From feature models to feature toggles in practice

Jean-Marc Jézéquel, Jörg Kienzle, Mathieu Acher

From Feature Models to Feature Toggles in Practice

Jean-Marc Jézéquel
Univ Rennes, CNRS, Inria, IRISA
Rennes, France
jezequel@irisa.fr

Jörg Kienzle
McGill University
Montreal, Canada
Joerg.Kienzle@mcgill.ca

Mathieu Acher
Univ Rennes, IUF, CNRS, Inria, IRISA
Rennes, France
mathieu.acher@irisa.fr

ABSTRACT

Feature Toggles (often also referred to as Feature Flags) are a powerful technique, providing an alternative to maintaining multiple feature branches in source code. A condition within the code enables or disables a feature at runtime, hence providing a kind of runtime variability resolution. Several works have already identified the proximity of this concept with the notion of *Feature* found in *Software Product Lines*. In this paper, we propose to go one step further in unifying these concepts to provide a seamless transition between design time and runtime variability resolutions. We propose to model all the variability using a feature model. Then this feature model can be partially resolved at design time (yielding an incomplete product derivation), the unresolved variability being used to generate feature toggles that can be enabled/disabled at runtime. We first demonstrate these ideas on the toy example of the *Expression Product Line*, and then show how it can scale to build a configurable authentication system, where a partially resolved feature model can interface with popular feature toggle frameworks such as *Togglz*.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

Configuration, Feature toggles and flags, Binding times, Variability

ACM Reference Format:
Jean-Marc Jézéquel, Jörg Kienzle, and Mathieu Acher. 2022. From Feature Models to Feature Toggles in Practice. In *26th ACM International Systems and Software Product Line Conference - Volume A (SPLC '22), September 12–16, 2022, Graz, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3546932.3547009

1 INTRODUCTION

Feature Toggles, often also referred to as Feature Flags, are providing an alternative to maintaining multiple feature branches in source code when following trunk-based development [39]. A condition within the code enables or disables a feature at runtime, hence providing a kind of runtime variability resolution.

Feature toggles are becoming popular for many kinds of applications, since they support many use cases in a unified way:

Privileged Access is used to deploy a feature in beta mode on a live application by choosing who can see it (*e.g.,* Newbie vs. Power User), or offering an opt-in option. That can also be used to allow Incremental Roll Outs.

A/B testing and Kill Switch Run A/B tests of features to see which features perform better, or kill a feature that performs badly.

Block Users to protect features from users by excluding them from ever seeing them, *e.g.,* to deal with regulations that differ by state or by country.

Calendar Driven Launches or Retirement so that a feature appears or disappears on a specific date.

Adaptation to System Load Disabling resource-intensive features when the system is under heavy load.

Several works have already identified the synergy of this concept with the notion of *Feature* found in Software Product Lines [26, 32]. In particular, Meinicke et al. [26] found that while feature toggles are usually meant to be temporary and developers often intend to remove them, they rarely do so unless forced by policy or technical steps. Removal can indeed be difficult and cumbersome. The authors also found that feature interactions are not really dealt with beyond the rule of thumb to try to keep it simple. However, both issues have been extensively addressed in the SPL community.

In this paper, we propose to provide a seamless transition between design time and runtime variability resolutions for mainstream programming platforms such as Java or C++, without resorting to exotic programming level concepts. Since most of the software industry is quite conservative with respect to exoticism, in particular those which already make use of *Feature Toggles* in their products, we put ourselves under the constraint of only using well-established design level formalism and mainstream programming level constructs. We thus propose to model all the variability using a feature model. Then this feature model can be partially resolved at design time (yielding an incomplete product derivation), the unresolved variability being used to generate feature toggles that can be enabled/disabled at runtime. We demonstrate these ideas by building an *Authentication* system, where we use Concern-Oriented Reuse (CORE) [1] – a model-driven approach that supports feature-oriented modularization. We show how to partially resolve design-time variability, and then interface with popular feature toggle frameworks, such as *Togglz* [25], to provide runtime variability.

Remainder. Section 2 discusses the motivation of this paper; Section 3 presents several possible implementations of the approach for a toy SPL, targeting a typical OO language such as Java; Section 4 analyses the pro and cons of these implementations and evaluates whether these approaches scale for a real application; Section 5 describes related work and Section 6 concludes our paper.

SPLC '22, September 12–16, 2022, Graz, Austria

ACM ISBN 978-1-4503-9443-7/22/09…$15.00
https://doi.org/10.1145/3546932.3547009

2 BACKGROUND AND MOTIVATION

As a motivating example, let us consider a configurable *Authentication Library* that might be reused for a large set of applications. It can use several types of credentials (password, biometrics, ...), with or without expiry, and even two-factor authentication.

Different systems have different authentication needs, and not every system needs all the features that the *Authentication Library* offers, hence the need for both design time and runtime feature selections as illustrated in Figure 1.

Clearly, all the practical use cases listed above for using feature toggles are relevant for this *Authentication Library*: Privilege Access, Incremental Roll Outs, Block Users, A/B testing and Kill Switch, Calendar Driven Launches or retirement. Beyond these runtime choices, several operations could also be made at design time using a typical SPL approach:

Feature removal Keeping code for unused feature toggles may cause problems which can have a severe impact, such as code complexity, dead code, and system failure. For example, the erroneous repurposing of an old feature toggle caused Knight Capital Group, an American global financial services firm, to go bankrupt due to the implications of the resultant incorrect system behavior [23]. It can thus be extremely important to remove no longer useful features, or to build specific deployments for specific regulations (e.g., when strong cryptography is forbidden to be exported to some countries), or even for cybersecurity purposes, such as, reducing the attack surface of the software. However, removing all the code related to a feature is not easy to do in practice [18], since features crosscut different files while the tracing of features to code might not be explicit.

Feature introduction Conversely, it might be important to progressively deploy a new feature without having it initially in the deployed version (as in trunk-based development)

Feature interactions Despite the general rule of thumb adopted by practitioners that feature interactions should be avoided, this is hard to maintain in the long run. In [41], authors mine feature toggles and their interactions in five open-source projects, and they show that 7% of feature toggles interact with each other, 33% of them interact with another code expression, and their interactions tend to increase over time (22%, on average). An explicit design time variability model would then allow to more easily specify and reason about feature interactions.

Higher level of abstraction Having an explicit design time variability model would allow connecting feature toggles to explicit goal models (*e.g.,* 3% of users see a given feature) that have been routinely used in the SPL domain [4, 5, 24, 27]. Further, toggling of runtime features could be easily decided by a reasoning engine to transform this form of a dynamic SPL into a Self-adaptive system [15, 37].

3 APPROACH

This section discusses possible implementations of the core idea of this paper: supporting a continuum between design time *Feature Models* and runtime *Feature Toggles*. That is, practitioners can move from one (design-time variability and feature models) to another (runtime variability and toggles) continuously. As exemplified in Figure 1, the overall approach consists in:

(1) modeling all features using a de facto standard variability modeling language such as *Feature Models*
(2) providing realization mechanisms that can be triggered either at derivation time or at runtime
(3) allow a partial resolution of the variability
(4) use the residual variability to generate one or more adapters for feature toggle frameworks

3.1 The Expression Product Line

In order to illustrate the approach, we use the expression product line (EPL) as a running example and as described in [22]. We consider the following grammar for expressions:

Exp ::= Lit | Add | Neg
Lit ::= <non-negative integers>
Add ::= Exp "+" Exp
Neg ::= "-" Exp

Two different operations can be performed on the expressions described by this grammar: first, *printing*, which returns the expression as a string, and second, *evaluation*, which computes the value of the expression. The set of products in the EPL can be described with a feature model, as illustrated in Figure 2. It has two orthogonal feature sets, the ones concerned with data *Lit*, *Add*, *Neg* and the ones concerned with operations *Print* and *Eval*, which makes it representative of the kind of modular decomposition problems we find in the real world. In this example, initially *Lit* and *Print* are mandatory features. The features *Add*, *Neg* and *Eval* are optional.

For instance, if we would select features *Add* and *Eval*, the typical code we would obtain would look like Listing 1:

```
interface Exp { void print(); int eval(); }
class Lit implements Exp {
  int value;
  Lit(int n) { value=n; }
  void print(){ System.out.print(value); }
  int eval(){ return value; }
}
class Add implements Exp {
  Exp expr1; Exp expr2;
  Add(Exp a, Exp b) { expr1=a; expr2=b;}
  void print(){ expr1.print();
    System.out.print("+"); expr2.print();}
  int eval(){ return expr1.eval() + expr2.eval();}
}
```

Listing 1: Expression Problem in Java

If we decide to postpone the choice among using *Add* or not using it (i.e., the resolution of feature *Add* is deferred until runtime as a feature toggle), we need two things:

- a way that both versions of the code can co-exist in the implementation
- some runtime support for making the choice among them.

We can consider 3 different ways of implementing this in languages such as Java or C++:

(1) build a single class hierarchy containing all the possibilities (aka a 150% program) with dynamic tests of the kind: `if Feature.isEnabled("FeatureX") ...`

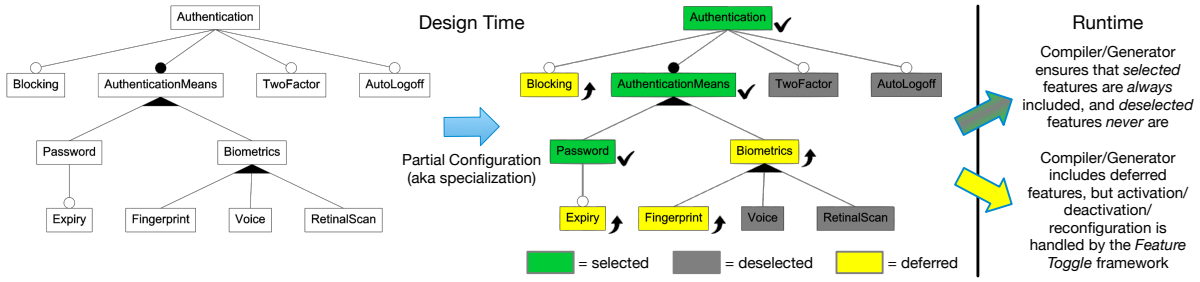

Figure 1: Configuring an Authentication Library at Design time and Runtime

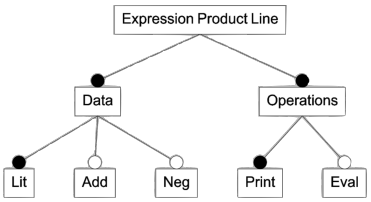

Figure 2: Feature Model for Expression Problem Product Line

(2) rely on polymorphism and dynamic binding among several class hierarchies (each representing a possible combination of feature selections) and configure it with the *Factory* design pattern.
(3) as previously, but using genericity/template parameters to have only one class hierarchy at the source code level (but with the same semantics as with parallel class hierarchies)

All these approaches can make sense in a specific context. Before evaluating their pros and cons, we are going to explain how they work, and how the tedious implementation part could be automated.

3.2 A Simple Java Implementation

The first possible mechanism is simply to guard any toggle feature with an `if` statement, as illustrated in Listing 2, and to raise an exception whenever code from this feature is called while the feature is not selected.

```
interface Exp { void print(); int eval(); }
class Lit implements Exp {
  int value;
  Lit(int n) { value=n; }
  void print(){ System.out.print(value); }
  int eval(){
    if !Feature.isEnabled("Eval")
      throw UnsupportedOperationException;
    return value; }
}
class Add implements Exp {
  Exp expr1; Exp expr2;
  Add(Exp a, Exp b) {
    if !Feature.isEnabled("Add")
      throw UnsupportedOperationException;
    expr1=a; expr2=b;}
  void print(){ expr1.print();
    System.out.print("+"); expr2.print();}
  int eval(){
    if !Feature.isEnabled("Eval")
      throw UnsupportedOperationException;
    return expr1.eval() + expr2.eval();}
}
```

Listing 2: EPL with Feature Toggles

This would work out of the box for most feature toggle frameworks, but it is both inefficient and inelegant, if only because the user would be exposed to constant exception handling.

3.3 Leveraging Polymorphism and Dynamic Binding

An alternative approach would be to rely on polymorphism to offer the same configurable choice. Dealing with a feature such as *Add* is easy because we can leverage built-in modularity mechanisms offered by OO approaches: the feature *Add* is mostly encapsulated in the class *Add*, and the use of a companion *Factory* pattern makes it possible to control whether we allow instances of it to be created. For the *Eval* feature, things are a bit more complicated, which precisely makes this EPL example so useful for the SE community. Within a standard OO approach, it is very difficult to modularize this feature, because it crosscuts all the other concepts of the OO decomposition: Lit, Add and Neg (aka tyranny of the dominant decomposition paradigm [28]).

A first possibility is to build two parallel class hierarchies: a base one with the feature *Print*, and a second one with *Print* and *Eval*.

To distinguish among the two class hierarchies, we can use a naming convention or even better the same names in different name spaces, *e.g.*, several different packages (*e.g.*, expBase and expEval), as illustrated in Listings 3 and 4.

```
package expBase;
public interface Exp {void print();}
class Lit implements Exp {
  int value;
  Lit(int n) {value=n;}
  public void print(){System.out.print(value);}
}
class Add implements Exp {
  E expr1; E expr2;
  Add(E a, E b) { expr1=a; expr2=b;}
  public void print() { expr1.print();
    System.out.print("+"); expr2.print(); }
}
```

Listing 3: EPL with Polymorphism

```
package expEval;
public interface Exp extends expBase.Exp{int eval();}
class Lit extends expBase.Lit implements Exp {
  Lit(int n) {super(n)}
  public int eval() { return value; }
```

```
}
class Add extends expBase.Add implements Exp {
  Add(Exp a, Exp b){ super(a, b);}
  public int eval(){return expr1.eval() +
    expr2.eval();}
}
```

Listing 4: EPL with Polymorphism: the Eval feature

A *Factory* can help configure our product line at runtime, as illustrated in Listing 5. Note that we have a typing issue here, since instantiated objects could be from either packages. The factory is thus parameterized with the wanted type.

```
public interface ExpFactory<E extends Exp> {
  E newLit(int value); E newNeg(E e);
  E newAdd(E e1, E e2);}
public class BaseFactory implements ExpFactory<expBase.Exp>{
  public expBase.Exp newLit(int value)
    {return new expBase.Lit(value);}
  public expBase.Exp newNeg(expBase.Exp e)
    {return Feature.isEnabled("Neg") ?
       new expBase.Neg(e): null;}
  public expBase.Exp newAdd(expBase.Exp e1, expBase.Exp e2)
    {return Feature.isEnabled("Add") ?
       new expBase.Add(e1,e2): null;}
}
```

Listing 5: A Factory to configure the EPL

A similar implementation is necessary for *EvalFactory* that implements *ExpFactory< expEval.Exp >*. Finally, we still need to choose among these two factories as illustrated in Listing 6.

```
public ExpFactory<?> getFactory() {
  return Feature.isEnabled("Eval") ?
    new  EvalFactory() : new BaseFactory();
}
```

Listing 6: Choosing among the 2 Factories

3.4 Leveraging Polymorphism and Genericity

To reduce the amount of code that needs to be written to only one source code hierarchy, we can come up with a solution using formal type parameters from the beginning, as illustrated in Listing 7.

```
public interface Exp {void print();}
public interface ExpEval extends Exp {int eval();}
class Lit<E extends Exp> implements Exp {
  int value;
  Lit(int n) {value=n;}
  public void print(){System.out.print(value);}
}
class LitEval extends Lit<LitEval> implements ExpEval{
  LitEval(int n) {super(n);}
  public int eval() { return value; }
}
class Add<E extends Exp> implements Exp {
  E expr1; E expr2;
  Add(E a, E b) { expr1=a; expr2=b;}
  public void print() { expr1.print();
    System.out.print("+"); expr2.print(); }
}
class AddEval extends Add<ExpEval> implements ExpEval{
  AddEval(ExpEval a, ExpEval b){ super(a, b);}
  public int eval(){return expr1.eval() + expr2.eval();}
}
```

Listing 7: EPL with Polymorphism and Genericity

The big advantage of this approach is that with the very same code base, we can now choose what is decided at derivation time, and what is still there to be decided at runtime. For instance, if we decide that Add is not selected at design time, the Listing 5 can (automatically) be specialized (along the lines proposed in [20]) so that there is no more reference to the class `Add`, which is then excluded from the jar built for this application.

The obvious drawback of this approach is that it is quite heavyweight, even for a simple SPL. In particular, if we now have a second cross-cutting feature such as *Compile*, we would need to have a new interface `ExpCompile` featuring the method `compile()`, and also an interface `ExpEvalCompile` extending both `ExpEval` and `ExpCompile`, plus all the needed implementation classes. That would soon become unmanageable, and that might explain why this path has never been adopted in industry.

This has been a well known problem for a very long time, and it was one of the motivating factors for introducing the notion of *Aspects* in the late 90s, as well as other ways of building software in a more modular way, for instance with Feature-Oriented Programming (FOP), or more recently for Delta-Oriented Programming (DOP), and Concern-Oriented Reuse (CORE). In the rest of this section we describe how this derivation time–runtime continuity could work for DOP and CORE.

3.5 Using Delta-Oriented Programming

In Delta-Oriented Programming [38], a product line is represented by a core module and a set of delta modules. The core module provides an implementation of a valid product that can be developed with well-established single application engineering techniques. Delta modules specify changes to be applied to the core module to implement further products by adding, modifying and removing code. Application conditions attached to delta modules allow handling combinations of features explicitly. A product implementation for a particular feature configuration is generated by incrementally applying all delta modules with valid application condition to the core module. To implement feature toggles with Delta-Oriented Programming, we could modify the solution proposed in [38] with either a runtime test as in Listing 2 or with a factory as in Listing 5. As illustrated in Listing 8, in the case of a feature such as *Add*, the solution is straightforward with a simple modification (addition) of a factory method `newAdd()` in the `ExpFactory` class.

```
delta DAdd when Add {
  adds class Add implements Exp {
    Exp expr1; Exp expr2;
    Add(Exp a, Exp b) { expr1=a; expr2=b; }
  }
  modifies class ExpFactory {
    public Exp newAdd(Exp e1, Exp e2){
      return Feature.isEnabled("Add") ?
        new Add(e1,e2): null;}
    }
}
```

Listing 8: Delta module for Add feature

The case of a cross-cutting feature such as *Eval* is a bit more complex, since it implies adding `eval()` methods in each construct of the EPL. We can of course again rely on a runtime check (as in Listing 2)

to raise an exception if *Eval* is deferred at design time, but not active at runtime when the operation is invoked (see in Listing 9).

```
delta DEval when Eval {
  modifies interface Exp { adds int eval();}
  modifies class Lit {
    adds int eval() {
      if !Feature.isEnabled("Eval")
        throw UnsupportedOperationException
      return value;}
  }}}
```

Listing 9: Delta modules for Eval feature

However, this solution has the same drawbacks as discussed in Section 3.2. However DOP can also be used to allow both versions of a class such as `Lit` to co-exist at runtime as discussed in Sections 3.3 and 3.4, that is one with the method `eval()` and the other one without it, either using parallel class hierarchies or generics. For the sake of space, we only present the first one in Listing 10.

```
delta DEval when Eval {
  add interface expEval.Exp extends expBase.Exp{
    int eval();
  }
  add class expEval.Lit extends expBase.Lit
        implements expBase.Exp {
    int eval() {return value;}
  }
  modifies class Factory {
  public ExpFactory getFactory() {
  return Feature.isEnabled("Eval") ?
    new expEval.Factory() : new expBase.Factory();
} } }
```

Listing 10: Delta modules for Eval feature

3.6 Using CORE

Concern-Oriented Reuse (CORE) [1] is a model-driven approach for building reusable software artifacts that supports feature-oriented modularization. In particular, when building a product line such as the EPL, each feature can be associated with realization models expressed in some modeling language. CORE comes with an algorithm that, given a feature selection, composes all associated realization models using aspect-oriented composition technology.

Figure 3 illustrates how we modeled the *EPL* product line using CORE. The figure shows six design models expressed using the *Reusable Aspect Models* (RAM) modelling language [21]. A RAM model consists of a class diagram describing the design structure, and one or several sequence diagrams describing the design behavior. For space reasons, the sequence diagrams have been omitted.

EPL Model contains the main classes of the *EPL* product line that are present in any configuration, namely the abstract `Exp` class, as well as the class `Lit` and the class `ExpFactory` with an operation `newLit` (because literals are a mandatory feature). `Exp` and `Lit` also define the operation `print`, since *Print* is also a mandatory feature.

The remaining models are all extensions of *EPL Model*. In other words, they increment *EPL Model* with additional classes and operations. *Neg Model* contains the class `Neg` and adds a `newNeg` operation to the factory. Likewise, *Add Model* defines the class `Add` and augments the factory with the `newAdd` operation. *Eval Model* extends the `Exp` and `Lit` classes with the `eval` operation. Finally, the model *NegEval Model* defines the `eval` operation for the class `Neg`, whereas *AddEval Model* defines it for the class `Add`.

The extension dependencies between the six models form a directed acyclic graph (DAG), visualized on the right-hand side of Figure 4 with dashed black dependency arrows. For example, *AddEval Model* extends both *Add Model* and *Eval Model*, which in turn both extend *EPL Model*.

The last step to enable model and code generation of the EPL product line using CORE is to associate the RAM realization models with the features of the EPL. This is shown in Figure 4 with the blue dotted lines. A feature can be realized by potentially several realization models, and a realization model can realize one or more features (e.g., *AddEval Model* realizing *Add* and *Eval* features). Thanks to this many-to-many mapping, feature interactions can explicitly be dealt with.

The CORE derivation algorithm first determines the set of realization models that are to be composed as follows. Given a valid feature selection *Sel*, it first removes all features from *Sel* that do not have any associated realization models. The algorithm then looks for a realization model that realizes all the remaining features in *Sel*. If there is no such realization model, the algorithm looks for realization models that realize a subset of features in *Sel* of size $|Sel| - 1$, then of size $|Sel| - 2$ and so on. Once one or several realization models are found, they are added to the list of realization models to be composed, and the features they realized are removed from *Sel*. The algorithm continues until *Sel* is empty. For example, for the valid EPL feature selection $S = EPL, Data, Lit, Add, Operations, Print, Eval$ the algorithm would first prune the features *Data*, *Lit*, *Operations* and *Print*, since they do not have associated realization models. The algorithm would then not be able to find a realization model that realizes the three remaining features *EPL*, *Add* and *Eval*, so it would look for realization models that realize a subset of size 2. It would therefore pick *AddEval Model*, which realizes *Add* and *Eval*, and then *EPL Model* to cover the remaining feature *EPL*.

Following the extension dependency DAG, the algorithm then determines that it needs to compose the models *AddEval Model*, *Add Model*, *Eval Model* and *EPL Model*. To accomplish this, it combines the models in top-down order by successively invoking the aspect-oriented RAM model weaver. By default, the RAM weaver merges model elements with the same name or signature. In our example, *Add Model* is first woven into *EPL Model*. Next, *Eval Model* is composed with the result of the previous composition, and finally *AddEval Model* is woven. The resulting model is shown in Figure 5. Again, the sequence diagrams describing the behavior of the different operations have been omitted for space reasons.

From the woven RAM model, the TouchCORE code generator generates a fully functioning Java implementation. For example, the generated code for the class `ExpFactory` is shown in Listing 11. Note that the runtime variability is handled in the `newAdd` factory method, just like in the Java and Delta implementations.

```
class ExpFactory {
    protected Features myFeatures;
    ExpFactory(Features f) {
        this.myFeatures = f;
    }
    Exp newLit(int value) {
```

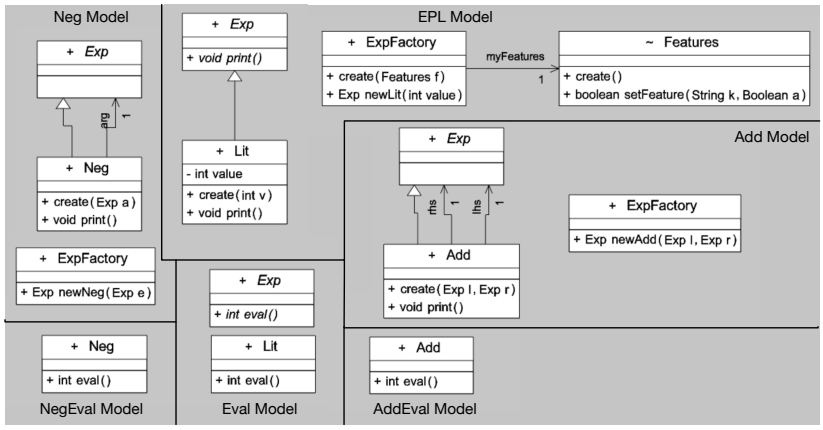

Figure 3: CORE Realization Models for the EPL (each is a standalone UML class diagram)

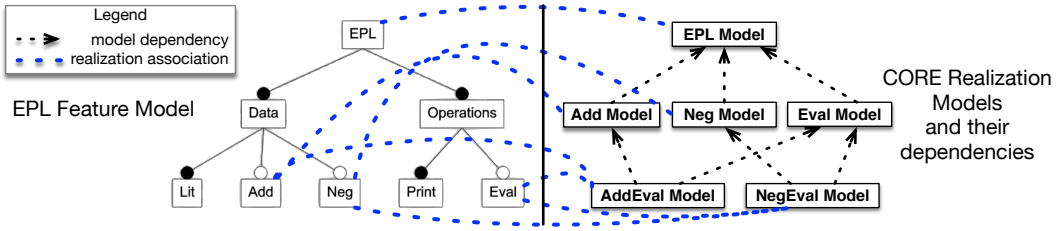

Figure 4: EPL Features and their mappings to CORE Realization Models

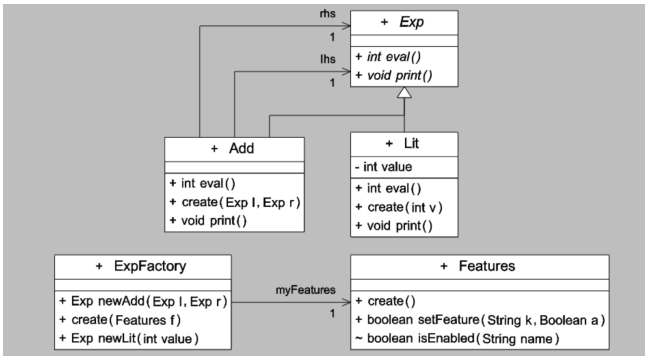

Figure 5: Woven Realization Model for features *EPL*, *Data*, *Lit*, *Add*, *Operations*, *Print*, *Eval*

```
        Lit lit = new Lit(value);
        return lit;
    }
    Exp newAdd(Exp l, Exp r) {
        boolean isEnabled =
            myFeatures.isEnabled("Add");
        Add add = null;
        if (isEnabled) {
            add = new Add(l, r);
        }
        return add;
    }
}
```

Listing 11: Java code of `ExpFactory` generated by CORE

4 EVALUATION

The evaluation of our approach is twofold. First we are going to discuss pros and cons of the variants we have proposed in Section 3. Then we are going to evaluate how the proposed approach scales for a real use case, i.e. the *Authentication* system proposed as a motivation for this work.

4.1 Evaluation of the Variants

In Section 3 we have discussed several possible implementations for a seamless integration of design time and runtime variability resolution. We now ask the following Research Question: (RQ1) What are the pros and cons of variability implementation approaches?

We are going to evaluate them in a qualitative way along the following criteria:

(1) Conceptual simplicity: the amount of effort needed to master the approach for a typical Java programmer.
(2) Source code compactness: the number of lines of code that need to be manually written
(3) Bytecode/binary code compactness: ability to embed only the necessary features (avoid ghost variability) with a possible impact on memory footprint and performance.
(4) *Xor* feature support: ability to support Xor features, *e.g.*, in the EPL the base type returned by `eval()` could have been either `int` or `double`.
(5) Safety: ability to run the SPL in a usable and safe manner, i.e. preventing unsupported features to be activated by accident as described in [23].
(6) Extensibility at runtime: ability to add/remove features at runtime by loading/unloading the relevant code.
(7) Dynamic reconfiguration support: ability to change the configuration (conforming to the feature model) at runtime (i.e.

Approach →	Dyn. Tests	Par. hier-archies	Genericity	DOP	CORE
Simplicity	++	+	-	-	- -
Source compact.	++	- -	+	+	+
Binary compact.	+	- -	+	++	++
Xor features	- -	++	-	-	++
Safety	- -	-	-	++	++
Add/rem. feat.	-	++	-	++	++
Dyn. reconf.	++	-	-	-	+
Maintainability	+	- -	-	++	+

Table 1: Evaluation Summary

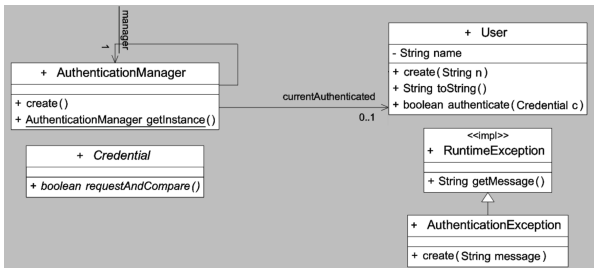

Figure 6: Design Class Diagram for the Authentication Base

not just at startup time), which might require support for data conversion between old and new data formats.

(8) Maintainability: how easy it is to evolve the code.

We now get back to the various approaches discussed in Section 3 and then synthesize our evaluation in Table 1.

Dynamic Tests. With the solution based on only one class hierarchy and dynamic tests of the kind: `if Feature.isEnabled("FeatureX") ...` the code is rather compact and very simple to understand and maintain. A dynamic reconfiguration can easily change the value of fields because they are always there. However, it is impossible to add/remove features after deployment, and quite often difficult to deal with *xor* features related to object static types. Also, the memory footprint could be significant due to possibly useless fields always being present, and the client code has to deal with the inelegant `UnsupportedOperationException` and is subject to have features activated by accident as described in [23].

Parallel Class Hierarchies. With this solution we need as many class hierarchies as the cartesian product of all features adding attributes or code in existing classes. The good thing though of having any combination of feature encapsulated in its own class hierarchy is that we can easily add/remove features after deployment using a dynamic class loader. Changing the runtime state of the program after a reconfiguration however requires explicit data conversion between any possible old structures towards any possible new ones. In complex SPLs, that could be a lot of code to write (but this code could be generated). On the downside, there is a huge duplication of code, both at source and binary/bytecode level, which makes it hard to properly test and maintain.

Using Genericity/Templates. Since this solution is somehow a source code-level trick to reduce the code explosion of parallel class hierarchies, this approach shares most pros and cons with the previous one except for the source compactness which is now much better. This come at the price of a code slightly more difficult to understand and maintain. Finally, the runtime footprint depends on the programming language: in Java generic code is shared so that's a pro, while in C++ there is still lots of binary code duplication due to template expansion.

Using DOP or CORE. Both DOP and CORE are generative solutions, and thus can potentially generate any of the 3 Java code variants discussed above. Their strong point is that only what is really needed at runtime is generated (no ghost variability left). So at code level they share with previous solutions most of their pros and cons (in Table 1 we thus took the best choice for them). They only differ from these previous solutions by the fact that they can generate most of the tedious and repetitive code. This includes conversions among data structures in case of dynamic reconfigurations, i.e., to generate a serializer and a sloppy deserializer for any possible realization model. Thus, code maintainability and evolution is no longer a problem. Still, in either case the user has to learn a new formalism (DOP or CORE), which might be a problem in practice.

To summarize, there is no best solution in general, so depending on the project, one could choose among these possibilities or even come up with new ones – we do not claim that we have been exhaustive in listing them. We were interested in an existential proof only.

4.2 The Authentication Case Study

In this subsection, we answer our second Research Question (RQ2): Can we transition between design time and runtime variability on a real-case SPL?

As discussed above, a generative approach is helpful to limit the amount of code that has to be manually written. We thus specialize this RQ2 in the context of using CORE for designing and implementing an *Authentication System*, i.e., a reusable library offering authentication functionality with different kind of credentials. It also offers other optional features such as expiring credentials, automated blocking caused by repeated failed authentication attempts and automatic logout due to inactivity. The feature model of *Authentication* was shown in Figure 1. The feature-oriented modularization of our implementation of *Authentication* is done using CORE, and the runtime enabling / disabling of features is done with the *Togglz* framework.

4.2.1 Authentication Base. The base design class model of *Authentication* defines a `User` class that represents the entities that can be authenticated, as well as an abstract class `Credential` that defines an abstract operation `requestAndCompare`. The idea is that different kind of credentials subclass the `Credential` class, add the state needed to store credential-specific information, and override the `requestAndCompare` operation with behavior that prompts the user to provide specific credentials and compares the obtained information with the credential information that is already stored in the system. The base model also defines an *AuthenticationException* that can be thrown if authentication fails.

4.2.2 Designing Individual Credentials. Figure 7 on the right shows the design of the *Password* feature. The class `Password`, a subclass of `Credential`, is added to the system if the *Password* feature is chosen. On the left side of the figure is a model that uses a direct reference

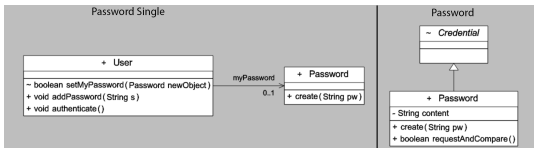

Figure 7: Design Class Diagram for Password

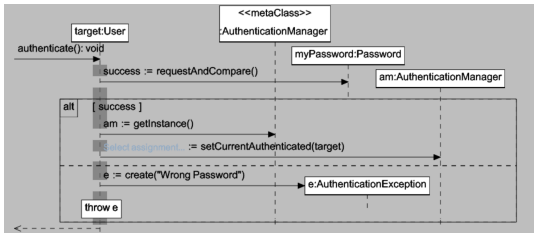

Figure 8: Sequence Diagram for authenticate Operation

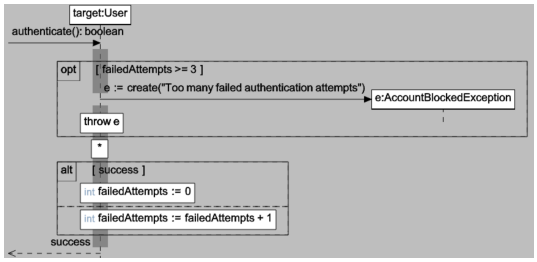

Figure 9: Aspect Sequence Diagram for Blocking

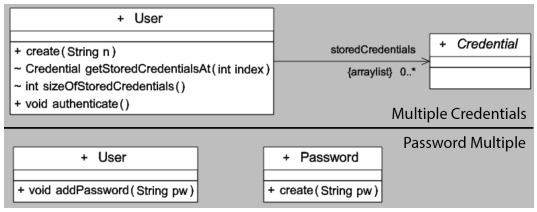

Figure 10: Design Class Diagram for Multiple Credentials

with multiplicity 0..1 for linking users to their password. This model is only used when *Password* is the only active authentication means feature.

Figure 8 shows a sequence diagram that specifies the behavior of the `authenticate` operation in the case where the only available credentials are passwords. A similar design was done for the other credentials, but not shown here for space reasons.

The design of the *Blocking* feature shown in Figure 9 exemplifies how existing behavior can be augmented. The aspect-oriented sequence diagram injects behavior that verifies that there have not been too many failed authentication attempts before the original behavior of `authenticate`, represented by a white box with a $*$ inside it. After the original behavior, the number of failed attempts are updated depending on the result.

4.2.3 Handling Multiple Credentials. The situation is different if multiple means for authentication are desired. In that case, a user can have multiple credentials. This is achieved with the model *MultipleCredentials* shown at the top of Figure 10. It prescribes that a user has an array list of associated credentials. The models dealing with a specific kind of credential then simply have to add a corresponding credential instance into the array list.

The default behavior of the `authenticate` operation with multiple credentials has to look into the array list to find all the credentials stored for a given user. It then prompts the user to choose which form of authentication should be used, and then calls the `requestAndCompare` operation of the chosen credential.

In case the *Two Factor* feature is selected, the user must provide two forms of authentication. To achieve this, an aspect sequence diagram was designed that wraps a loop fragment around the original behavior of `authenticate` in order to execute it twice.

4.2.4 Using the Authentication Concern. Given a design-time configuration of the system containing *selected* features, *deselected* features and *deferred* features, our approach generates code that contains all structure and behavior of selected and deferred features and does not contain any structure and behavior of deselected features. Furthermore, it generates code that uses the *Togglz* framework to manage the activation and deactivation of deferred features at runtime.

The first step is to generate the necessary configuration files for the *Togglz* framework. The API of *Togglz* requires the developer to implement a Java `enum` that lists the features that are to be managed by *Togglz* at runtime. In our case, all deferred features should be handled by *Togglz*. For example, let us assume the user selects the features *Password*, deselects *Voice*, *RetinalScan*, *TwoFactor* and *AutoLogoff*, and delays *Fingerprint*, *Expiry* and *Blocking*. Listing 12 shows the Java code that is generated for that configuration[1]. It essentially tells *Togglz* that there are three features that the framework needs to manage.

```
import org.togglz.core.Feature;
import org.togglz.core.annotation.Label;
import org.togglz.core.context.FeatureContext;

public enum DelayedFeatures implements Feature {
    @Label("Blocking")
    BLOCKING,
    @Label("Expiry")
    EXPIRY;
    @Label("Fingerprint")
    FINGERPRINT;
    public boolean isActive() {
        return FeatureContext
        .getFeatureManager().isActive(this);
    }
}
```

Listing 12: Feature Configuration for Togglz

The next step is to determine the realization models that the chosen configuration requires. Using the realization mappings illustrated in Figure 11 and the algorithm explained in section 3, the model *PasswordFingerprint* (which covers *Password* and *Fingerprint*), as well as the models *Blocking* and *Expiry* are chosen.

The next step consists in preparing the behavioral realization models that realize features that are deferred, which in our case are *PasswordFingerprint*, *Blocking* and *Expiry*. These models must

[1]There is also a *TogglzConfig* class that has to be generated, but its content is straightforward and therefore omitted in this paper for space reasons.

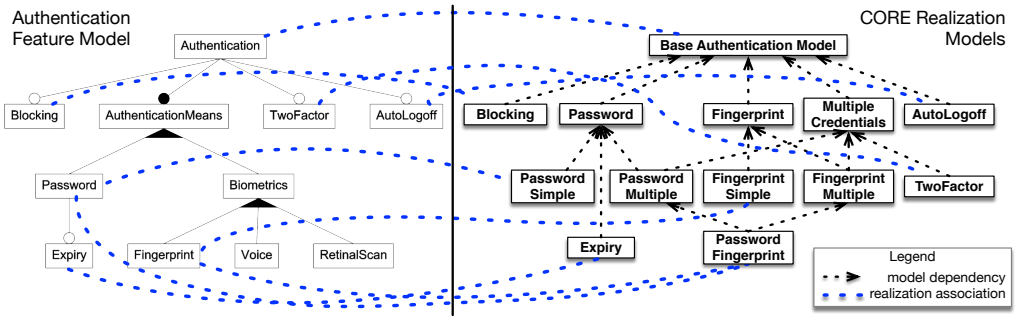

Figure 11: Authentication Features and their mappings to CORE Realization Models

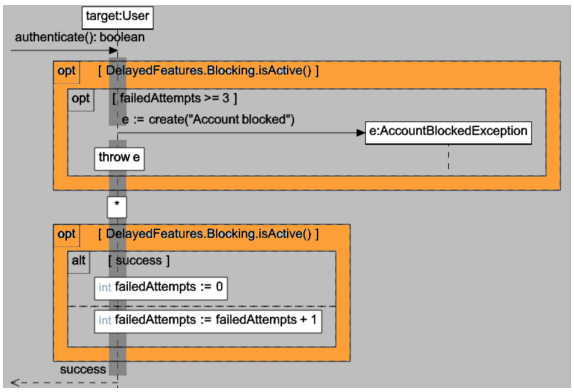

Figure 12: Blocking behavior with Togglz Check

be augmented with a call to the *Togglz* framework to determine whether the corresponding feature is active or not before executing the feature's behavior. For example, Figure 12 shows how the original behavior of *Blocking* from Figure 9 is encapsulated within **opt** fragments that are conditioned using the Togglz-provided API call DelayedFeatures.Blocking.isActive().

The final step is now to compose the realization models corresponding to the selected features with the augmented realization models of the deferred features, as well as with all the models that these models depend on (in our case *BaseAuthenticationModel*, *Password*, *Fingerprint*, *MultipleCedentials*, *PasswordMultiple*, *FingerprintMultiple*, *PasswordFingerprint*, *Blocking*, and *Expiry*). The composition is done pairwise in top-down order of the dependency graph illustrated in Figure 11. The final result of the composition is shown in Figure 13. The colors indicate from which realization model the model elements originated from.

4.2.5 Discussion. *Authentication* is a realistic example of a product line that can benefit significantly from our proposed integration of feature models and feature toggles. Different systems have different authentication needs, and it is therefore very useful to express the functionality offered by the *Authentication* concern as a feature model that groups related features and hence streamlines the selection of desired features for the concern user.

Furthermore, systems requiring authentication are typically concerned with security. Dead code, i.e., code that is included in an executable, but never used at runtime, is considered a security risk, because it enlarges the attack surface available for intrusion. With our approach, features that are not needed ever can be explicitly excluded by *deselecting* the feature in the feature model. This also optimizes the resource consumption of the implementation, because the state kept in memory for authentication is minimal. Similarly, to get the most efficient implementation, features that should always be present in the system are *selected* in the feature model. As a result, no unnecessary checks are executed at runtime. Finally, features that are neither selected nor deselected are deferred to runtime, meaning that the decision whether to activate them or not is done at runtime using the feature toggles framework. For example, if suspicious activity is detected, it would be possible to temporarily enforce two-factor authentication using a feature toggle until the situation is resolved.

5 RELATED WORK

Unifying design time and runtime variability is far from being a new idea. For instance [20] discussed how the use of an abstract factory pattern coupled with static analysis makes it possible to remove unselected features at compile time.

Damiani et al. [9] present a core calculus that extends DOP with the capability to switch the implemented product configuration at runtime. A dynamic delta-oriented SPL has a dynamic reconfiguration graph that specifies how to switch between different feature configurations. Dynamic DOP supports also (unanticipated) software evolution such that at runtime, the product line declaration, the code base and the dynamic reconfiguration graph can be changed in any way that preserves the currently running product, which is essential when evolution affects existing features. DynamicDOP can be seen a special case of this paper, where no decision on the Feature Diagram is made at design time, and all the variability is dealt with at runtime. Our approach uses CORE to hardcode selected features and exclude deselected features from the generated executable, and runtime variability is handled by the feature toggle framework. Another difference is that reference [9] only proposes a calculus, but no implementation, because it would need mechanisms that are not available even in recent JVMs.

For the implementation of dynamic SPLs [6, 7, 16] in FOP, Rosenmüller et al. [35] support flexible feature binding to allow selecting features statically at compile-time using superimposition or dynamically at build-time using the decorator pattern. The variability of feature binding is achieved by code transformations for integrating static and dynamic feature bindings. They also use transformation rules on feature models to provide composition safety of dynamic binding. Rosenmüller et al. [36] extend their approach to support runtime adaptation and self-configuration on top of flexible binding

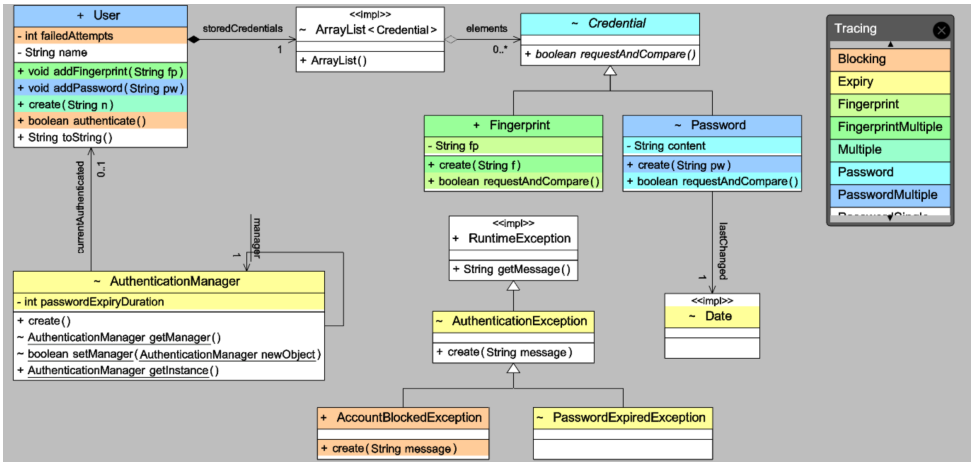

Figure 13: Composed Authentication Design Structure for <Selected: Password, Deferred: Expiry, Fingerprint, Blocking>

units. They use feature-based adaptation rules to describe SPL adaptation in a declarative way. The adaptation mechanism transforms the feature model of an SPL according to the binding units of the generated DSPL, thus making it possible to change the behavior of a program at runtime. However, this dynamic adaptation approach does not support changing the state of the program (i.e., a dynamic reconfiguration cannot add/remove or change the value of fields).

FeatureC++ [3] provides means to dynamically compose feature modules. In particular, Apel et al. investigate the combination of FOP and aspect-oriented programming (AOP) to eliminate shortcomings of FOP to capture dynamic cross-cutting modularity. However, runtime reconfiguration – including an update of the heap structures according to the new feature configuration – is not supported. Günther and Sunkle [13, 14] present an extended version of rbFeatures, a FOP implementation in Ruby, which provides runtime adaptation, variant modification and configuration of software product lines. Chakravarthy et al. [8] describe a technique to provide binding-time flexibility in a modular manner by using a combination of design-patterns and AOP. A pattern encapsulates the variation point and targeted aspects set the binding times of the pattern participants. However, they do not consider the evolution of feature models and, therefore, handle only anticipated changes. Ribeiro et al. [34] investigate whether AspectJ provides modularity when implementing features with flexible binding times. This study leads to the conclusion that, in a general case, AspectJ does not provide modularity in a DSPL. Andrade et al. [2] create three AspectJ-based idioms to implement flexible feature bindings and evaluate those using case studies. The idioms are based on exploiting specific AspectJ features and weaving capabilities. Dinkelaker et al. [11] propose an approach for DSPLs which combines dynamic aspects, runtime models of aspects, as well as detection and resolution of aspect interactions, but they do not consider the change of features at runtime. In contrast, we leverage feature toggles that offer a practical and mainstream support for reconfiguring software systems. Besides, developers can control what features should be (de-)activated at compile-time.

Post et al. [29] propose to lift compile-time variability into runtime variability. The purpose of lifting is to verify SPLs by then leveraging static analysis and model checking techniques. In [42], a formalization of variability lifting is proposed through the consideration of Featherweight Java.

Feature toggles are often discussed in grey literature [10, 12, 17, 25, 40] and have recently received increased attention in academia. Works include the study of best practices [10, 23, 31, 32], the classification and implementation [12, 25] as well as the removal of feature toggles [19, 33]. The differences between feature toggles and configuration options are also discussed in [26, 30, 32]. In this paper, we leverage feature toggles to realize runtime variability, allowing a flexible continuum with design-time variability.

Overall, despite several research efforts, the seamless integration of variability at design time and at run time is still lacking in SPL engineering. We aim to fill this practical gap, and we hope that formalization, methods and tools will emerge from this line of work.

6 CONCLUSION

Several works have already identified the proximity of the concept of *Feature Toggle* with the notion of *Feature* found in *Software Product Lines*. In this paper, we proposed to go one step further in unifying these concepts to provide a seamless transition between design time and runtime variability resolutions for industry mainstream programming platforms such as Java or C++.

We proposed to model all the variability using a feature model. Then this feature model could be partially resolved at design time (yielding an incomplete product derivation), the unresolved variability being used to generate feature toggles that can be enabled/disabled during runtime. We discussed pros and cons of several possible implementations of these ideas on the toy example of the *Expression Product Line*, and then showed how it could scale to build a configurable authentication system, where a partially resolved feature model can interface with a popular feature toggle framework. We have shown that a proper handling of this seamless design-time/runtime variability resolution for mainstream OO languages require specific structuration patterns that can be tedious to handle manually and hence call for some form of at least partial code generation. We have discussed how CORE could be used for that. As future work, we aim to further apply our approaches in different engineering contexts, technological spaces, and industries.

REFERENCES

[1] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. 2013. Concern-Oriented Software Design. In *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - MODELS 2013 (Lecture Notes in Computer Science, Vol. 8107)*. Springer Berlin Heidelberg, 604–621. https://doi.org/10.1007/978-3-642-41533-3_37

[2] Rodrigo Andrade, Henrique Rebêlo, Márcio Ribeiro, and Paulo Borba. 2013. Aspectj-based idioms for flexible feature binding. In *2013 VII Brazilian Symposium on Software Components, Architectures and Reuse*. 59–68.

[3] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. 2005. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *International Conference on Generative Programming and Component Engineering*. 125–140.

[4] Mohsen Asadi, Ebrahim Bagheri, Dragan Gašević, Marek Hatala, and Bardia Mohabbati. 2011. Goal-driven software product line engineering. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. 691–698.

[5] Clarissa Borba and Carla Silva. 2009. A comparison of goal-oriented approaches to model software product lines variability. In *International Conference on Conceptual Modeling*. 244–253.

[6] Rafael Capilla and Jan Bosch. 2011. The promise and challenge of runtime variability. *IEEE Computer* 44, 12 (2011), 93–95.

[7] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. 2014. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *Journal of Systems and Software* 91 (2014), 3–23.

[8] Venkat Chakravarthy, John Regehr, and Eric Eide. 2008. Edicts: implementing features with flexible binding times. In *Proceedings of the 7th international conference on Aspect-oriented software development*. 108–119.

[9] Ferruccio Damiani, Luca Padovani, Ina Schaefer, and Christoph Seidl. 2017. A core calculus for dynamic delta-oriented programming. *Acta Informatica* 55, 4 (Jan. 2017), 269–307. https://doi.org/10.1007/s00236-017-0293-6

[10] Andy Davies. 2018. Feature Toggles The Good, The Bad, and The Ugly with Andy Davies. https://www.youtube.com/watch?v=r7VI5x2XKXw.

[11] Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, and Mira Mezini. 2010. A dynamic software product line approach using aspect models at runtime. In *5th Domain-Specific Aspect Languages Workshop*.

[12] Martin Fowler. 2021. Feature Toggles (aka Feature Flags). https://martinfowler.com/articles/feature-toggles.html.

[13] Sebastian Günther and Sagar Sunkle. 2010. Dynamically adaptable software product lines using ruby metaprogramming. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*. 80–87.

[14] Sebastian Günther and Sagar Sunkle. 2012. rbFeatures: Feature-oriented programming with Ruby. *Science of Computer Programming* 77, 3 (2012), 152–173.

[15] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic Software Product Lines. *Computer* 41, 4 (2008), 93–95. https://doi.org/10.1109/MC.2008.123

[16] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic software product lines. *IEEE Computer* 41, 4 (2008), 93–95.

[17] Santosh Hari. 2020. Feature flags: the toggle, the A/B test and the canary - NDC Oslo 2020. https://www.youtube.com/watch?v=FD5fX02QCmY.

[18] Juan Hoyos, Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Albeiro Espinosa Bedoya. 2021. On the Removal of Feature Toggles. *Empirical Software Engineering* 26, 2 (Feb. 2021). https://doi.org/10.1007/s10664-020-09902-y

[19] Juan Hoyos, Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Albeiro Espinosa Bedoya. 2021. On the Removal of Feature Toggles. *Empirical Software Engineering* 26, 2 (2021), 1–26.

[20] Jean-Marc Jézéquel. 1998. Reifying configuration management for object-oriented software. In *20th International Conference on Software Engineering (ICSE)*. Kyoto, Japan. https://hal.inria.fr/inria-00372744

[21] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. 2009. Aspect-Oriented Multi-View Modeling. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development - AOSD 2009, March 1 - 6, 2009*. ACM Press, 87 – 98.

[22] Roberto E Lopez-Herrejon, Don Batory, and William Cook. 2005. Evaluating support for features in advanced modularization technologies. In *European Conference on Object-Oriented Programming*. 169–194.

[23] Rezvan Mahdavi-Hezaveh, Jacob Dremann, and Laurie Williams. 2021. Software development with feature toggles: practices used by practitioners. *Empirical Software Engineering* 26, 1 (2021), 1–33.

[24] Raúl Mazo, Juan C Muñoz-Fernández, Luisa Rincón, Camille Salinesi, and Gabriel Tamura. 2015. VariaMos: an extensible tool for engineering (dynamic) product lines. In *Proceedings of the 19th International Conference on Software Product Line*. 374–379.

[25] Mark McKenna and Josh Allen. 2016. Feature Toggles: Lunch & Learn. https://www.youtube.com/watch?v=gxm1C92XhCQ.

[26] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. 2020. Exploring differences and commonalities between feature flags and configuration options. In *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 233–242. https://doi.org/10.1145/3377813.3381366

[27] Gunter Mussbacher, João Araújo, Ana Moreira, and Daniel Amyot. 2012. AoURN-based modeling and analysis of software product lines. *Software Quality Journal* 20, 3 (2012), 645–687.

[28] Harold Ossher and Peri Tarr. 2002. *Multi-Dimensional Separation of Concerns and the Hyperspace Approach*. Springer US, Boston, MA, 293–323. https://doi.org/10.1007/978-1-4615-0883-0_10

[29] Hendrik Post and Carsten Sinz. 2008. Configuration lifting: Verification meets software configuration. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 347–350.

[30] Eduardo S Prutchi, Heleno de S. Campos Junior, and Leonardo GP Murta. 2021. How the adoption of feature toggles correlates with branch merges and defects in open-source projects? *Software: Practice and Experience* 52, 2 (2021), 506–536.

[31] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C Rigby, and Bram Adams. 2016. Feature toggles: practitioner practices and a case study. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 201–211.

[32] Md Tajmilur Rahman, Peter C. Rigby, and Emad Shihab. 2018. The modular and feature toggle architectures of Google Chrome. *Empirical Software Engineering* 24, 2 (July 2018), 826–853. https://doi.org/10.1007/s10664-018-9639-0

[33] Murali Krishna Ramanathan, Lazaro Clapp, Rajkishore Barik, and Manu Sridharan. 2020. Piranha: Reducing feature flag debt at uber. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 221–230.

[34] Márcio Ribeiro, Rodrigo Cardoso, Paulo Borba, Rodrigo Bonifácio, and Henrique Rebêlo. 2009. Does aspectj provide modularity when implementing features with flexible binding times?. In *Third Latin American Workshop on Aspect-Oriented Software Development (LA-WASP 2009), Fortaleza, Cear'a, Brazil*. 1–6.

[35] Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. 2011. Flexible feature binding in software product lines. *Automated Software Engineering* 18, 2 (2011), 163–197.

[36] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. 2011. Tailoring dynamic software product lines. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*. 3–12.

[37] Pete Sawyer, Raul Mazo, Daniel Diaz, Camille Salinesi, and Danny Hughes. 2012. Using constraint programming to manage configurations in self-adaptive systems. *IEEE Computer* 45, 10 (2012), 56–63.

[38] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond*, Jan Bosch and Jaejoon Lee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 77–91.

[39] Gerald Schermann, Jürgen Cito, and Philipp Leitner. 2018. Continuous Experimentation: Challenges, Implementation Techniques, and Current Research. *IEEE Software* 35, 2 (2018), 26–31. https://doi.org/10.1109/MS.2018.111094748

[40] Split. 2020. Feature Flag Maintenance. https://www.youtube.com/watch?v=qbVNbMSzy0.

[41] Xhevahire Tërnava, Luc Lesoil, Georges Aaron Randrianaina, Djamel Eddine Khelladi, and Mathieu Acher. 2022. On the Interaction of Feature Toggles. In *VaMoS 2022 - 16th International Working Conference on Variability Modelling of Software-Intensive Systems*. Florence, Italy. https://doi.org/10.1145/3510466.3510485

[42] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming* 85, 1 (2016), 125–145.