiment to compare emergent interfaces and emergent contract interfaces. For the experiment, we recruited students of a course on software product lines and let them complete comprehension tasks. The results indicate that emergent contract interfaces improve correctness during comprehension tasks, but have no significant effect on answer time.

In summary, we make the following contributions. First, we introduce emergent contract interfaces and a catalog of rules to compute contracts for a given product line and maintenance point. Second, we assess the potential of emergent contract interfaces to improve program comprehension with a controlled experiment with a total of 25 participants working on two product lines.

2. MOTIVATING EXAMPLE

When maintaining software product lines, developers can easily miss cross-feature dependencies resulting in undesired behavior [41]. For example, assume we assign 0 instead of 1 to a variable, which is overridden to 10 if feature A is selected and later used as a divisor if feature B is selected. Then, we introduce division-by-zero fault only in products that have feature B but not A. Such problems are hard to foresee by developers as it is not clear which features of potentially thousand of features are relevant to the change.

To minimize problems related to feature dependencies, we have proposed emergent interfaces previously [40]. An emergent interface is an abstraction to represent data-flow dependencies in terms of *def-use* chains [39]. A feature may provide data to others and require data from others. When reading and analyzing the interface, developers become aware of feature dependencies. Instead of opening and analyzing many features and reasoning about their combina-

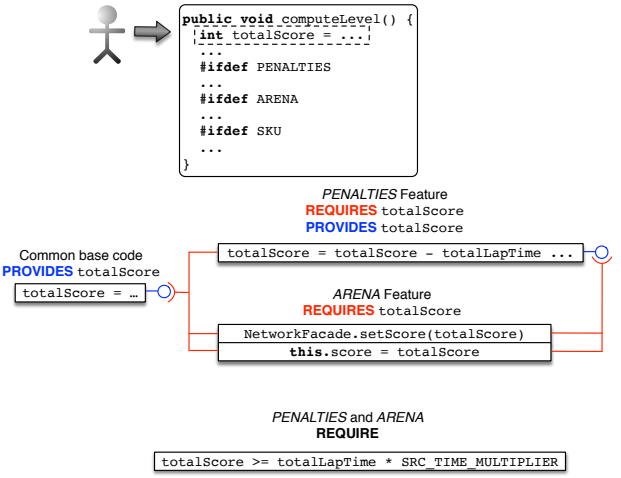

Figure 2: Emergent interface for the initialization of variable `totalScore`.

tions during a maintenance task, developers can look at the automatically derived emergent interface and focus on the features related to the current task, preventing them from analyzing unrelated features and their associated code.

To better understand how emergent interfaces work, consider the scenario illustrated in Figure 1. This scenario shows a commercial car-racing mobile game named *Best Lap* developed by Meantime Mobile Creations. Players must achieve the best lap time to qualify for the pole position. In this game, there is a method named `computeLevel` responsible for computing the game score. Inside this method, we have two optional features: feature *PENALTIES* adds penalties to the player's score (e.g., in case they often crash the car) and feature *ARENA* publishes the score online to encourage competition among players. Assume the developer is supposed to change how the score is calculated. To do so, the developer may change the initialization of variable `totalScore` inside the `computeLevel` method (the dashed rectangle in Figure 1), which happens to be the *maintenance point* for this particular task. That is, a maintenance point describes the code statement a developer intends to change during a code change task. Before changing the variable assignment, the developer consult an emergent interface generated by the integrated development environment to understand the feature dependencies.

Figure 2 illustrates the emergent interface for the given maintenance point. Among some other dependencies not shown for brevity, feature *PENALTIES* requires data from the common base code and provides data to feature *ARENA*. Emergent interfaces provide benefits in terms of reducing effort and number of introduced errors [39]. For instance, as feature *SKU* does not use variable `totalScore`, the interface does not contain this feature, helping developers to focus on the features that indeed use such a variable.

Nevertheless, emergent interfaces may be improved for better feature modularization (i.e., independent comprehensibility and changeability) [36]. Ideally, with assistance of interfaces, a developer is capable of changing and understanding a feature without looking into the code of other features. However, knowing that features *ARENA* and *PENALTIES* require `totalScore` is not enough information for the code change task. In particular, the developer needs to manu-

```
1. public class NetworkFacade {
     ...
2.   //@requires totalScore >= 0;
3.   //@ensures this.score == totalScore;
4.   public static void setScore(int totalScore) {
5.     this.score = totalScore;
6.   }
     ...
7. }
```

Figure 3: A method contract in JML consisting of precondition and postcondition.

ally investigate the source code and documentation of both features to decide whether the planned changes can be applied. To minimize this problem and enable a better feature modularization, we introduce emergent contract interfaces, which improve emergent interfaces by relying on contracts to provide behavioral information about program elements.

3. EMERGENT CONTRACT INTERFACES

We extend emergent interfaces with contracts, and refer to these as *emergent contract interfaces*. Like an emergent interface, an emergent contract interface is derived on demand and contains information about the features that are influenced by a given maintenance point. Furthermore, it provides contracts describing how these features are influenced by the maintenance point.

In Section 3.1, we illustrate how contracts can improve the maintenance task of our running example. Then, we introduce a catalog describing how to determine emergent contract interfaces in Section 3.2. We illustrate how to calculate the emergent contract interface for our running example in Section 3.3. Finally, we discuss limitations of our current solution in Section 3.4.

3.1 Adding Contracts to Emergent Interfaces

In the running example, the maintenance task could be improved by making use of behavioral information. Assume that method `setScore` in feature *ARENA* requires the score to be non-negative. Such behavioral information is typically specified by means of contracts [20]. In Figure 3, we illustrate method `setScore` and its contract. The precondition states that the method may only be called with a positive parameter value and the postcondition states that the caller can rely on that the score is updated correctly.

The contract of method `setScore` is helpful for the abovementioned maintenance tasks, because changing the calculation of the score may potentially lead to negative scores. The idea of emergent contract interfaces is to make developers aware of this contract to reduce errors and speed up maintenance. However, it is not enough to simply show the precondition of method `setScore` in addition to the emergent interface. Rather, we need to additionally incorporate the variability and possible statements between our maintenance point and a contract.

In the example, method `computeLevel` contains three optional features, yielding eight possible products. Hence, an emergent contract interface should take these eight configurations into account. For that, we enrich contracts with application conditions. For instance, the precondition `totalScore >= 0` only applies to our maintenance point in Figure 1 if $ARENA \wedge \neg PENALTIES$. However, if both features are selected, the effective precondition at the maintenance point is `totalScore - totalLapTime * SRC_TIME_MULTIPLIER >= 0`, because feature *PENALTIES* changes the value of variable `totalScore` before method `setScore` is called. For all remaining configurations, the contract of method `setScore` does not apply to our maintenance point, as the code of feature *ARENA* is not included.

Having such behavioral information for a maintenance point may reduce the time spent on understanding feature code and prevent the developer from introducing errors for particular feature interactions. In our running example, the emergent contract interfaces makes the behavioral interaction of features *PENALTIES* and *ARENA* explicit.

3.2 Retrieving Emergent Contract Interfaces

We describe how emergent contract interfaces can be retrieved for a method body and a given maintenance point therein. To this end, our analysis is an intra-procedural analysis, which, nevertheless, takes contracts of called methods into account and can be extended to an inter-procedural analysis. Our formalization is based on a catalog of typical structural patterns that occur inside a method body (e.g., branching statements or loops). An emergent contract interfaces is derived by applying these patterns iteratively.

We assume that each statement in the method body has an annotation stating in which configurations the statement is present. This is a simplified view with regard to preprocessor-based product lines, because they may contain nested or undisciplined annotations [28]. However, undisciplined annotations, such as `ifdef` statements surrounding parts of a method name, can be transformed into disciplined annotations by replication [24] or by using macros and auxiliary variables [32]. Furthermore, nested `ifdef`s can be eliminated by conjoining their annotations for each statement. Hence, assuming an annotation for each statement is not a limitation of our approach, but simplifies our discussions.

Our catalog is based on the weakest precondition calculus [14], a famous calculus to verify programs. In essence, given the postcondition of a program, the weakest precondition is calculated that has to be fulfilled at beginning of the program, such that the postcondition will be fulfilled after running the program. Given a weakest precondition, one only has to verify that the given precondition implies the weakest precondition. Algorithmically, the weakest precondition is calculated in a step-wise manner; the algorithm starts with the postcondition and calculates the weakest precondition for the last statement of the program, then for the second last, and so on.

While we are not interested in program verification here, we can still apply the weakest precondition calculus to parts of a method. In particular, we focus on those parts between a maintenance point and the next statement being influenced by our maintenance point. If that statement refers to a contract, we use it as the basis to calculate an assertion that has to hold at the maintenance point. However, in contrast to the weakest precondition calculus, we traverse statements in their execution order and not vice versa.

Table 1 presents our catalog for the retrieval of emergent contract interfaces. Function $i_v(s)$ recursively retrieves a predicate on the variable v (maintenance point) based on the remaining statements s of the method body. The predicate $o(v, s)$ states whether the variable v syntactically occurs in the statement s. The function $pre_m(s_1, \ldots, s_n)$ returns the precondition of method m, whereas m's parameters are

Control structure	Contract retrieval
Sequential composition $s := s_0; s_1$	$i_v(s) := \begin{cases} i_v(s_1) & \text{if } \neg o(v, s_0) \\ i_v(s_0) & \text{if } fm \wedge pc(v) \models pc(s_0) \\ i_v(s_1) & \text{if } fm \wedge pc(v) \models \neg pc(s_0) \\ \big(pc(s_0) \Rightarrow i_v(s_0)\big) \wedge \big(\neg pc(s_0) \Rightarrow i_v(s_1)\big) & \text{otherwise} \end{cases}$
Conditional $s := \text{if } c \text{ then } s_0 \text{ else } s_1$	$i_v(s) := \begin{cases} i_v(c) & \text{if } o(v, c) \\ (c \Rightarrow i_v(s_0)) \wedge (\neg c \Rightarrow i_v(s_1)) & \text{otherwise} \end{cases}$
Loop $s := \text{while } c \text{ do } \{s_0\}$	$i_v(s) := \begin{cases} i_v(c) & \text{if } o(v, c) \\ c \Rightarrow i_v(s_0) & \text{otherwise} \end{cases}$
Constant $s := c$	$i_v(s) := \text{true}$
Variable $s := v_0$	$i_v(s) := \text{true}$
Assignment $s := (s_0 = s_1)$	$i_v(s) := \begin{cases} i_v(s_1) & \text{if } o(v, s_1) \\ \text{true} & \text{otherwise} \end{cases}$
Field access $s := s_0.f$	$i_v(s) := i_v(s_0)$
Method call $s := s_0.m(s_1, \ldots, s_n)$	$i_v(s) := \begin{cases} i_v(s_0) & \text{if } o(v, s_0) \\ pre_m(s_1, \ldots, s_n) & \text{if } \bigvee_{1 \le i, i \le n} s_i = v \\ \text{true} & \text{otherwise} \end{cases}$
Unary operator $s := op_u\ s_0$ with $op_u \in \{\texttt{+}, \texttt{-}, \texttt{!}, \ldots\}$	$i_v(s) := i_v(s_0)$
Binary operator $s := s_0\ op_b\ s_1$ with $op_b \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{\textbackslash}, \texttt{\%}, \texttt{==}, \texttt{!=}, \texttt{<}, \ldots\}$	$i_v(s) := \begin{cases} i_v(s_0) & \text{if } o(v, s_0) \\ i_v(s_1) & \text{otherwise} \end{cases}$

Table 1: Catalog of typical patterns for the retrieval of emergent contract interfaces.

replaced by s_i, respectively. The function $pc(s)$ returns the presence condition of a given statement s and fm refers to a propositional formula specifying the feature model. A feature model is typically used to define valid combinations of features [6]. If there is no feature model, then $fm = \text{true}$.

The most interesting case in Table 1 is the sequential composition of statements, where we distinguish four cases. First, if the variable v to be changed in our maintenance task does not occur in s_0, then we continue recursively with statement s_1. Second, if v occurs in s_0 and s_0 is always present when our maintenance point is present, we recursively continue with that statement and ignore s_1. Third, if v occurs in s_0, but s_0 is never present together with our maintenance point, we ignore s_0 and continue with s_1. Fourth, if v occurs in s_0 and is present in some configurations, we retrieve the contract for both statements s_0 and s_1. The presence condition of both results is directly encoded into the formula.

Conditional statements and loops are handled in a similar manner, whereas we do not have to consider presence conditions, because we assume disciplined annotations (as explained above). If variable v is contained in the condition c of the loop or the conditional statement, we continue with the condition and ignore the remaining part. Otherwise, we encode under which circumstances the following statements are executed and again use a recursive call. If we continue this process and only have a variable remaining, we cannot compute any contract and return true. For an assignment, we continue with the right-hand side, if it contains our variable of interest. For a field access, we always continue with the object at which the field is accessed.

Finally, contracts are taken into account in the case of a method call. If our variable of interest is passed as an argument to a method, we take the precondition of that method, replace all parameters by the given arguments, and return it. However, if variable v is already contained in s_0, we ignore this method call and continue with s_0, as it is executed before.

3.3 Example Retrieval

We illustrate our catalog by applying it to a slightly simplified version of our running example. To make application of rules more readable, we abbreviate statements as follows:

$$
\begin{array}{rcl}
v & := & \texttt{totalScore} \\
s_a & := & s_b; s_c; s_d \\
s_b & := & \texttt{totalScore} = \texttt{totalScore} - \\
 & & \texttt{totalLapTime} * \texttt{SRC\_TIME\_MULTIPLIER} \\
s_c & := & \texttt{NetworkFacade.setScore(totalScore)} \\
s_d & := & \texttt{System.out.println("Done.")}
\end{array}
$$

As a first step, we retrieve all statements in the method body below our maintenance point. At the beginning, we can only apply the rule for sequential composition. First, statement v contains `totalScore`. Second, with $pc(v) =$ `true`, $pc(s_b) = \mathit{PENALTIES}$, and assuming there are no constraints from the feature model (i.e., $fm = \text{true}$), we know that neither $fm \wedge pc(v) \models pc(s_b)$ nor $fm \wedge pc(v) \models \neg pc(s_b)$. Hence, the result of applying the sequential composition rule is the following:

$$i_v(s_a) = \big(\mathit{PENALTIES} \Rightarrow i_v(s_b)\big) \wedge \big(\neg \mathit{PENALTIES} \Rightarrow i_v(s_c; s_d)\big)$$

Next, we need to handle the recursive call $i_v(s_b)$. Applying rules for assignment, binary operators, and variables lead to $i_v(s_b) =$ `true` indicating that we cannot retrieve any contracts from this statement. In addition, we have a recursive call $i_v(s_c; s_d)$ to which we apply the composition rule:

$$i_v(s_c; s_d) = \big(\mathit{ARENA} \Rightarrow i_v(s_c)\big) \wedge \big(\neg \mathit{ARENA} \Rightarrow i_v(s_d)\big)$$

For statement s_c, we can actually include a precondition with the rule for method calls. As `NetworkFacade` does not contain v and the argument `totalScore` is indeed equal to v, we insert the argument into the precondition of method `setScore` resulting in $i_v(s_c) = \texttt{totalScore >= 0}$. Finally, to retrieve the contract for s_d, we apply the rules for field access and method call to retrieve $i_v(s_d) = \texttt{true}$, which means there are no assumptions for this statement. In the following, we insert all formulas and do some propositional simplifications to get the final contract:

$$\begin{aligned} i_v(s_a) &= (\mathit{PENALTIES} \Rightarrow \texttt{true}) \wedge \\ &\quad (\neg \mathit{PENALTIES} \Rightarrow i_v(s_c; s_d)) \\ &= \neg \mathit{PENALTIES} \Rightarrow i_v(s_c; s_d) \\ &= \neg \mathit{PENALTIES} \Rightarrow (\mathit{ARENA} \Rightarrow i_v(s_c)) \wedge \\ &\quad (\neg \mathit{ARENA} \Rightarrow i_v(s_d)) \\ &= \neg \mathit{PENALTIES} \Rightarrow (\mathit{ARENA} \Rightarrow i_v(s_c)) \wedge \\ &\quad (\neg \mathit{ARENA} \Rightarrow \texttt{true}) \\ &= \neg \mathit{PENALTIES} \Rightarrow (\mathit{ARENA} \Rightarrow i_v(s_c)) \\ &= \neg \mathit{PENALTIES} \wedge \mathit{ARENA} \Rightarrow i_v(s_c) \\ &= \neg \mathit{PENALTIES} \wedge \mathit{ARENA} \Rightarrow \texttt{totalScore >= 0} \end{aligned}$$

The resulting emergent contract interface states that if feature *PENALTIES* is not selected and feature *ARENA* is selected, we have to ensure that the total score is non-negative. Hence, the developer is aware that changing the assignment at the maintenance point may conflict with other features, which are also mentioned in the emergent contract interface itself. For comparison, an emergent interface would only state that *ARENA* uses `totalScore` and the developer would need to look into the feature code and the method `setScore` to find out that the value must be positive.

3.4 Principle and Current Limitations

The presented catalog illustrates how to calculate emergent contract interfaces for the most common language constructs, based on preconditions of methods only. For a real programming language and contract-specification language, this catalog has to be extended and could also incorporate class invariants, inheritance and so on [20], but such specification concepts are out of the scope of this paper.

Furthermore, one has to be aware that emergent contract interfaces may produce false negatives by definition. That is, they may not prevent from all errors that these contracts may reveal when checked for products at runtime. The reason is that the presented catalog is a heuristic and incorporates some contracts, while it necessarily ignores others. Otherwise, the emergent contract interface would not be determinable on arbitrary programs due to loops and recursion. However, such abstractions are common for dataflow analyses to cope with the inherent undecidability problem [35]. While false negatives are acceptable for emergent contract interfaces, false positives should be avoided completely. That is, showing a contract to the developer that actually does not apply to the maintenance point could have even a negative effect on program comprehension.

In the catalog, we need to retrieve the contract for a given method call. However, this is non-trivial for two reasons: First, identifying the definition that is actually called may depend on the feature selection and requires a variability-aware type system [23, 45]. If there are several definitions for that method, such as alternative definitions, they may also come with different contracts that need to be incorporated into the emergent contract interface in a similar way as the presence conditions in the concatenation rule. Second, with a statically-typed language, we should rely on behavioral subtyping [30], when retrieving contracts of a given method. The reason is that we might know the type of variables, but the actual value may be of a subtype, which can only be determined at run-time.

For our catalog, we assume that contracts do not themselves contain variability. That is, we do not yet support ifdef-blocks inside contracts. Whereas there are no studies on the necessary variability within contracts in preprocessor-based product lines, results on feature-oriented product lines suggest that variability in contracts are needed [44]. However, our catalog can be extended by allowing annotations of preconditions and postconditions. Similar as for statements, we can restrict the annotations to be disciplined and then extend the patterns presented in this paper accordingly. Disciplined annotations could mean to only permit a complete precondition to be annotated, whereas annotating parts of it is forbidden. Undisciplined annotations in contracts can be translated into disciplined annotations by means of repetition as for source code [24].

4. EMPIRICAL EVALUATION

We describe the controlled experiment that we conducted to evaluate emergent contract interfaces. In a nutshell, we let two groups of computer-science students answer comprehension questions with and without emergent contract interfaces, and evaluate the speed and accuracy with which they answered the questions. In a previous experiment, we already confirmed advantages of emergent interfaces over not using emergent interfaces [39]. Based on this result, we decided to compare emergent contract interfaces with emergent interfaces, to reduce the exploration space. In the following, we use the guidelines suggested by Jedlitschka and others [21]. To support replication, all experimental material is available at the project's website.[1]

Objective In Section 3, we argued why emergent contract interfaces may provide a benefit for comprehension. In particular, we assume that the answer time and correctness are improved, because developers see all relevant information at one place, without having to search the source code. Thus, we investigate the following research hypotheses:

RH1: Emergent contract interfaces *speed up comprehension* compared to emergent interfaces.

RH2: Emergent contract interfaces *improve correctness of comprehension* compared to emergent interfaces.

Variables As independent variable, we have our approach with two levels, either emergent contract interfaces (ECI) or emergent interfaces (EI). As dependent variable, we consider program comprehension based on tasks, measured in terms of answer time and correctness. The most relevant confounding parameters are summarized in Table 2. First, we controlled for programming experience, because this is the major confound when measuring program comprehension. To this end, we used the programming-experience questionnaire we developed [16]. Second, we kept the programming

[1] https://sites.google.com/a/ic.ufal.br/emergent-contract-interfaces/

Parameter	Control technique		Measured/ensured	
	How?	Why?	How?	Why?
Programming experience	Analyzed afterwards	Major confound	Programming-experience questionnaire	Reliable way to measure it
Programming language	Constant	Reliable	Languages participants knew	No training necessary
Familiarity with tools	Constant	Reliable	Tool familiar to participants	Realistic setting that does not require too much familiarity with tool

Table 2: Selection of confounding parameters, their measurement, and control techniques.

language constant by using only Java, as participants are most familiar with it. Last, we controlled for familiarity with tools by using Eclipse as the IDE and Antenna as preprocessor, which participants used in the course anyway.

Material As material, we selected two medium-sized software product lines, *Best Lap* and *Bomber*, which have been used in previous studies [17, 38]. *Best Lap* is our running example and a commercial product line with approximately 15 KLOC. *Bomber* is a mobile game for Nokia series 60 with approximately 10 KLOC. Both product lines are written in Java, our language of choice, and have a suitable size, so that participants neither can understand the source code at first glance, nor need too much time to understand it.

We presented the product lines in two adapted Eclipse versions—one for each product line—with an additional plug-in that we implemented to record time and participants' answers. Our plug-in checks the answers against the right ones that we previously defined for the multiple-choice comprehension questions. The plug-in prompts these questions when the participant presses the finish button to indicate the task is completed. However, when a participant enters an incorrect answer (i.e., the task is not complete yet), the plug-in does not stop the time and increases the number of errors for that participant by one. Then, we let the participant continue until the correct answer is entered. The time limit was set to 60 minutes, but no participant reached this limit. Participants were instructed to start time recording when starting a task. The plug-in stops time measurement automatically once the participant enters the correct answer. We recorded a screencast during the experiment to inspect the behavior of participants after the experiment.

In a debriefing questionnaire, we assessed programming experience of participants and perceived task difficulty.

Participants As participants, we selected students of a course on software product lines at the University of Magdeburg, Germany. In the course, we taught students, among others, the underlying concepts of software product lines, emergent interfaces, and contracts, so students have sufficient background knowledge.

In Table 3, we show the four experimental groups to which we randomly assigned the 25 participants. We switched the order of the product line and the interface, so that learning and ordering effects rule out. Participants were aware that they took part in an experiment, and that their performance does not affect the grade for their course. Participants were required to participate as part of the lecture and in exchange could omit one homework assignment. In the right column of Table 3, we show the average programming experience of each group according to the programming-experience questionnaire that we used [16]. For individual students, the values ranged from 0.727 to 3.635. A Kruskall-Wallis test [4] did not reveal any significant differences between groups ($\chi^2 = 1.643$, $df = 3$, $p = .6497$).

Group	Session 1	Session 2	APE
A	*Best Lap*, ECI	*Bomber*, EI	1.723
B	*Best Lap*, EI	*Bomber*, ECI	1.776
C	*Bomber*, ECI	*Best Lap*, EI	2.068
D	*Bomber*, EI	*Best Lap*, ECI	1.804

Table 3: Experimental design and average programming experience (APE) of participant groups.

Tasks Our experiment contains two sets of tasks, namely tasks related to one feature (Set 1) and tasks related to two features (Set 2). For each product line, we designed one task of each set. In those four tasks, participants were given an uninitialized variable and should indicate in multiple-choice comprehension questions the valid ranges to correctly initialize the variable without causing any error to the product-line features. Uninitialized variables are a common source of errors in product lines [1, 31].

As example, we describe the task for product line *Bomber* in Set 1. In this task, participants should determine valid values to initialize `GAME_DEBRIS` at the following assignment: `FORCE_ANG = CRATER_SIZE * GAME_DEBRIS`. The emergent contract interface for this maintenance point is "Feature *CRATER* requires `FORCE_ANG` $\geq$ `0 && FORCE_ANG` $\leq$ `360 * Common.FIXED`." By analyzing the value of the constants `CRATER_SIZE` and `Common.FIXED`, participants could determine valid values for `GAME_DEBRIS` that maintains `0` $\leq$ `FORCE_ANG` $\leq$ `360 * Common.FIXED`. In contrast, the emergent interface only states that "Feature *CRATER* requires `FORCE_ANG`." Thus, it did not provide any information regarding the valid range for the variables, but required participants to locate the corresponding source-code fragments to determine the correct initialization values.

The remaining tasks can be found at the project's website and follow a similar pattern, with the task description adapted to the interface (ECI vs. EI) and product line (*Bomber* vs. *Best Lap*). In addition to those four tasks, we had a warm-up task in a different system to illustrate the experiment procedure and to introduce the tooling. The results of this task were not analyzed.

Execution We conducted the experiment in December 2013 as part of a regular lecture session in a computer lab. Since there were not enough computers for all participants, we conducted the experiment in two consecutive appoint-

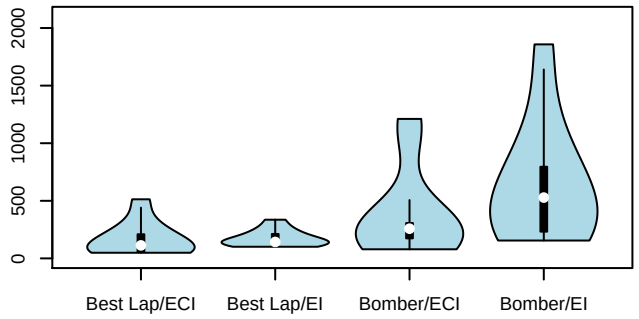

Figure 4: Answer times (in s) for Set 1.

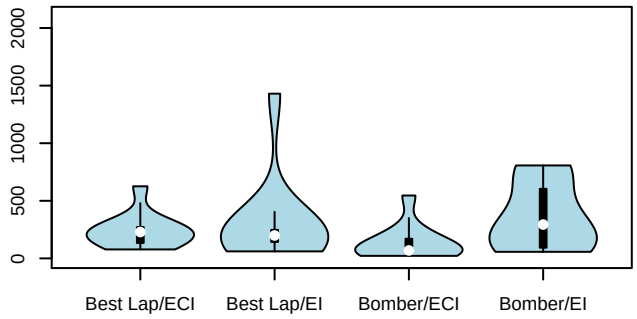

Figure 5: Answer times (in s) for Set 2.

ments at the same day without any break in between. We started with a general introduction to the purpose of the experiment without revealing our hypotheses, and completed the warm-up task with all participants. Then, participants worked on the tasks on their own. After the tasks were finished, participants completed the debriefing questionnaire.

Deviations Some participants forgot to start the time recording. From the screencasts, we could reconstruct the task duration for those participants. However, for three participants, the screencast was not recorded correctly. We excluded the data of those participants to reduce bias. All values reported on programming experience incorporate only the remaining 22 participants (cf. Table 3).

5. RESULTS AND DISCUSSION

In this section, we present and discuss the results. We first present descriptive statistics and significance test for the research hypothesis, and continue in a separate section with the discussion of the data. This way, we allow the readers to draw their own conclusion about our data. All data and analysis scripts are available on the project's website.

5.1 Analysis

RH1: Emergent contract interfaces *speed up comprehension* compared to emergent interfaces. For each product line, we have one task of each set (i.e., related to one or two features). We illustrate the results for each set separately. Figure 4 presents the time measurements for Set 1 per product line. On average, participants completed the tasks of Set 1 about 1.5 times faster with emergent contract interfaces. However, we observe larger differences for tasks of product line *Bomber*. These results indicate an interaction between the interfaces (i.e., EI and ECI) and the product lines (i.e., *Bomber* and *Best Lap*).

To evaluate the first hypothesis, we conducted an 2 x 2 ANOVA with the two variables interface and product lines. Since our sample size is large enough (i.e., 11 participants for each set), the ANOVA is robust against non-normally distributed data. We summarize the results in Table 4. For Set 1, there is a significant effect of the product line, but not for the interface. Thus, for Set 1, the interface had no effect on answer time, but the product line had. Specifically, participants completed the tasks faster in *Best Lap* than in *Bomber*. The interaction of both variables is not significant.

We present the results for Set 2 in Figure 5. Here, participants were, on average, 1.8 times faster with emergent contract interfaces. The average times of both techniques

Set	Factor	df	F	p-value
1	Interface	1	1.7613	0.192
1	Product line	1	11.3093	0.002
1	Interface & product line	1	1.4983	0.228
2	Interface	1	3.1761	0.082
2	Product line	1	0.0820	0.776
2	Interface & product line	1	1.0394	0.314

Table 4: ANOVA for Set 1 and Set 2.

to complete the tasks in *Best Lap* are similar. As for Set 1, we also observe the interaction effect between the interface and product line. An ANOVA did not reveal any significant differences, neither for the interface, nor for the product line (cf. Table 4). There is also no interaction effect. Thus, we cannot accept the first research hypothesis.

RH2: Emergent contract interfaces *improve correctness of comprehension* compared to emergent interfaces. We present the results for the number of errors per participant in both sets of tasks in Table 5. With emergent interfaces, participants made 85 errors in total; with emergent contract interfaces, they made even less than half as much errors (36). For Set 1, participants made in general more errors for *Bomber* and, in Set 2, there are more errors with emergent interfaces, compared to emergent contract interfaces, indicating that only for two features of interest, contracts support developers in making less errors.

To evaluate whether there is a benefit for correctness, we conducted a Fisher's exact test [4]. The difference in favor of ECI is significant for Set 1 ($p = 0.048$, $Z = 1.979$) and for Set 2 ($p = 0.001$, $Z = 3.208$). That is, for one and two features, ECI helped participants to avoid errors. Thus, we can accept the second research hypothesis.

Set	Product line	Interface		Sum
		EI	ECI	
1	*Best Lap*	3	6	9
1	*Bomber*	35	13	48
1	Sum Set 1	38	19	57
2	*Best Lap*	24	16	40
2	*Bomber*	23	1	24
2	Sum Set 2	47	17	64
1+2	Sum Set 1+2	85	36	121

Table 5: Errors by interface and product line.

5.2 Discussion

In our tasks, participants should indicate valid ranges to initialize variables without introducing any problem to the features that depend on these variables. Although we found benefits in favor of ECI regarding correctness, the improved answer times with ECI were not significant.

While ECI provide significant advantages with respect to correctness for both sets, the reduction of errors was larger for Set 2, in which participants had to deal with two features. Errors were reduced by 50% for one feature and by 64% for two features. For Set 2, we found in some screencasts that participants committed errors because they forgot to check the upper bound of a variable, which is in another feature than the lower bound. We believe that this kind of error is minimized with ECI, since both bounds and their respective features are presented to the participants.

In the first set of tasks, answer times were not significantly different regarding the interface, but with respect to the product line. Looking at the details of *Best Lap* and *Bomber* tasks, we can see that the task related to *Bomber* is slightly more complex in the sense that the method-call depth to reach the information about the valid ranges is deeper when compared to the *Best Lap* task (2 vs. 1). This might explain the difference between *Best Lap* and *Bomber*. When looking at the screencasts for both tasks, we found that the time to reach the first method call is roughly comparable in both product lines. Hence, the differences are likely to be caused by the different method-call depth.

For the second set, we found that the difference between answer times is larger in favor of ECI, but only marginally significant. In this set, participants had to work with two features at the same time. On average, ECI leads to better answer time than EI for both sets, with a slightly larger difference for two features than for one feature. This is likely to happen because developers using EI need to compute and reason about variable ranges for different features and their combinations. In contrast, when using ECI, they already have such information. Thus, further experiments are needed to evaluate larger numbers of involved features. In particular, while we controlled the number of features already as a variable, larger numbers have not yet been considered, but would be more realistic for large product lines.

Even though we could not accept research hypothesis RH1, most participants noted that ECI helped them to complete the tasks. We believe that in real-world product lines with more than two features, ECI can provide even more benefit because reasoning about semantic information that features and their combinations share with each other gets more difficult and challenging in these product lines, which makes techniques like ECI important to support maintenance. In particular, we need further studies to deepen our understanding of how ECI affect product-line maintenance. Specifically, with larger, more realistic product lines and professional developers we might learn valuable insights into how we can improve maintenance of product lines.

5.3 Threats to Validity

We annotated the features of *Bomber*, which threatens internal validity in the sense we might have chosen code snippets and functionalities where we could apply contracts with some computation to reason about variable ranges. This choice may lead to complex computations where we would favor emergent contract interfaces. However, we reduced this threat by considering simple ranges that depend only on very few constants scattered throughout the code. Also, the product line may not have typical characteristics and thus our results cannot be generalized. We minimized these threats by considering a second commercial product line with *Best Lap*. All students reported that they have never seen the source code of both product lines before.

To avoid bias due to incorrect time measurement (cf. Section 4), we excluded three participants from the analysis (they forgot to start the time recording), ending up with 22 participants. This exclusion does not introduce bias to our data, as, fortunately, the number of remaining participants was balanced in accordance to the product line and interface (i.e., groups A and C contain together 11 participants).

Regarding external validity, there are threats caused by the selection of participants and Java as programming language. With respect to participants, we tried to address this threat by recruiting students who were enrolled in a product-line course, in which they completed theoretical and practical assignments regarding variable software. Thus, although students might not be as familiar with variable software as professional developers, our participants have at least fundamental experience with it. As our students typically have part-time jobs during their studies, our results may even be generalizable to junior developers. Java is an often used programming language, also in the context of variable software systems [3, 2]. Thus, our results are applicable for a considerable amount of practically relevant product lines. Also, our tasks are simple, requiring only a few minutes to be accomplished. Nevertheless, they are realistic to some extent as uninitialized variables cause real bugs in product lines.[2]

6. RELATED WORK

Virtual separation of concerns CIDE is a tool that avoids pollution of product line source code by removing `ifdef` statements and considering background colors instead, making the source code easier to read and understand [22]. CIDE relies on virtual separation of concerns, where developers can hide the feature code not relevant to the current task. However, hiding the entire feature code is problematic, since developers are not aware of feature dependencies. To increase awareness, emergent contract interfaces complement virtual separation of concerns in the sense that dependencies between the maintenance point and hidden features are made explicit. Prior experiments showed that using `ifdef` statements is harmful when considering comprehension [26] and that background colors can indeed improve program comprehension, independently of size and language of the projects they used [15].

Interfaces for product lines To ease maintenance, other kinds of interfaces have been proposed for product lines. In prior work [42], we present feature context interfaces to derive members, which can be safely accessed within a feature module. Similar to emergent contract interfaces, feature context interfaces emerge for a given maintenance point. Cafeo et al. propose to derive feature interfaces from a product-line implementation [9]. They focus on dependencies between all features, while we focus on a maintenance point and make use of the feature model. Kästner et al. offer a module system for preprocessor-based product lines,

[2]https://bugzilla.kernel.org/show_bug.cgi?id=59521

whereas a module specifies features and members visible to other modules [25]. However, all these approaches consider only structural dependencies between features, whereas we also incorporate contracts defining behavioral dependencies. Similarly, contracts have also been proposed to enable modular reasoning for aspects [37, 34], whereas we focus preprocessors, as they are often used to implement product lines.

Contracts for product lines Emergent contract interfaces heavily rely on the fact that a product line is specified with contracts. However, several researchers propose to specify product lines by means of contracts. Contracts in product lines are usually variable (i.e., contracts may depend on the feature selection) [44]. However, to simplify reasoning by humans and verifiers, features (and aspects) should not always be able to arbitrarily change contracts [44, 19, 34]. Contracts in product lines have been used for verification by means of static analysis [44], theorem proving [7, 13, 44], model checking [44] and runtime assertion checking [46]. Furthermore, contracts can simplify both, the development within a product line [46] and across product lines [47] by acting as a behavioral interface between implementation artifacts. With our experiment on emergent contract interfaces, we give further evidence that contracts for product lines are useful. To reduce the burden of exhaustively specifying contracts, there are cases in which contracts can be derived from the implementation [18].

Analysis of product lines One of the main goals of emergent contract interfaces is to avoid errors in development. Still, emergent contract interfaces cannot prevent from *all* errors for three reasons. First, contracts typically do not completely specify the behavior of a product line. Second, to automatize the retrieval of emergent contract interfaces without running into undecidability problems, we use a heuristic, which may miss certain contracts. Third, developers may still misunderstand or ignore contracts. Hence, it is still necessary to analyze the product line for errors. The area of product-line analysis and testing is an active research area [45, 10]. Ideally, emergent contract interfaces should be combined with product-line analyses and testing.

Refactoring of product lines While we focus on maintenance of assignments, refactoring is another typical maintenance tasks. Refactoring of product lines makes sure that the semantics of all products is preserved [43, 29]. For refactoring, it can be ensured that behavior is preserved, while maintenance on assignment statements typically includes intended changes to behavior. Hence, only heuristics are feasible to support developers in maintaining assignments.

7. CONCLUSIONS AND FUTURE WORK

We presented emergent contract interfaces, an extension of emergent interfaces to improve independent feature comprehensibility and changeability. While emergent interfaces use dataflow analyses to retrieve which features are potentially influenced by a change in a product line, we enrich them by behavioral specifications in terms of contracts. This way, depending on the task, a developer can reason about feature dependencies by only looking at the interface; there is no need to analyze the code of other features potentially not under the developer's responsibility. We also present a catalog of typical patterns for the retrieval of emergent contract interfaces for the most common language constructs.

We designed and conducted an experiment to evaluate the potential of emergent contract interfaces to improve program comprehension. The majority of the 25 participants reported that emergent contract interfaces were more helpful than emergent interfaces. While we measured significant improvements for correctness, answer time was not significantly faster. As this is most likely due to the small number of features in our experiment, we plan to replicate the experiment with more realistic tasks involving more than two dependent features. Future work may also evaluate scalability, how to include multiple maintenance points, and improved heuristics with respect to false positives and false negatives.

8. ACKNOWLEDGMENTS

We thank Paulo Borba and Henrique Rebêlo for discussions on emergent contract interfaces and gratefully acknowledge the constructive reviews. This work is partly funded by CNPq 460883/2014-3, 573964/2008-4, CAPES/PROCAD 175956, DEVASSES PIRSES-GA-2013-612569, BMBF 01IS14017B, and DFG SI-2045/2-1.

9. REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *ASE*, pages 421–432. ACM, 2014.

[2] V. Alves, I. Cardim, H. Vital, P. Sampaio, A. Damasceno, P. Borba, and G. Ramalho. Comparative Analysis of Porting Strategies in J2ME Games. In *ICSM*, pages 123–132. IEEE, 2005.

[3] V. Alves, P. Matos, L. Cole, P. Borba, and G. Ramalho. Extracting and Evolving Mobile Games Product Lines. In *SPLC*, pages 70–81. Springer, 2005.

[4] T. Anderson and J. Finn. *The New Statistical Analysis of Data*. Springer, 1996.

[5] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and Verification: The Spec# Experience. *Comm. ACM*, 54:81–91, 2011.

[6] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, pages 7–20. Springer, 2005.

[7] D. Bruns, V. Klebanov, and I. Schaefer. Verification of Software Product Lines with Delta-Oriented Slicing. In *FoVeOOS*, pages 61–75. Springer, 2011.

[8] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. *STTT*, 7(3):212–232, 2005.

[9] B. B. P. Cafeo, C. Hunsen, A. Garcia, S. Apel, and J. Lee. Segregating Feature Interfaces to Support Software Product Line Maintenance. In *Modularity*, pages 1–12. ACM, 2016.

[10] I. D. Carmo Machado, J. D. McGregor, Y. a. C. Cavalcanti, and E. S. De Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *IST*, 56(10):1183–1199, 2014.

[11] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software Dependencies, Work Dependencies, and Their Impact on Failures. *TSE*, 35(6):864–878, 2009.

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

[13] F. Damiani, O. Owe, J. Dovland, I. Schaefer, E. B. Johnsen, and I. C. Yu. A Transformational Proof System for Delta-Oriented Programming. In *FMSPLE*, pages 53–60. ACM, 2012.

[14] E. W. Dijkstra. Go To Statement Considered Harmful. *Comm. ACM*, 11(3):147–148, 1968.

[15] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. Do Background Colors Improve Program Comprehension in the #Ifdef Hell? *EMSE*, 18(4):699–745, 2013.

[16] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring Programming Experience. In *ICPC*, pages 73–82. IEEE, 2012.

[17] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *ICSE*, pages 261–270. ACM, 2008.

[18] C. Flanagan and K. R. M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *FME*, pages 500–517. Springer, 2001.

[19] R. Hähnle and I. Schaefer. A Liskov Principle for Delta-Oriented Programming. In *ISOLA*, pages 32–46. Springer, 2012.

[20] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral Interface Specification Languages. *CSUR*, 44(3):16:1–16:58, 2012.

[21] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer, 2008.

[22] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *ICSE*, pages 311–320. ACM, 2008.

[23] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *TOSEM*, 21(3):14:1–14:39, 2012.

[24] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *OOPSLA*, pages 805–824. ACM, 2011.

[25] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *OOPSLA*, pages 773–792. ACM, 2012.

[26] D. Le, E. Walkingshaw, and M. Erwig. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *VL/HCC*, pages 143–150. IEEE, 2011.

[27] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SEN*, 31(3):1–38, 2006.

[28] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *ICSE*, pages 105–114. IEEE, 2010.

[29] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-Aware Refactoring in the Wild. In *ICSE*, pages 380–391. IEEE, 2015.

[30] B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *TOPLAS*, 16(6):1811–1841, 1994.

[31] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *ICSE*, 2016. To appear.

[32] F. Medeiros, M. Ribeiro, R. Gheyi, and B. Fonseca. A Catalogue of Refactorings to Remove Incomplete Annotations. *J.UCS*, 20(5):746–771, 2014.

[33] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1st edition, 1988.

[34] T. Molderez and D. Janssens. Modular Reasoning in Aspect-Oriented Languages from a Substitution Perspective. *TAOSD*, pages 3–59, 2015.

[35] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2010.

[36] D. L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Comm. ACM*, 15(12):1053–1058, 1972.

[37] H. Rebêlo, G. T. Leavens, R. Lima, P. Borba, and M. Ribeiro. Modular Aspect-Oriented Design Rule Enforcement with XPIDRs. In *FOAL*, pages 13–18. ACM, 2013.

[38] H. Rebêlo, R. Lima, M. Cornélio, G. T. Leavens, A. Mota, and C. Oliveira. Optimizing JML Feature Compilation in Ajmlc Using Aspect-Oriented Refactorings. In *SBLP*, 2009.

[39] M. Ribeiro, P. Borba, and C. Kästner. Feature Maintenance with Emergent Interfaces. In *ICSE*, pages 989–1000. ACM, 2014.

[40] M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba. Emergent Feature Modularization. In *SPLASH*, pages 11–18. ACM, 2010.

[41] M. Ribeiro, F. Queiroz, P. Borba, T. Tolêdo, C. Brabrand, and S. Soares. On the Impact of Feature Dependencies when Maintaining Preprocessor-Based Software Product Lines. In *GPCE*, pages 23–32. ACM, 2011.

[42] R. Schröter, N. Siegmund, T. Thüm, and G. Saake. Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines. In *SPLC*, pages 102–111. ACM, 2014.

[43] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. Variant-Preserving Refactoring in Feature-Oriented Software Product Lines. In *VaMoS*, pages 73–81. ACM, 2012.

[44] T. Thüm. *Product-Line Specification and Verification with Feature-Oriented Contracts*. PhD thesis, University of Magdeburg, Germany, 2015.

[45] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *CSUR*, 47(1):6:1–6:45, 2014.

[46] T. Thüm, S. Apel, A. Zelend, R. Schröter, and B. Möller. Subclack: Feature-Oriented Programming with Behavioral Feature Interfaces. In *MASPEGHI*, pages 1–8. ACM, 2013.

[47] T. Thüm, T. Winkelmann, R. Schröter, M. Hentschel, and S. Krüger. Variability Hiding in Contracts for Dependent Software Product Lines. In *VaMoS*, pages 97–104. ACM, 2016.