Towards Assessing the Complexity of Object Migration in Dynamic, Feature-oriented Software Product Lines

(Note the corrigenda to Sections 4.3 and 7 as well as to Figure 4 marked by change bars, contributed in August 2014. Added text is underlined, deleted text is striked out. See Footnote 3 for an explanation.)

Stephan Adelsberger, Stefan Sobernig, Gustaf Neumann
Institute for Information Systems and New Media
WU Vienna, Austria
{stephan.adelsberger|stefan.sobernig|gustaf.neumann}@wu.ac.at

ABSTRACT

Dynamic Software Product Lines (DSPLs) implement features of a product family, from which products can be derived and reconfigured at runtime. This way, systems can alternate their configurations without service interruption. The activation and deactivation of features at runtime pose challenges for the implementation of a DSPL, in particular for handling object states such as runtime changes to object-scoped variables, their value assignments, and the variable properties. To quantify the complexity of this *object migration*, we propose a systematic code-level measurement approach which harvests feature implementations and the corresponding variability models for code introductions responsible for critical changes to object states.

We have applied our measurement process tentatively to data sets representing 9 SPLs implemented using Fuji. This way, we arrived at first insights on object-migration complexity in SPLs. For example, we observed that the number of feature-specific object states is distributed very unequally in Fuji SPLs, with a few objects having an overly complex map of potential object states and the majority of objects potentially seeing transitions between 1 and 5 object states. We also evaluated different tactics of applying SAT solvers to analyze variability models in this context.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Complexity measures*; D.1.5 [**Software**]: Programming Techniques—*Object-oriented Programming*

General Terms

Measurement, Languages

Keywords

Dynamic software product line, feature-oriented programming, feature binding, object migration, constructor anomaly

1. INTRODUCTION

A software product line (SPL) provides a common code base for a family of related software products; a variability model (e.g., a feature model) specifies the set of valid products that can be built from the product line. Including a feature into a program is referred to as binding a feature. Feature binding can occur at several binding times (preprocessing time, compile time, load time, program execution time), and in certain binding modes (i.e., fixed, changeable, or dynamic; [11, 26]). In contrast to static SPLs, where variability is commonly bound before runtime, dynamic software product lines (DSPLs) allow features to be bound and to be unbound at program-execution time [14, 18].

In feature-oriented programming (FOP), features are implemented as distinct and composable units of functionality referred to as feature modules [3]. FOP environments often extend an object-oriented base language by abstractions suitable for representing features, in particular, object collaborations, roles, and class refinements. A product derived from a product line, which is implemented using FOP, is realized as a composed set of collaborating objects (*collaboration*). The behavior defined by an object, i.e., the set of methods owned by or invocable on an object, can reflect different *roles* [3] played by an object in different feature modules. In FOP, role-implementing method sets specific to a given feature are commonly organized into dedicated composition units, i.e., class *refinements*.

Under dynamic feature binding, an object can take or discard roles by receiving or by dismissing the corresponding refinements; that is, it *migrates* from one behavioral configuration to another. Changing the class of an (already created) instance (*dynamic object re-classification*) is an established example of object migration in dynamic object-oriented programming (e.g., [13, 30, 12]). In a feature-oriented product-line implementation, dynamic feature binding raises questions over the way existing objects should be handled when a product changes during runtime. Lacking other options, existing objects might simply remain unchanged and a new, decoupled product must be derived. An interesting alternative, however, is to migrate existing object states. In this case, the programmer realizing a feature implementation must anticipate object migration to a certain extent.

The attempt to anticipate this migration, however, turns out to be tricky: Under static feature binding, there is often an implicit order in which feature implementations are meant to be composed. Note that this order is not necessarily made explicit in the variability model or the feature implementation. Often, the composition order is implicitly established by coding convention or the build environment. Under dynamic feature binding – and provided that there is no order-enforcing mechanism and the composition operation is commutative – a product (e.g., from a Graph Product Line; GPL [20]) containing 2 features (e.g., `Colored`, `Weighted`) can be derived in 2 different ways: by binding feature `Colored` first and `Weighted` second, or vice versa. Both ways yield one and the same feature configuration.

Features are typically implemented having a feature-local focus. In particular, the way objects as class instances are initialized using constructor methods (which, e.g., assign an initial or default value to fields) considers initial object states for the scope of only one feature implementation. For example, programmers limit constructor methods introduced by one feature implementation to touch on fields introduced by this feature only. This improves the cohesiveness of a feature implementation [1]. However, under dynamic feature binding, decisions on state initialization taken by programmers of other feature implementations, which become bound or unbound before a given feature, might interfere. This way, constraints imposed on an object's state by other programmers or other feature implementations of the same programmer may end up conflicting at the time of binding or unbinding. The situation becomes even more tedious when, in a dedicated feature-oriented language kit (FeatureC++ [25], ObjectTeams [17]), special-purpose initialization methods (e.g., *lifters* in ObjectTeams) are executed at the binding time of feature complement constructor methods. In addition, the orthogonal object-oriented inheritance hierarchies must be kept in mind when it comes to anticipating the linearization order of constructor methods.

An interesting question arises: How many object-state transitions (in terms of bindable and unbindable features) must be anticipated by the programmer of a given feature implementation in existing product-line implementations? To shed light on this, we propose a measurement approach to analyze object-state migration paths in the code bases of feature-oriented product lines. We explored the procedure using nine different product lines from the Fuji repository[1] [2]. Based on these insights, in future work, we hope to devise appropriate Design-by-Contract techniques [28] and variants of FOP-specific access modifiers [2] to help FOP programmers in coping with object-migration complexity. By presenting and reflecting on our ongoing research project in terms of this research-in-progress report, we hope to receive valuable feedback from the reviewers and workshop participants already in this early stage.

In Section 2, we explain the background of our research-in-progress by eliciting the notions of object migration and constructor anomalies, among others. To better communicate the motivation of our research to the reader, we then provide an example of constructor anomalies under object migration in Section 3. The actual analysis and measurement approach is then introduced and critically reviewed upon in Section 4. Section 5 highlights our next steps and, in Section 6, we briefly iterate over closely related work. The paper is summarized in Section 7.

2. BACKGROUND

In this research project, we assume some background knowledge on object migration and constructor anomalies, formal approaches to analyzing variability models (i.e., SAT solving), and dynamic FOP techniques (i.e., ObjectTeams [17]).

Object migration. A method implementation as part of an object's or a role's behavior definition operates typically under assumptions on the *object state*, i.e, the object-scoped variables, their value assignments (at a given time), and the variable properties (e.g., multiplicity and data-type constraints). Object migration in FOP for DSPLs, as the process of moving a live object from one behavioral configuration to another, challenges such fundamental assumptions:

- *Combinatorial complexity*: The number of possible, valid feature combinations may grow exponentially leading to hundreds of possible state configurations, even for medium sized examples like the Graph Product Line (GPL; [20]).
- *Symmetry of binding/unbinding*: For feature bindings to be fully dynamic, the binding operation must be reversible [14]. This is also necessary to form valid products during runtime when facing mutually exclusive features. As a result, object migration must support both initializing and de-initializing object states according to a given feature configuration.
- *Non-monotonicity*: Refinements that change the type of an object variable require migrating the currently assigned value. Only monotonic changes that widen the valid value-range of object variables are without issues.
- *Constructor anomalies*: In mainstream object-oriented languages such as Java and C#, but also others (e.g., Eiffel), the construction of fresh instances from a given class can result in exceptional, unrecoverable program states [10]. For example, during the execution of a constructor method, fields of non-null types may contain null values and thereby break code that assumes non-null values, e.g., yielding null pointer exceptions in Java. Such anomalies can be due to constructor methods often being statically bound, so constructors are dispatched on not fully initialized objects. In addition, a constructor may call methods that are dynamically bound, i.e., the specific method implementation called is unknown at compile time (at least in Java and `C#`). In addition, a constructor must implicitly or explicitly call the super constructor at the appropriate location.

Analyzing feature models using SAT solvers. In this research project, we consider feature models [11] as a specific kind of variability modeling technique. Feature models capture hierarchies of features, including their inclusion constraints (e.g., mandatory, optional) and their hierarchical variation relationships (e.g., exclusive-or and inclusive-or feature groups). In addition, non-hierarchical dependency relationships (so-called *cross-tree constraints*) can be expressed. For automatic analysis, a feature model can be expressed as a propositional formula [21]. The respective features are represented as logic variables with `true` meaning that a feature is selected. Satisfiability (SAT) solvers can be used to check whether the propositional formula has an interpretation and to find all interpretations representing all product variants which honor the constraints of the feature model. The propositional formula is typically turned into its Conjunctive Normal Form (CNF)

[1]`http://fosd.net/fuji`, last accessed: 12.11.2013.

to allow efficient processing by SAT solvers. Performance is critical, as the possible number of product variants in the worst case is 2^n, where n is the number of (purely optional) features. In our ongoing research, we employ SAT solvers to compute the sets of valid feature configurations from a given variability model.

ObjectTeams. ObjectTeams [17] is a Java extension that supports aspects and collaboration-based designs which are dynamically composable via delegation layers [23]. Classes and collaborations (called `teams`) can be nested. This way, ObjectTeams realizes the FOP concept of *roles* using classes nested in teams. Role-representing nested classes are specified using the `playedBy` keyword. Roles as well as base classes can act as teams. A role may explicitly set a superclass. In addition, a role can implicitly extend the role class that shares the same name of the enclosing class' super collaboration. The constructor of a role may call up to 3 different constructors: `super()` calls the superclass constructor, `base()` calls the base constructor of a bound role, and `tsuper()` calls the implicit constructor. All 3 constructor calls are optional, as long as the suitable base constructor is implicitly invoked. Furthermore, the order, in which the constructors are to be called, is not restricted.

3. A MOTIVATING CONSTRUCTOR ANOMALY IN FOP

A critical part of object migration is the correct initialization of the object state. In a dynamic setting, refinements may add an object variable of a specified type to a class (as in object re-classification in a non-FOP context). This object variable needs to be instantiated. The state initialization involves pitfalls known as constructor anomalies.

The example below shows a problem related to constructor anomalies in plain Java. Consider the execution of the constructor `G()`. The super-class constructor `F()` is called, then the dynamically bound method `G.m()` is executed, which in turn accesses the uninitialized state, i.e., the field `value` is not properly initialized yet. As a consequence, a `NullPointerException` is raised.

```
public class F {
    public F() { m(); }
    public void m() { }
}
public class G extends F {
    private Object value = null;
    public G() { super(); value = new Object(); }
    public void m() { System.out.print(value.toString()); }
}
```

Dynamic collaborations are one of the approaches to FOP of DPLs [26], and specifically interesting for object migration. A common Java extension for dynamic collaborations is ObjectTeams (OT; [17]). As mentioned in the section "Background", a role class in OT can call 3 different constructors in any particular order and constructor calls may be omitted (as long as `base` is called explicitly or implicitly via superclass constructors). The following example shows a constructor anomaly problem in OT:

```
1  public team class BaseCollaboration {
2     public team class Base {
3        protected Object baseField;
4        public void Base() { }
5     }
6  }
7  public team class Outer {
8     public team class Super {
9        protected class Role {    // implicit super role
10          protected Object superTeamField;
11          public Role() { superTeamField = new Object(); }
12       }
13    }
14
15    public team class Collaboration extends Super playedBy
         BaseCollaboration {
16       protected class SuperRole { // explicit super role
17          protected Object superField;
18          public SuperRole() { superField = new Object(); }
19       }
20
21       protected class Role extends SuperRole
22                      playedBy Base<@base> {
23          public Role() {
24             // choice of three possible constructors to call:
25             // base(), super() or tsuper()
26             base(); // calls BaseCollaboration.Base.Base()
27             superField.toString(); // exception!
28             superTeamField.toString(); // exception!
29 } } } }
```

Consider lines 23–28 in the above example, in which the constructor implementation may access the state (i.e., object variables) of all other classes, including team classes and also the field `baseField` (the latter via callin mechanism). Furthermore, all 4 constructors can call methods that may be dynamically bound to methods of the class role defined in line 21. Finally, the constructor defined in the lines 23–28 may explicitly call all other 3 constructors: `base()` calls `BaseCollaboration.Base`, `tsuper` would call `Super.Role.Role()` and super would call `Collaboration.SuperRole.SuperRole()`. However, only the `base()` constructor call is mandatory. Consequently, accessing any of the uninitialized fields (lines 27 and 28) leads to a `NullPointerException`.

Compared to the plain Java example, the constructor anomalies emerge by merely accessing object variables. It becomes clear that the 2 additional types of constructors (`base` and `tsuper`) add new dimensions to the constructor-anomaly problem. An extended example would include dynamic method calls from within the above constructors and, furthermore, `Base` itself may have a super class that in turn has bound roles. We argue that the implicit constraints may quickly surpass what can be anticipated manually by a programmer. The resulting complexity calls for programmer assistance to overview state initialization and object state migration in a FOP-based DSPL implementation.

4. COMPLEXITY OF OBJECT MIGRATION PATHS

The aim of our research is to analyze the variability models of SPLs in conjunction with code-level measurements (i.e., code units introduced by feature modules) to assess the complexity of object migration under dynamic feature binding. The established degree of complexity should, for example, guide us in assessing the risk level of experiencing unwanted constructor anomalies. In the following, we give a brief synopsis of the process to analyze the complexity of object migration in DSPLs.

4.1 Measurement Process

The chosen measurement approach is based on the object migration graphs for all objects of a product, when the set of activated features changes. For every role of every collaboration an object migration graph is calculated. It is not necessary to calculate a set of such graphs for every valid configuration (feature selection), but only for these feature selections that contain for a particular role implementation *state-affecting refinements* (refinements that can change the object state, e.g., by setting field values). The state-affecting refinements are identified by analyzing the feature specific constructor refinements of a role. Only the state-affecting features are required for building the object migration graph for this role. The nodes of the object migration are labeled by the elements of the power set of the state-affecting features. Assuming that transitions between all feature sections are possible (e.g., multiple features can be activated at once), the result is a complete graph where all nodes are connected. Note that this assumption can be altered to include a particular (often implicit) order of composing features.

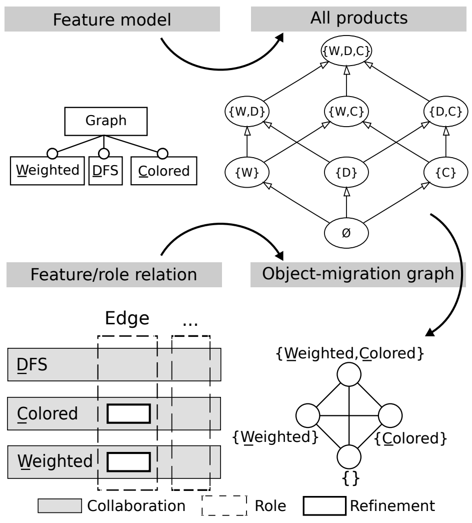

Figure 1: Calculating object-migration paths using an excerpt from the Graph Product Line (GPL)

Consider the minimal example depicted in Figure 1, a small excerpt from the Graph Product Line (GPL; [20]). The GPL models variability for different types of graphs (colored, weighted, etc.). The Figure depicts in the top row the feature model and valid configurations (features abbreviated) and in the lower row the implementation aspects for a role and the entailed object migration path. For example, the feature `Colored` allows to attach a value representing edge colors, while `DFS` (Depth First Search) is a feature that implements the basic graph search algorithm. The implementation of the GPL includes the role `Edge` that implements the functionality to store edges of a graph. The state-affecting features for the role `Edge` are `Colored` and `Weighted`, but not `DFS` because `DFS` does not require any state-affecting refinement of role `Edge` (see lower lower left graphic of Figure 1). The nodes of the object migration path are built from the elements of the power-set of the state-affecting features for this role. Provided that all object migrations are permitted, this leads to the migration paths depicted as a complete graph (lower right graphic in Figure 1).

The subsequent activities of our measurement process are depicted in Figure 2. Based on code-level measurements of an SPL implementation, data about all refinements, including their type (e.g., constructor refinement) is gathered for all roles, in addition to information about accessed variables. This data is queried for state-affecting constructor refinements. A relation between features and roles is recorded by the result of the query (depicted as *feature/role relation* in Figure 2). A feature and a role are related if the feature introduces a state-affecting constructor refinement. The second input to the process is the feature model of an SPL, which is exported to a propositional formula in CNF. Currently, we employ 2 alternative SAT-based tactics. As SAT platform, we have decided to use picosat [6], a reportedly fast single-threaded SAT solver.

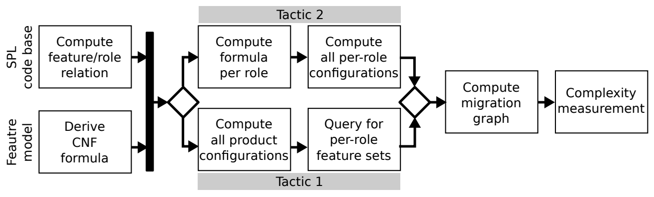
Tactic 2
SPL code base
Compute feature/role relation
Compute formula per role
Compute all per-role configurations
Compute migration graph
Complexity measurement
Feautre model
Derive CNF formula
Compute all product configurations
Query for per-role feature sets
Tactic 1

Figure 2: Overview of the measurement procedure

Tactic 1. All valid product configurations (i.e., all introductions) are computed via a SAT solver once and are then queried for partial feature combinations which are relevant for each role, based on the feature/role relation.

Tactic 2. An alternative tactic involves role-wise SAT operations. For each role, there are 2^r possible combinations of refinements, with `r` being the number of state-affecting constructor refinements. For each of these role-specific combinations a per-role CNF formula is computed and then checked with a SAT solver whether it is valid (i.e., whether there is at least one introduction).

Tactic 2 allows for scaling our measurement approch to SPLs which are large in features. SAT-solver calls per role can be executed in parallel. Assuming the same performance results as described by Mendonca et al. [21] as an approximation, an individual SAT call can be estimated in the order of 100 milliseconds (ms) for feature models of about 1,000 nodes and more. For 16 refinements per role (2^{16} combinations), for instance, approximately one second would be required considering 100ms per SAT call using 8 parallel SAT processes.

In our initial analysis (see Section 4.3), we adopt both tactics. A tactic is selected on a per-SPL basis by choosing the calculation path (i.e., all products or per-role) according to the observed feature-model complexity (e.g., the number of optional features, number of state-affecting constructor refinements). In the final 2 steps, a graph of the possible state-migration paths of a role is constructed from the resulting data of the SAT analysis (using either tactic 1 or 2). To quantify the complexity of this fully connected graph, we currently count the number of nodes. Using this quantification of complexity for each role of an SPL, it is possible to analyze the SPL with descriptive statistics, which in turn allows for comparing 2 or more SPLs. We address the limitations of the described measurement setup in Section 4.4.

4.2 Data Sets

Table 1 summarizes important characteristics of the currently analyzed Fuji data sets [2] and illustrates the difference in size of the respective feature models. `SLOC` refers to the source lines of code of the SPL implementation, `roles` is the number of roles, `refs` is the total number of refinements, `cf` refers to the number of concrete features of the respective feature models and `f` stands for the total number of

	SLOC	#roles	#refs	#tactic	#cf	#f
BerkeleyDB	45 000	408	620	1	99	114
EPL	111	5	15	2	12	16
GameOfLife	1 461	37	39	2	15	23
GPL	1 930	16	57	2	25	36
MobileMedia8	4 189	60	127	2	47	54
PKJab	3 373	51	68	2	8	12
Prevayler	5 268	158	149	2	6	8
TankWar	4 845	22	88	2	37	37
Violet	7 151	67	157	1	83	96

Table 1: Overview of the data sets. refs: refinements; cf: concrete features; f: total features

features (abstract and concrete). *Abstract features* are part of the feature model, but do not have any impact on the implementation level, unlike concrete features [27]. *Concrete features* are implemented as dedicated feature modules and are usually depicted as leafs in the feature diagram [27].

Featurewise, the data sets results from SPLs of various sizes (e.g., TankWar having 37 features). The two largest data sets, both in terms of features and SLOC, are BerkeleyDB and Violet. In the SAT-analysis step, tactic 1 has been applied to the 7 SPLs of low- to medium feature-model complexity (see column `tactic` of Table 1). Tactic 2 has been used to analyze BerkeleyDB and Violet, the 2 most complex SPLs.

4.3 Preliminary Observations

We started by extracting data (variability models, code introductions by features) from the 28 feature-oriented product lines available in the Fuji repository [2]. Nine out of these 28 SPLs provide an explicit variability model (see also Table 1). In the following, we refer to 3 notable examples of Fuji SPLs: GPL, Violet, and BerkeleyDB. [2]

Refinements per role. Recall from Figure 1 that at the beginning the relation between features and roles is established from analyzing code introductions and structural references in the SPL code bases. More specifically, we are interested in state-affecting constructor refinements. Therefore, we query refinements that initialize object state (e.g., object variables) in a constructor. In terms of operationalization, we queried for refinements which introduce constructor methods that access and initialize at least one field.

Figure 3 summarizes the refinements per role for the 3 SPLs. The top plot shows statistics on all refinements per role, while the bottom plot shows the corresponding statistics on the subset of state-affecting constructor refinements. We use box-plots to visualize the distribution structure of the data sets, including dispersion and skewness. Outliers in the data are displayed as individual points. The boxes depict the first, second (i.e., median) and third quartile. A line (whisker) is drawn between the top end of the box and the last value that is within the third quartile plus the interquartile range (IQR). If all statistics have comparable values, the box is degenerated to a single, thick line.

For BerkeleyDB in general, we find many refinements per role. As for state-affecting constructor refinements, the maximum is an outlier at only 4 refinements. Violet shows much

[2]Note that for BerkeleyDB and Violet, we were not able to compute all possible product configurations (see Section 4.4).

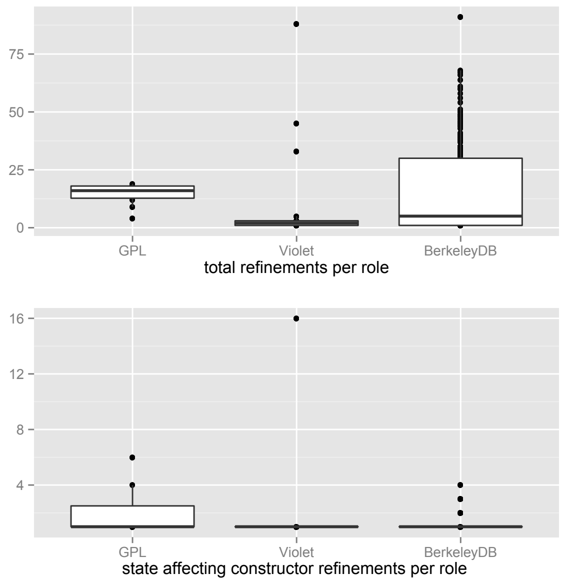

Figure 3: Boxplots for the observed refinements per role

fewer refinements per role, but also shows a high outlier in the bottom box-plot that represents 16 constructor refinements (!) within just one role. The topmost outlier for the GPL example are six different constructor refinements within one role. For all 3 SPLs, the observation is that the majorities of roles have a comparably low number of state-affecting constructor refinements (1-2 refinements per role).

Size of the migration graph. To quantify possible object migrations, the migration graph for each role is summarized by the order of the graph (number of nodes). Each node represents a state-affecting feature selection for this role (i.e, in Figure 1 the state-affecting feature selections of the role `Edge`). When all features are optional, n state-affecting features require the object migration to be of an order of 2^n. If we allow migrations between all nodes, $2^n * (2^n - 1)$ migrations are possible for this role.

Figure 4 shows histogram and kernel density plots of this size measurement for the object migration graphs for each role in the 3 Fuji SPLs (i.e., the number of roles having a certain number of nodes in their object migration graphs).

For the BerkeleyDB, the topmost plot illustrates one role with 8 nodes. Furthermore, there are 2 roles having 4 nodes. However, the vast majority of roles have $0 - 2$ nodes in the object migration graph. Violet is notable (see the middle plot in Figure 4): The scale had to be changed to a logarithmic scale, because the order of the object migration graphs of ~~2 roles~~ a role was already quite high: The maximum outlier, role ~~"UMLEditor"~~ EditorFrame, has an object migration graph containing ~~530,000~~ 480 nodes ~~-a second outlier represents the role "EditorFrame", has a migration graph with 480 nodes~~.[3]

[3] Corrigendum: The published revision of this research-in-progress report referred to the value of 530,000 nodes (UMLEditor) as the maximum outlier. This number turned out incorrect, being reported due to a mistake in one of our data-processing scripts: Instead of counting constructor refinements only, refinements of ordinary methods were taken into account. The corrected observation of an outlier graph

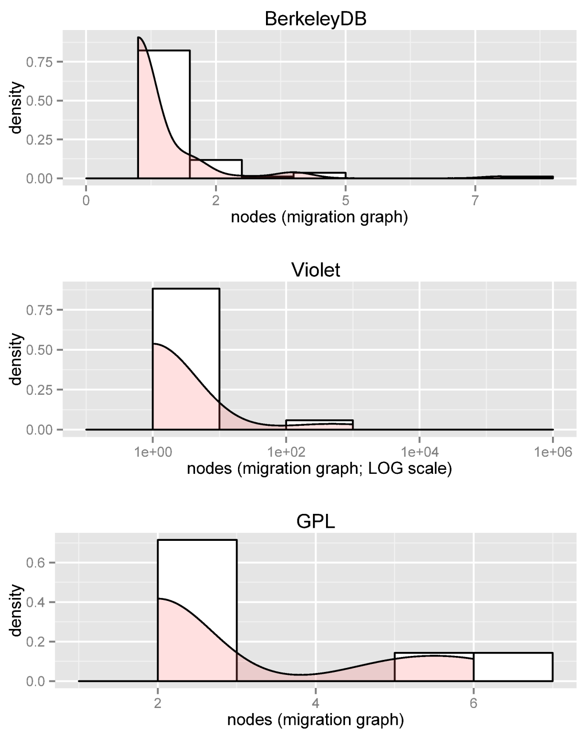

Figure 4: Histograms and kernel densities of the orders of the per-role migration graphs[3]

The bottom plot in Figure 4 visualizes the results for the GPL. While there is an outlier representing an order of 6, most roles have only small migration graphs (2-3 nodes).

Tentative discussion. It is noteworthy that, despite their similarity in terms of their feature sizes and the numbers of refinements per role, BerkeleyDB and Violet show a major discrepancy ~~in the order of magnitude and~~ in the number of nodes of the migration graphs, that is, ~~530,000~~ 480 versus 8 nodes as outliers.[3]

All 3 SPLs, despite their differences in SLOC and feature sizes, show similar distribution patterns of node sizes of object-migration graphs per role. All histograms showed migration graphs centered around 1 to 5 nodes with outliers having a comparatively high number of nodes. Therefore, all histograms show distributions skewed to the left.

BerkeleyDB showcases that even for a complex SPL (given that there are 114 features, 155 different roles with up to 93 refinements per role), the migration complexity can remain limited. The maximum number of state-affecting refinements amounts to just 4 refinements. Therefore, there are at most 2^4 nodes in the object migration graph of a role.

Likewise, the GPL example demonstrates that, despite 6 state-affecting constructor refinements per role, the resulting migration graph of a role has just 6 nodes. This is due to refinements belonging to features that exclude each other mutually (XOR). Thus, multiple state-affecting refinements may not always contribute to complex object states, especially not in the case of XOR-features.

with 480 nodes (EditorFrame) still supports our finding that the resulting object-state spaces risk becoming too complex for a programmer to keep track of.

4.4 Limitations and Threats to Validity

Our SAT-based approach shares threats that have already been reported by Mendonca et al. [21].

External validity. Most importantly, an external threat is the applicability of our approach to realistic feature models and SPLs. On the one hand, the Fuji data sets included only 2 SPLs that had more than 37 features. On the other hand, only smaller examples (e.g., TankWar having 37 features) can be processed efficiently and timely. Finding all valid products of the two largest SPL in terms of features (Violet and BerkeleyDB; see also Table 1) is challenging. Clearly, for variability models with 100 or more features, a different approach is needed to measure the complexity of object migration accurately and timely. As shown by Mendonca et al. [21], it is easy to generate variability models of 300 features and more that cannot be processed by SAT solvers in reasonable time, not even to check whether a single configuration is valid. Timeliness, however, is critical because running a SAT solver for days makes it impossible to develop adequate tool support for object migration.

Internal validity. The limited scalability of a SAT-based approach, as described above, is also an important internal threat. After having run the SAT solver for 3 days on the two largest SPLs in terms of features (Violet and BerkeleyDB; see also Table 1), we added a hard-wired propagation limit and worked with all the product variants that had been found to that time limit. At the point of interruption, about one million product configurations had been computed. To this end, our observations might be due to a product sample showing biased characteristics (e.g., actual number of nodes of the migration graph potentially being higher).

Another internal threat to validity is that our measurement data is gathered from static SPL examples. While Fuji-based product lines are static, this repository is the only one which provides the necessary data (feature-aware code bases, variability models) to run and to explore the suggested measurement process. This way, we have found evidence of structural complexities that can lead to constructor anomalies in the dataset that are valid for static and dynamic SPLs. Note that there are FOP approaches which allow for static and dynamic feature binding using identical product-line implementations such as FeatureC++ [25] and NX [22, 26], for which our actual measurement provides valid and interesting insights. Data from actual dynamic SPLs, however, may reveal additional and different problems, e.g., problems that are associated with the unbinding operation of features. As a next step, we will consider data sets extracted from actual DSPL code bases (e.g., ObjectTeams).

5. FUTURE WORK

We will extend our measurement approach, include further data sets, and tackle the SAT-related scalability issues.

Additional measurements: First, for future measurements, we intend to put forth a measurement plan to systematically link measurement objectives, research questions, and indicator measures. One option is to use the Goal-Question-Metric approach (GQM; [8]). Second, we will revise our measurement constructs. Currently, only constructors that explicitly change a single object variable have been investigated. A next step is to include constructors that affect object state implicitly, either via method calls that reference object variables or via superclass constructor calls that change the object state. Another option is to assign complexity weights to constructors, e.g., to distinguish that a constructor that references five different object variables is more complex than one that changes a single variable.

Analysis of additional data sets: We will apply our analysis to actual DSPL code bases. ObjectTeams supports dynamic role binding, in which a role object can be added to a base object at runtime. ObjectTeams does not support object migration or object re-classification, as defined in this paper; however, there exists a related concept called *lifter* methods to initialize the state of a newly bound role. There are real world programs implemented in ObjectTeams, notably the Eclipse plugin "Equinox"; the code is freely available. The only drawback is that these plug-ins do not make the implemented variability explicit. A possible solution is to derive a feature model, e.g., all plug-ins are optional features and plugin dependencies are modeled as cross-tree constraints. In addition, we also plan to contact the ObjectTeams community and ask for further SPL examples.

Strategies to resolve scalability issues: As pointed out in Section 4.4, the tactic of calculating all valid products (see Figure 2) does not scale to SPLs of approx. 100 or more features. Similarly, deriving per-role formulae is also limited by the combinatorial complexity of per-role feature combinations. For the Fuji data set, we observed up to six state-affecting refinements per role, with outliers of up to 16 refinements for Violet (see Figure 3).

Next, we will explore a third, hybrid tactic to calculate all valid product configurations with a very narrow propagation constraint first (i.e., a variant of tactic 1) and then process missing feature combinations per role each in a single SAT call (i.e., tactic 2). Additionally, we plan to investigate alternatives to using a SAT solver in our measurement process, e.g., using Constraint Satisfaction Problem (CSP) solvers and Binary Decision Diagrams (BDD; see [5, 4]).

6. RELATED WORK

Research closely related to our work falls into 3 categories: object migration, SAT-based analysis of variability models, and structural metrics in object-oriented systems.

Object Migration. The notion of object migration can be found in literature on distributed systems and distributed-object mobility and on dynamic object-oriented database and language systems. Our work adds to the state of the art in the latter field. In object-oriented databases, (see, e.g., [12]) object migration deals with dynamic properties, especially when changing an object's class. Without considering object persistence, this is also discussed as *dynamic object re-classification* in language-engineering communities (e.g., CLOS [19]).

First destroying an object as an instance of the old class and then recreating it as an object of the new class is often not feasible, especially when considering foreign-key constraints in database systems persisting the changing objects. In the following, object migration was first studied between immediate subclasses and superclasses. Later, object migration was studied in the context of roles [13, 30] as means of object evolution. While objects can play different roles over time, they typically allow only a limited or predefined set of role transitions (or, object-migration paths in our terminology). In contrast to this early line of research [12, 30], we study arbitrarily complex object-migration paths resulting from dynamic feature binding and unbinding in DSPLs. For optional features, the migration graph is complete.

Object migration shares challenges faced by other communities, in particular Context-Oriented Programming (COP; [9]) and Dynamic Updates [29, 24, 16, 31]. Wernli et al. [29] allow dynamic updates based on contexts in a multi-threaded setting by keeping 2 real copies of each object and providing bidirectional transformations between the 2 copies. Previtali and Gross [24] propose a method to perform dynamic software updates based on aspects and aim at an automatic update process, with newly added attributes still requiring manual glue code. Dynamic software updates have been proposed for static programming languages as well, including C++ [16]. Our work is specifically tailored towards object migration in the context of DSPLs, as realizable using dynamic FOP techniques such as ObjectTeams (OT; [17]). This adds further challenges to the issue of object migration, namely the combinatorial complexity in integrated variability analyses.

SAT-based analysis of variability models. Checking with a SAT solver whether a configuration is a valid product is efficient for typical SPL examples [21], but artificial feature models can be generated where the problem is NP complete [21, 4]. In addition, SAT solvers do not scale well on multi-core servers [15] and therefore, we decided to use picosat [6], a fast single-threaded SAT solver that can derive all models of a propositional formula.

The performance and scalability problems we experienced are due to the operation that is classified by Benavides et. al [5] as "all products", i.e., meaning to calculate all valid configurations of a feature model. Other operations, such as calculating the number of valid products are not sufficient for deriving the migration graph of a role. Besides SAT-based approaches to calculate all products, there are also approaches based on description logic [5] and constraint programming [5, 4].

Object-oriented software measurement. It is interesting to observe that in software measurement approaches for object-oriented systems, constructors are often not taken into account. They are rather excluded from measurement as data points, for example, when calculating cohesion measures [7]. The definition of the measures *tight class cohesion* (TCC) and *loose class cohesion* (LCC) requires the exclusion of constructors. As for cohesion, constructors are considered problematic because they typically access most of the class attributes, thereby linking the constructor with any method that accesses at least one of these attributes. This can result in reporting an overstated value of cohesion [7]. In our approach, constructors are key data points to identify critical refinements and to construct object-migration graphs.

7. SUMMARY & OUTLOOK

We set out to quantify the complexity of object migration in dynamic software product lines (DSPLs) implemented using feature-oriented programming (FOP) techniques. Our aim is to estimate the number and the complexity of migration steps and to characterize problems that, paired with complex object migrations, constitute a possible source of unanticipated, unrecoverable program states.

To this end, we have developed a measurement procedure that combines variability analysis using SAT solvers and code-level measurements. We have applied the process to an initial selection of SPLs, available from the Fuji repository [2]. Our preliminary findings have illustrated circumstances that lead to complex object states. For Violet, for instance, multiple optional features refine the same role causing changes to the constructor by initializing object states. The optional features of Violet lead to a combinatorial explosion in the number of possible object migrations. In one case, the resulting object-migration graph indicates ~~530,000~~ 480 different states of a single object.[3] This complexity introduces risks, especially, when a programmer is required to manually implement functionality to handle transitions between object states (e.g., *lifter* methods in ObjectTeams).

We argue that for state-affecting features to be fully dynamic, as described by Hallsteinsen et al. [14], the problem of constructor anomalies under object migration must be addressed. Furthermore, object-migration graphs can be used to identify situations in which object migration is impossible and should be prevented since the resulting constructor anomalies would cause runtime exceptions. Migration graphs can be used to generate tooltips in an IDE to highlight colliding constructors or to warn a programmer about otherwise implicit state constraints.

Acknowledgements

Special thanks are due to our colleagues providing and maintaining the Fuji SPL repository, namely Sergiy Kolesnikov and Sven Apel. We also would like to thank Stephan Herrmann for answering our technical questions about OT via private communication. Finally, we thank the three anonymous VAMOS reviewers for the valuable feedback that entered our manuscript revision.

8. REFERENCES

[1] S. Apel and D. Beyer. Feature cohesion in software product lines: An exploratory study. In *Proc. ICSE'11*, pages 421–430. ACM, 2011.

[2] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich. Access control in feature-oriented programming. *Sci. Comput. Program.*, 77(3):174–187, 2012.

[3] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE T. Software Eng.*, 34(2):162–180, 2008.

[4] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49(12):45–47, 2006.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inform. Syst.*, 35(6):615–636, 2010.

[6] A. Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[7] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empir. Softw. Eng.*, 3(1):65–117, 1998.

[8] V. R. B. G. Caldiera and H. D. Rombach. The goal question metric approach. *Encyclopedia of software engineering*, 2(1994):528–532, 1994.

[9] N. Cardozo, S. González, K. Mens, and T. D'Hondt. Safer context (de)activation: through the prompt-loyal strategy. In *Proc. of the 3rd Int'l Workshop Context-Oriented Programming*, COP '11, pages 2:1–2:6, New York, NY, USA, 2011. ACM.

[10] T. Cohen and J. Gil. Better construction with factories. *JOT*, 6(6):103–123, 2007.

[11] K. Czarnecki and U. W. Eisenecker. *Generative Programming — Methods, Tools, and Applications.* Addison-Wesley, 6th edition, 2000.

[12] M. E. El-Sharkawi and Y. Kambayashi. Object migration mechanisms to support updates in object-oriented databases. In *Databases, Parallel Architectures and Their Applications,. PARBASE-90, Int'l Conf.*, pages 378–387. IEEE, 1990.

[13] G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. *ACM Trans. Inf. Syst.*, 14(3):268–296, 1996.

[14] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *IEEE Computer*, 41(4):93–95, 2008.

[15] Y. Hamadi, S. Jabbour, and L. Sais. Manysat: a parallel sat solver. *JSAT*, 6(4):245–262, 2009.

[16] C. Hayden, E. Smith, M. Hicks, and J. Foster. State transfer for clear and efficient runtime updates. In *Data Eng. Workshops (ICDEW), 2011 IEEE 27th Int'l Conf.*, pages 179–184. IEEE, 2011.

[17] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. *Lect. Notes Comput. Sc.*, pages 248–264, 2003.

[18] M. Hinchey, S. Park, and K. Schmid. Building dynamic software product lines. *IEEE Computer*, 45(10):22–26, 2012.

[19] G. Kiczales, J. Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[20] R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Generative and Component-Based Software Engineering*, pages 10–24. Springer, 2001.

[21] M. Mendonca, A. Wąsowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *Proc. of the 13th Int'l Software Product Line Conf.*, pages 231–240. Carnegie Mellon University, 2009.

[22] G. Neumann and S. Sobernig. An overview of the next scripting toolkit. In *Proc. of the 18th Annual Tcl/Tk Conference*, Manassas, Virginia, USA, November 2011.

[23] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proc. 16th Europ. Conf. Object-Oriented Programming (ECOOP'02)*, pages 89–110. Springer, 2002.

[24] S. Previtali and T. Gross. Dynamic updating of software systems based on aspects. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE Int'l Conf.*, pages 83–92. IEEE, 2006.

[25] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. Flexible feature binding in software product lines. *Autom. Softw. Eng.*, 18:163–197, 2011.

[26] S. Sobernig, G. Neumann, and S. Adelsberger. Supporting multiple feature binding strategies in nx. In *Proc. of the 4th Int'l Workshop on Feature-Oriented Software Development*, pages 45–53. ACM, 2012.

[27] T. Thum, C. Kastner, S. Erdweg, and N. Siegmund. Abstract features in feature modeling. In *Software Product Line Conf. (SPLC), 2011 15th Int'l*, pages 191–200. IEEE, 2011.

[28] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. Applying design by contract to feature-oriented programming. In *Lect. Notes Comput.*

Sc., pages 255–269. Springer, 2012.

[29] E. Wernli, M. Lungu, and O. Nierstrasz. Incremental dynamic updates with first-class contexts. *Objects, Models, Components, Patterns*, pages 304–319, 2012.

[30] R. K. Wong, H. L. Chau, and F. H. Lochovsky. A data model and semantics of objects with dynamic roles. In *Data Eng., 1997. Proc. 13th Int'l Conf.*, pages 402–411. IEEE, 1997.

[31] T. Würthinger, C. Wimmer, and L. Stadler. Unrestricted and safe dynamic code evolution for Java. *Sci. Comput. Program*, 2011.