Variability Encoding: From Compile-Time to Load-Time Variability

Alexander von Rhein[a,**], Thomas Thüm[b], Ina Schaefer[b], Jörg Liebig[a], Sven Apel[a,*]

[a]*University of Passau, Innstraße 33, D-94032, Germany*
[b]*TU Braunschweig, Mühlenpfordtstraße 23, D-38106, Germany*

Abstract

Many software systems today are configurable. Analyzing configurable systems is challenging, especially as (1) the number of system variants may grow exponentially with the number of configuration options, and (2) often existing analysis tools cannot be used for configurable systems. Recent work proposes to automatically transform compile-time variability into load-time variability—called *variability encoding*—with the goal of reusing existing analysis tools for analyzing configurable systems and improving analysis performance compared to analyzing all system variants in a brute-force manner. However, it is not clear whether one can automatically find an efficiently analyzable load-time configurable system for any given compile-time configurable system. Also, for many analyses, we need guarantees that the load-time configurable system precisely encodes the behavior of all system variants that can be statically derived. We address both issues (1) by developing a formal model of variability encoding based on FEATHERWEIGHT JAVA, (2) by proving that variability encoding preserves variant behavior with respect to a core set of language mechanisms, (3) by discussing how our work extends to more complex language mechanisms that elude our formal model, and (4) by sharing our experience with implementing and using variability encoding in real-world applications.

Keywords: configurable systems, variability encoding, Featherweight Java

1. Introduction

Many software systems today provide a rich set of *configuration options*. Users can derive a custom *system variant* by selecting configuration options according to

*Corresponding author
**Principal corresponding author
Email addresses: `rhein@fim.uni-passau.de` (Alexander von Rhein), `t.thuem@tu-braunschweig.de` (Thomas Thüm), `i.schaefer@tu-braunschweig.de` (Ina Schaefer), `joliebig@fim.uni-passau.de` (Jörg Liebig), `apel@fim.uni-passau.de` (Sven Apel)

their requirements. Variability is a major success factor in software engineering, but also a source of complexity. Configuration can take place at different points in time, including at *compile*, *load*, and *run* time. While compile-time configuration is most resource efficient and safe in that unnecessary and possibly vulnerable functionality is excluded from the compiled system variants, load-time and run-time configuration are more flexible in that systems are configurable in later stages of the deployment process.

Analyzing configurable systems is challenging. With only few configuration options, one can typically derive myriads of different system variants. This exponential explosion rules out exhaustive analysis approaches that derive and analyze all system variants individually. While considering only a small subset of variants is viable (cf. sample-based analysis [45]), the statements one can make about the configurable system as a whole are necessarily limited to the subset and are therefore incomplete with respect to the entire configurable system. For example, it is certainly desirable to know for the developers of a crypto library, such as OpenSSL with 589 compile-time configuration options [34], whether it leaks confidential information in *any* of its variants. For example, the heartbleed bug of OpenSSL, discovered in April 2014, occurs only if a certain configuration option is enabled.

Recently, researchers have developed dedicated *variability-aware* analysis tools that operate directly on the code of the configurable system, not on the code of the generated system variants [45]. The key idea is to represent the configurable system including all variability in a single representation, for example, an abstract syntax tree or a control-flow graph that covers all system variants. Analyzing such a representation requires to make the analysis method and the corresponding tool variability-aware [26]. While designing variability-aware analyses has been successful for a number of analysis problems [45], including type checking [2, 16, 28], static analysis [9, 10, 34], testing [30, 38], and model checking [12, 31], their implementation is tedious and error prone.

As an alternative to making the analysis tools variability-aware, one can *lift* compile-time variability to load-time variability. The key idea is to generate, for a given compile-time configurable system, a corresponding *variant simulator* (a.k.a. *product simulator* [5] or *metaproduct* [46]), which simulates the behavior of every individual system variant, based on configuration parameters set at load time. The resulting simulator is solely implemented in the host language and incorporates all variability. It can be analyzed as a whole, considering all variability and configuration knowledge. Existing tools that can handle variability (or non-determinism) at run time can be used to analyze behavioral properties of variant simulators and project the results on the expected behavior of the individual system variants. The overall transformation process is called *variability encoding* [4] (or *configuration lifting* [41]).

A number of recent analysis approaches take advantage of variability encoding to *simultaneously* explore *all* possible execution paths induced by the variability of the corresponding configurable system [4, 5, 13, 30, 41, 45, 46, 47]. These approaches require that the behavior of the variant simulator captures the behavior of the variants as accurately as possible. So far, it has only been discussed informally whether variability encoding guarantees behavior preservation. We close this gap by defining a *behavior-preservation property*, which states that, for each valid execution path of a system variant, a path with the same observable behavior exists in the variant simulator, and vice versa. We prove this property for variability encoding of a simple, formal JAVA-like language.

In general, it is always possible to create a corresponding variant simulator for every configurable system. A trivial way is to generate all system variants, and to create a wrapper that dispatches among them based on given command-line parameters. However, this is certainly not desirable. The goal is that the variant simulator, while accurately capturing the behavior of the individual system variants, still avoids replicating code that is reused among different variants. It is exactly this kind of *sharing* that makes variability-aware analysis efficient [26], but that makes variability encoding also challenging to implement [46].

Although being used successfully in the analysis of configurable systems [45], there is no evidence that variability encoding can be implemented sound and complete while still creating a variant simulator with a high degree of sharing for a given configurable system. We develop a formal model of variability encoding based on FEATHERWEIGHT JAVA (FJ) [24] and show that behavior preservation of variant simulators is ensured. To model compile-time variability, we use COLORED FEATHERWEIGHT JAVA (CFJ) [28], an extension of FJ with support for presence conditions on program elements that control the elements' inclusion or exclusion at compile time. By using CFJ, we abstract from actual compile-time configuration mechanisms including feature modules and conditional inclusion with preprocessors, because they can be seamlessly translated to this canonical representation [27, 29]. To model load-time variability and configuration, we introduce FEATHERWEIGHT SIMULATION JAVA (FJSIM), which extends FJ by a conditional construct that dispatches at run time between different execution paths based on configuration options set at load time. We define a transformation that translates a given CFJ program (compile-time variability) to a corresponding FJSIM program (load-time variability), and show that this transformation preserves the behavior of all system variants. In particular, we use a trace semantics and weak bisimulation to prove behavior preservation.

Proving behavioral correctness of variability encoding for full-fledged languages such as JAVA is certainly elusive. That is why we formalize and prove variability encoding for a core language. Based on this core model, we discuss extensions of our model and proof for further language mechanisms. Finally, we report on our experience with implementing and applying variability encoding to real-world systems.

In summary, we make the following contributions:

- A formal model of variability encoding capturing the transformation of the code base of a configurable program (CFJ) into a variant simulator (FJSIM).
- A proof based on a trace semantics and weak bisimulation showing that every variant simulator in FJSIM models the behavior of all variants of the corresponding configurable system in CFJ.
- A discussion of the implementation and correctness of variability encoding in the presence of language constructs common in mainstream programming languages.
- A report of our experience with generating variant simulators for practical applications.

```
 1 class Printer {                    Feature module BasicPrinter
 2   void print(Page p) {
 3     ... // basic printing
 4   }
 5   void print(Page front, Page back) {
 6     printMulti(front, back);
 7   }
 8   void printMulti(Page front, Page back) {
 9     ... // print both pages on one sheet
10   }
11 }
```

```
12 class Printer {                    Feature module Duplex
13   void print(Page front, Page back) {
14     printDuplex(front, back);
15   }
16   void printDuplex (Page front, Page back) {
17     ... // duplex printing
18   }
19 }
```

```
20 class Printer {                    Feature module Color
21   void print(Page p) {
22     if (p.isColored()) {
23       ... // color printing
24     } else { original(p); }
25   }
26 }
```

(a) Using feature modules

```
 1 class Printer {
 2   void print(Page p) {
 3   #if (Color)
 4     if (p.isColored()) {
 5       ... // color printing
 6     }
 7   #else
 8     ... // basic printing
 9   #endif
10   }
11   void printMulti(Page front, Page back) {
12     ... // print both pages on one sheet
13   }
14   #if (Duplex)
15   void printDuplex(Page front, Page back) {
16     ... // duplex printing
17   }
18   #endif
19   void print(Page front, Page back) {
20   #if (Duplex)
21     printDuplex(front, back);
22   #else
23     printMulti(front, back);
24   #endif
25   }
26 }
```

(b) Using preprocessor directives

Figure 1: Two implementations of the printer driver with compile-time variability

2. Background

In this section, we provide the necessary background on compile-time and load-time variability as well as on variability encoding.

2.1. Compile-Time Variability

There are different mechanisms to implement compile-time variability (for an overview, we refer the reader elsewhere [1, 15, 43]). We illustrate two prominent mechanisms for compile-time variability and describe a canonical representation into which both mechanisms can be translated. Our model of variability encoding is based on this canonical representation.

Let us introduce a running example: a configurable printer driver with the three features *BasicPrinter*, *Duplex*, and *Color*. In this example, features are implemented in dedicated feature modules, as shown in Figure 1a. Selected feature modules are composed in a given order to get a system variant [3]. Feature module *BasicPrinter* implements single-page printing as well as printing of two pages side-by-side. This basic implementation can be extended by selecting the optional features *Duplex* and *Color*, which add automatic duplex printing and color printing, respectively. When a feature module re-defines a method that an earlier feature module already introduced, the original method is extended, which is called *method refinement* [1, 7]. In our example, method print(Page) of *Color* (Line 21) extends method print(Page) of *BasicPrinter*

(Line 2). To invoke the original method (print(Page) of *BasicPrinter*), we use the keyword original in the refining method (Line 24). This allows to build method *refinement chains* in which each feature module adds functionality to a method and invokes the original implementation.

In Figure 1b, we show an alternative implementation of the configurable printer driver using C-preprocessor directives. We use #if directives of the C preprocessor to specify under which conditions (i.e., feature selections) parts of the code should be included in a variant. Before preprocessing, we can define values for the #if variables Color and Duplex. Then, during preprocessing, #if conditions are evaluated, and the enclosed code is either included or excluded in the variant. With preprocessor directives, we can express variability at a fine-grained level, for example, at the level of method bodies (Lines 3–9), which is difficult with feature modules [27].

Feature modules and preprocessor directives are compile-time variability mechanisms, which are similar in that the statements of the program are re-arranged at compile-time based on a given configuration. A canonical representation of this static variability is an abstract syntax tree, in which each node is annotated with a presence condition [49, 29]. A *presence condition* ϕ is a predicate over features that determines whether the annotated code is present in a given configuration [14]. We base our approach on annotated abstract syntax trees to abstract from the specifics of different implementation techniques for compile-time variability. In our formal model, we focus on variability in method bodies and on optional methods. This is not a severe limitation, because variability in other language elements—even undisciplined annotations—can always be converted accordingly using code duplication [29, 33], as we show in Section 5. Furthermore, we require that alternative implementations of methods have the same return type and discuss variable types in Section 5.

An important asset of a configurable system is its *variability model* (a.k.a. *feature model* [1]). It defines which configuration options are compatible to form valid variants, and which options are not. In our example, *BasicPrinter* is mandatory, and *Duplex* and *Color* are optional. We encode the variability model $\hat{\Phi}$ as a propositional formula over the set of options of a given configurable system. Each satisfying assignment of the formula is a *valid* selection of options, which can be used for the derivation of a system variant. *Configurations*, denoted with Φ, and *presence conditions*, denoted with ϕ, are also propositional formulas over options. For example, $\phi_1 = \mathit{Color} \vee \mathit{Duplex}$ is a presence condition. Each valid configuration has exactly one satisfying assignment (i.e., each configuration option is either selected or deselected). As $\hat{\Phi}$ is a propositional formula that captures only the constraints among configuration options, not all configuration options of a system need to appear in this formula. For example, the variability model of the printer driver is the trivial propositional formula *BasicPrinter*, because there are no constraints on *Color* and *Duplex*. We use function *sat* to denote satisfiability of propositional formulas. Function $\mathit{sat}(\phi \wedge \hat{\Phi})$ returns true iff there is a variable assignment $\mathcal{A}$ under which ϕ and $\hat{\Phi}$ hold (i.e., $\exists \mathcal{A} : \mathcal{A} \models \phi \wedge \hat{\Phi}$). In terms of configurations, this means that there exists a configuration that is valid under the constraints ϕ and $\hat{\Phi}$.

2.2. Variant Simulators

A *variant simulator* (a.k.a. *product simulator* [5] or *metaproduct* [46]) is a system that contains the code of all variants of a corresponding configurable system (even of mutually exclusive configuration options), and that is able to simulate the behavior of all valid system variants based on load-time parameters (e.g., global variables set by command-line parameters).

Figure 2 shows a variant simulator of our running example. The variant simulator contains all classes and methods of all features modules of Figure 1a. Additionally, it contains a *feature variable* for each configuration option (Line 2). For each method refinement, there is a *feature choice* (i.e., a conditional statement) that dispatches between the refined and the original method implementation. The functionality of the features *Duplex* and *Color* can be activated or deactivated with the newly introduced feature variables Options.Duplex and Options.Color in Figure 2. In this generated variant simulator, most of the code is *shared* among multiple program variants. The code of duplex printing is shared between the two variants that have *Duplex* enabled (Line 13). The code for basic printing has been duplicated (Lines 8 and 9). To further improve sharing, we could restructure the method to avoid duplication (if statement for condition !Options.Color || !p.isColored).

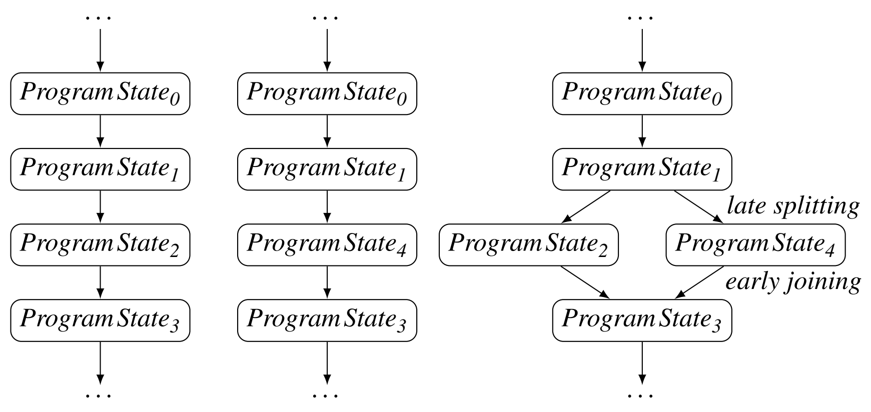
```
                                                Variant Simulator
1  class Options {
2    static boolean Duplex, Color;
3  }
4  class Printer {
5    void print(Page p) {
6      if (Options.Color) {
7        if (p.isColored()) { ... // color printing
8        } else { ... } // basic printing
9      } else { ... } // basic printing
10   }
11   void print(Page front, Page back) {
12     if (Options.Duplex) {
13       printDuplex(front, back);
14     } else {
15       printMulti(front, back);
16     }
17   }
18 }
```

Figure 2: Excerpt of a variant simulator of the printer-driver example

2.3. Analysis Speedup with Variant Simulators

Variant simulators are useful to efficiently analyze the entire variant space of a configurable program. For this simulator-based analysis, one can use, for example a model checker that chooses partial configurations when reaching a feature choice and backtracks to explore other choices when the chosen path terminates. Using a naive variant-based (a.k.a. product-based) approach [45, 48] instead, we would have to analyze each valid variant separately, which is often infeasible in practice [34]. If we analyze the variant simulator of Figure 2, we can find defects using only one analysis run, compared to four runs for analyzing all variants individually. In a series of experiments with variability-aware model checking based on variant simulators, we observed speedups of up to 30 compared to the sequential analysis of all variants [5].

A closer look at our experiments revealed that the observed speedup was due to the effects of *late splitting* and *early joining* in simulator-based analysis [5]. Model checking is basically an exploration of execution traces through the space of possible states that a program can assume at run time. Late splitting means that a common prefix of execution traces of different variants is explored only once for all or a subset of all system variants. A longer common prefix (i.e., later splitting) means that less states



(a) Variants (Deviation in 3rd step) (b) Simulator

Figure 3: Comparison of the state spaces of two variants and a corresponding simulator.

must be explored in total, which often results in a speedup. Figure 3a illustrates that exploration of individual variants is not able to share any parts of traces even though some states are equal. Figure 3b illustrates that, in the corresponding variant simulator, the traces are split only when necessary (after *Program State$_1$*). *Early joining* merges the traces as soon as they reach equal states (*Program State$_3$*), thereby reducing the number of states to be explored. These effects and observations are documented in more detail elsewhere [5, 34].

2.4. The Need for a Formal Correctness Proof

Variability encoding has been used in several research projects and achieved considerable speedup compared to variant-based analysis [4, 5, 44]. In these projects, the subject systems used only relatively simple language features, so it was reasonable to assume that the implementation of variability encoding worked as expected and that the results were valid.

However, in discussions we were often asked if variability encoding is correct in the presence of other language features, such as inheritance, overriding, or switch statements. Implementations of variability encoding mainly deal with such language features separately and regularly miss interactions among them. We chose to approach this problem formally. We define variability encoding based on a small, formal language that already contains many language features that we deem problematic (e.g., method overriding), and we prove behavior preservation in this setting. Then, we discuss how other language features can be dealt with (e.g., switch case statements).

3. A Formal Model of Variability Encoding

In this section, we develop a formal model of variability encoding. We illustrate the process of variability encoding as well as the property of behavior preservation

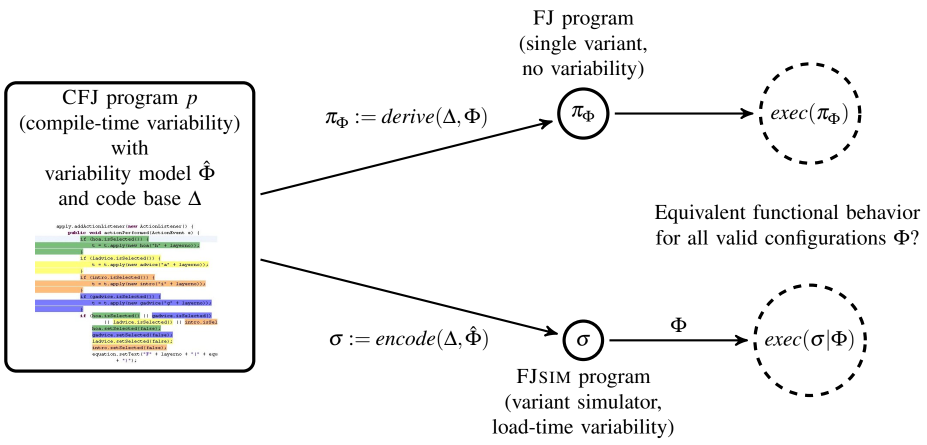


Figure 4: Variability encoding and behavior preservation

between variants and the corresponding variant simulator in Figure 4. Given a configurable program p with variability model $\hat{\Phi}$ and code base Δ, we use weak bisimulation to prove that the execution of any variant π_Φ (generated with configuration Φ) and the variant simulator σ, restricted to Φ, yields the same observable behavior (Section 4). This proof shows the soundness and completeness of variability encoding. In particular, we derive the variant π_Φ with $derive(\Delta, \Phi)$ and encode the variant simulator σ with $encode(\Delta, \hat{\Phi})$ (both functions are defined in Section 3.4). With $exec(\pi_\Phi)$ and $exec(\sigma|\Phi)$ we denote the execution of variant and variant simulator assuming configuration Φ, respectively.

Our model supports all language constructs of FJ, including classes, methods, fields, inheritance, dynamic typecasts, and method overriding. Note that FJ does not support method overloading. If a method m has a certain signature, all other methods named m in the inheritance hierarchy must have exactly the same signature [39, p. 257, *Valid method overiding*]. In Section 5, we discuss how allowing overloading as in JAVA would affect variability encoding. Essentially, in our model a configurable program can have optional methods (either included or excluded from the code) and variable method bodies (alternatives for the default method body).

In Section 3.1, we give an overview of FJ, which is the basis language for our formal model. Next, we describe CFJ [28], which we use to represent compile-time configurable programs (Section 3.2). Then, we introduce the language FJSIM, which we use to represent variant simulators (Section 3.3). Finally, we define how variants and variant simulators are derived from CFJ programs (Section 3.4).

*3.1. Featherweight Java (*FJ*)*

FJ is a functional subset of JAVA with a precise syntax definition, a sound type system, and evaluation rules [39]. We focus on the definitions relevant to our model. In the following description, we use a short notation for lists: $\bar{a}$ denotes a list of syntax elements. Sequences of field declarations, parameter names, and method declarations

P	::=	$(\overline{\mathsf{L}}, \mathsf{t})$	*program*	t	::=		*terms:*
L	::=	class C extends C { $\overline{\mathsf{C\ f;}}$ K $\overline{\mathsf{M}}$ }	*class decl.*			x	*variable*
K	::=	C ($\overline{\mathsf{C\ x}}$) { super($\overline{\mathsf{x}}$); $\overline{\mathsf{this.f=f;}}$ }	*constr. decl.*			t.f	*field access*
M	::=	C m ($\overline{\mathsf{C\ x}}$) { return t; }	*method decl.*			t.m($\overline{\mathsf{t}}$)	*method inv.*
v	::=		*values:*			new C($\overline{\mathsf{t}}$)	*obj. creation*
		new C($\overline{\mathsf{v}}$)	*obj. creation*			(C)t	*cast*

Figure 5: The syntax of FJ [39]

are assumed to contain no duplicate names. For example, the parameter list of a method definition is denoted as $(\overline{\mathsf{C\ x}})$, which expands to $(\mathsf{C}_1\,\mathsf{x}_1, \ldots, \mathsf{C}_n\,\mathsf{x}_n)$.

Figure 5 shows the syntax rules of FJ, a subset of the syntax of JAVA, including rules for class and method declarations and terms, such as field accesses and method invocations. The syntax does not include an assignment operator; fields are only assigned once in the constructor. It also does not contain conditional (e.g., if) or loop constructs (e.g., while). A method body consists of only a single return statement with a term that may contain other nested terms. FJ supports the keyword super only as first statement in constructor bodies. Despite its limitations, FJ is Turing complete, as one can encode the lambda calculus in FJ [39].

An FJ program consists of a class table *CT* and a *start term* init. The evaluation of a program begins with the start term, which is similar to the main method in JAVA. We assume that there is a special variable this, but that this is never used as the name of an argument for a method call. It is considered to be implicitly bound in every method declaration. During evaluation this is substituted with an appropriate object (Figure 6, E-INVKNEW).

Figure 6 gives the small-step operational semantics of FJ. The evaluation rules are designed to be conform as much as possible with JAVA. For example, the rule E-INVKNEW defines method-call resolution. During evaluation, a start term (new C($\overline{\mathsf{v}}$)).m($\overline{\mathsf{u}}$) is evaluated using E-INVKNEW. This rule applies alpha-equivialent substitution ($[\mathsf{x} \mapsto \mathsf{y}]\mathsf{t}_0$) replacing occurences of x in t_0 with y. The term (new C($\overline{\mathsf{v}}$)).m($\overline{\mathsf{u}}$) is evaluated to the body t_0 of method m with substitutions of the keyword this and of uses of formal parameters. The result of this evaluation step is a new term. The term is a fully evaluated value (new C($\overline{\mathsf{v}}$)), or further evaluation steps can be applied to the term. A class inheritance relation, in which class C extends class D, induces a subtype relation C <: D.

$$\frac{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{C\ f}}}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{f}_i \rightarrow \mathsf{v}_i} \qquad \text{(E-PROJNEW)}$$

$$\frac{\mathsf{C} <: \mathsf{D}}{(\mathsf{D})(\mathsf{new\ C}(\overline{\mathsf{v}})) \rightarrow \mathsf{new\ C}(\overline{\mathsf{v}})} \qquad \text{(E-CASTNEW)}$$

$$\frac{\mathit{mbody}(\mathsf{m}, \mathsf{C}) = (\overline{\mathsf{x}}, \mathsf{t}_0)}{\begin{array}{c}(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{u}}) \rightarrow \\ [\overline{\mathsf{x} \mapsto \mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}(\overline{\mathsf{v}})]\,\mathsf{t}_0\end{array}} \qquad \text{(E-INVKNEW)}$$

congruence rules are omitted

Figure 6: Evaluation rules of FJ [39]

3.2. *Colored Featherweight Java (*CFJ*)*

As the source language for variability encoding, we use CFJ [28], which extends FJ with support for compile-time variability using presence conditions attached to program elements. A CFJ program is, in fact, a nested term structure with presence conditions on some terms. This structure is isomorphic to a variable abstract syntax tree, which is a canonical representation of compile-time variability [27, 29]. Given a configuration Φ, we can derive a variant (an FJ program) from a CFJ program. This derivation is, basically, done by checking for all program elements e whether their presence conditions ϕ_{e} are satisfied ($sat(\phi_{\mathsf{e}} \wedge \Phi)$) and removing all elements with unsatisfied conditions. The function *derive* is formally defined in Section 3.4.

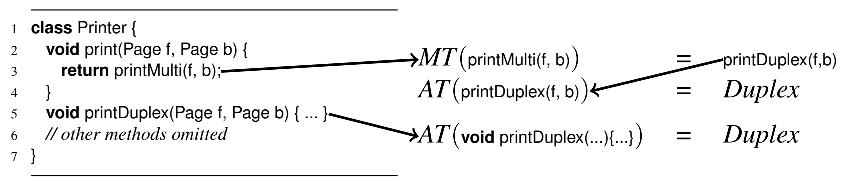
We restrict variability in CFJ such that only complete methods and method bodies can be variable. Also, a method name can be used only once per class. Alternative method implementations can be expressed as alternative expressions in the return statement. This restriction significantly improves the readability of the definitions and the behavior-preservation proof. The described restrictions are no severe limitations to the applicability of our approach as one can always transform more fine-grained variability, such as optional parameters, to our restricted version of CFJ [29, 33]. To prove behavior correctness for a language with more fine-grained variability, one would need to prove either that the translation from this language to our model preserves behavior or that variability encoding works correctly on the language with fine-grained variability. Both is well beyond the scope of this paper, however, we discuss informally how to handle fine-grained variability in some program elements (e.g., optional program variables) in Section 5.

A CFJ program consists of a code base $(CT, AT, MT, \mathsf{init})$ and a variability model $\hat{\Phi}$. The code base consists of a class table CT, an annotation table AT, a metaexpression table MT, and a start term init. The class table and start term are structures as in FJ (Section 3.1).

The variability information of a CFJ program is defined in terms of an annotation table AT and a metaexpression table MT. The annotation table AT contains a presence condition for each program element defining in which configurations (i.e., system variants) the program element is present. The metaexpression table MT contains for each variable program element a either an alternative program element a_1 or the empty program element $\bullet$ (denoting that there is no alternative). Correspondingly, AT contains a presence condition for each alternative program element. Alternatives such as $\mathsf{a}_1 = MT(\mathsf{a})$ can again have alternatives $\mathsf{a}_2 = MT(\mathsf{a}_1)$. During variant derivation, for each program element a, MT is used to recursively search an alternative a_i for which $AT(\mathsf{a}_i)$ is satisfied by the configuration of the derived variant. If such an alternative is found, it substitutes a. We formally define how variants are generated from CFJ programs in Section 3.4. References in AT and MT to program elements are always unambiguous. Figure 7 illustrates how the printer driver can be represented in CFJ. References are illustrated with arrows. Expression printMulti(f,b) in Line 3 is replaced by printDuplex(f, b) iff feature *Duplex* is selected. In this case, the declaration of method printDuplex is included during program generation, too (its AT entry is *Duplex*). For more information on CFJ, we refer to Kästner et al. [28].

3.3. *Featherweight Simulation Java (*FJSIM*)*

We propose FJSIM as the target language of our model of variability encoding. FJSIM does not support compile-time variability, but load-time variability. FJSIM also supports access of superclass methods with the super keyword as in JAVA. Keyword super is necessary to correctly implement method-call resolution in the presence of



```
class Printer {
  void print(Page f, Page b) {
    return printMulti(f, b);
  }
  void printDuplex(Page f, Page b) { ... }
  // other methods omitted
}
```

$MT(\mathsf{printMulti(f, b)}) = \mathsf{printDuplex(f,b)}$

$AT(\mathsf{printDuplex(f, b)}) = \mathit{Duplex}$

$AT(\textbf{void}\ \mathsf{printDuplex(...)\{...\}}) = \mathit{Duplex}$

Figure 7: CFJ program for method print(f,b) of the printer driver; *AT* entries with condition *true* are omitted; arrows illustrate the relation between the code and corresponding entries in *AT* and *MT*.

Syntax

t	::=	...	*terms:*
		(ϕ ? t : t)	*feature choice*
		t.@C.m($\bar{\mathsf{t}}$)	*method-lookup annotation*

Figure 8: The syntax FJSIM adds to FJ. ϕ denotes a presence condition.

optional methods. Keyword super and the capability for load-time variability is only used in variant simulators (not in configurable programs or in variants).

An FJSIM program consists of a code base (CT, init) and a configuration Φ. The configuration is set at load time to simulate only a certain variant. Figure 8 shows the syntax of FJSIM. Similar to keyword this in FJ, we assume that there is a special variable named super that it is never used as the name of an argument to a method. It is considered to be implicitly bound in every method declaration. During evaluation, super is substituted with a *method-lookup-annotation* term that states in which class the lookup for the super method should start.

FJSIM provides a conditional-execution construct called *feature choice*. The feature-choice construct uses a presence condition over feature variables to select one of two alternative terms at run time. The selection is made depending on configuration Φ which has been fixed at load time.

Syntax. At the syntax level, we make two extensions. First, we introduce the ternary operator presence condition ? then : else with a similar semantics as in JAVA. During evaluation of a ternary operator, its presence condition ϕ is evaluated with respect to the given configuration Φ (i.e., $sat(\phi \wedge \Phi)$ means that the presence condition is satisfiable in configuration Φ). Second, we introduce the syntax construct t.@C.m($\bar{\mathsf{t}}$) which denotes that the lookup for method m is started in class C (and may continue in superclasses of C). When the method is executed, t is used as this. We use the syntax extension to model the method-lookup strategy as known from super in JAVA. In the variant simulator, super provides support for accessing overridden methods in subject programs. The syntax extension t.@C.m($\bar{\mathsf{t}}$) is necessary as a term starting with super does not contain information on which class it is embedded in (and where method lookup should start) [39].

Typing. Figure 9 shows the typing rules, auxillary functions and evaluation rules we introduce for FJSIM. The upper part of the figure shows typing rules for typing feature

Typing

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{C} \quad \Gamma \vdash \mathsf{t}_1 : \mathsf{C}}{\Gamma \vdash (\phi \ ? \ \mathsf{t}_0 : \mathsf{t}_1) : \mathsf{C}} \qquad \text{(T-VarEnc)}$$

$$\frac{\begin{array}{c} CT(\mathsf{E}) = \mathsf{class}\,\mathsf{E}\ldots\{\ldots \mathsf{F}\,\mathsf{m}(\overline{\mathsf{G}\,\mathsf{f}})\{\ldots\}\} \\ \mathsf{D} <: \mathsf{E} \quad \Gamma \vdash \mathsf{t} : \mathsf{C} \quad \mathsf{C} <: \mathsf{D} \quad \Gamma \vdash \overline{\mathsf{a} : \mathsf{H}} \quad \overline{\mathsf{H} <: \mathsf{G}} \end{array}}{\Gamma \vdash \mathsf{t}.@\mathsf{D}.\mathsf{m}(\overline{\mathsf{a}}) : \mathsf{F}} \qquad \text{(T-SuperRef)}$$

$$\frac{\begin{array}{c} \overline{\mathsf{x} : \mathsf{C}}, \mathsf{this} : \mathsf{C}_0, \mathsf{super} : \mathsf{D} \vdash \mathsf{t}_0 : \mathsf{E}_1 \quad \mathsf{E}_1 <: \mathsf{C}_1 \\ CT(\mathsf{C}_0) = \mathsf{class}\,\mathsf{C}\,\mathsf{extends}\,\mathsf{D}\{\ldots\} \\ override(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}} \to \mathsf{C}_1) \end{array}}{\Gamma \vdash \mathsf{C}_1\,\mathsf{m}(\overline{\mathsf{C}\,\mathsf{x}})\{\mathsf{return}\ \mathsf{t}_0;\}\ \mathsf{OK\ in}\ \mathsf{C}_0} \qquad \text{(Method Typing replaces rule from [39])}$$

Auxillary functions

$$\frac{\begin{array}{c} CT(\mathsf{C}_0) = \mathsf{class}\,\mathsf{C}_0\,\mathsf{extends}\,\mathsf{D}\{\overline{\mathsf{C}\,\mathsf{f}};\,\mathsf{K}\overline{\mathsf{M}}\} \\ \mathsf{B}\,m\,(\overline{\mathsf{B}\,\mathsf{x}})\{\mathsf{return}\,\mathsf{t};\} \in \mathsf{M} \end{array}}{mbody(\mathsf{m}, \mathsf{C}_0) = (\overline{\mathsf{x}}, [\mathsf{super} \mapsto \mathsf{this}.@\mathsf{D}]\,\mathsf{t})} \qquad \frac{\begin{array}{c} CT(\mathsf{D}) = \mathsf{class}\,\mathsf{D}\,\mathsf{extends}\,\mathsf{E}\{\overline{\mathsf{C}\,\mathsf{f}};\,\mathsf{K}\overline{\mathsf{M}}\} \\ \mathsf{C} <: \mathsf{D} \quad \mathsf{C} \neq \mathsf{D} \quad \mathsf{m} \in \overline{\mathsf{M}} \end{array}}{hasSuperImpl(\mathsf{C}, \mathsf{m})}$$

Evaluation

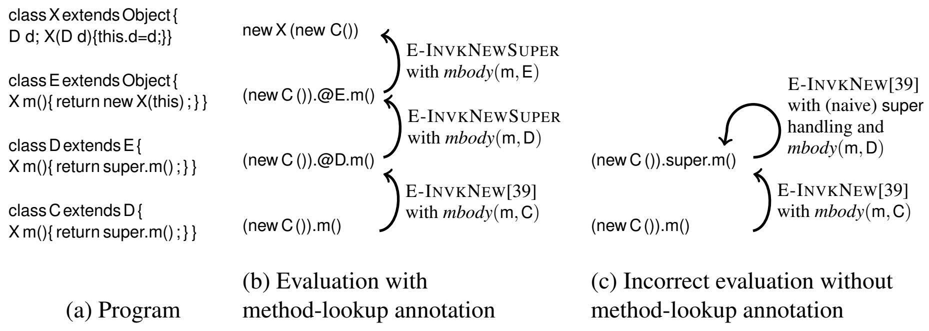
$$\frac{sat(\phi \wedge \Phi)}{(\phi \ ? \ \mathsf{t}_0 : \mathsf{t}_1) \to \mathsf{t}_0} \quad \text{(E-VarEnc-En)} \qquad \frac{mbody(\mathsf{m}, \mathsf{D}) = (\overline{\mathsf{x}}, \mathsf{t}_0)}{\begin{array}{c}(\mathsf{new}\,\mathsf{C}\,(\overline{\mathsf{v}})).@\mathsf{D}.\mathsf{m}(\overline{\mathsf{u}}) \to \\ [\overline{\mathsf{x} \mapsto \mathsf{u}}, \mathsf{this} \mapsto \mathsf{new}\,\mathsf{C}\,(\overline{\mathsf{v}})]\,\mathsf{t}_0\end{array}} \quad \text{(E-InvkNewSuper)}$$

$$\frac{sat(\neg\phi \wedge \Phi)}{(\phi \ ? \ \mathsf{t}_0 : \mathsf{t}_1) \to \mathsf{t}_1} \quad \text{(E-VarEnc-Dis)}$$

Figure 9: The typing rules, auxillary functions, and evaluation rules FJSIM adds to FJ; ϕ denotes a presence condition, Φ a configuration.

choices, as well as rules for typing the superclass lookup. The expression $\Gamma \vdash \mathsf{t} : \mathsf{C}$ denotes that the term t is of type C in context Γ, which maps bound variables to types. T-VarEnc enforces that the terms in the then and else branches of a feature choice have the same type. T-SuperRef enforces that one of the superclasses actually implements the method referred to in a call with lookup annotation. All other typing rules are identical to the rules of FJ [39] and omitted for brevity.

Evaluation. The lower part of Figure 9 shows the evaluation rules that FJSIM adds to FJ. The evaluation rule E-InvkNewSuper resolves references to superclasses ((new C(...)).@D) and searches for a method implementation in the superclass D. Keyword super itself is already handled earlier, during the call to the auxillary function *mbody*, which we redefined in Figure 9 compared to FJ. The redefined *mbody* function replaces super used in a class C with this.@D where D is the superclass of C. A method call (new C(...)).@D.m(...) executes the method m from class D on the object new C(...). Figure 10 shows an example evaluation of super and motivates why we need the @D notation to correctly evaluate super. In Figure 10b, the initial term (new C ()).m() is evaluated using the standard rule E-InvkNew [39] and *mbody*(m,C) = this.@D.m(). It substitutes this with new C() resulting in the term (new C ()).@D.m(), which is then eval-


```
class X extends Object{
D d; X(D d){this.d=d;}}

class E extends Object{
X m(){ return new X(this) ; } }

class D extends E{
X m(){ return super.m() ; } }

class C extends D{
X m(){ return super.m() ; } }
```
(a) Program
new X (new C())
E-INVKNEWSUPER with mbody(m, E)
(new C ()).@E.m()
E-INVKNEWSUPER with mbody(m, D)
(new C ()).@D.m()
E-INVKNEW[39] with mbody(m, C)
(new C ()).m()
(b) Evaluation with method-lookup annotation
E-INVKNEW[39] with (naive) super handling and mbody(m, D)
(new C ()).super.m()
E-INVKNEW[39] with mbody(m, C)
(new C ()).m()
(c) Incorrect evaluation without method-lookup annotation

Figure 10: An example of correct evaluation of super references in FJSIM (b) and incorrect evaluation in FJSIM without method-lookup annotations (c)

uated with E-INVKNEWSUPER; this in new X(this) is again substitued with new C(). The resulting term (new C ()).@E.m() is then evaluated to (new X ((new C ()))). If we would not insert the @D annotation, we could not know in which superclass to start searching for implementations of method m. In this case, we might select the implementation from D (the superclass of the this object) again and generate an (incorrect) endless loop (cf. Figure 10c).

The evaluation rules E-VARENC-EN and E-VARENC-DIS define how feature choices are evaluated. The evaluation of an FJSIM program is deterministic, as the configuration Φ used for evaluation of the feature choices has only one satisfying assignment. This way, in each step evaluating a feature choice, either E-VARENC-EN or E-VARENC-DIS is applied. As stated earlier, our model of load-time variability guarantees that each configuration has only one satisfying assignment, so no variability remains (Section 2.1). If we would not enforce this property, both evaluation rules might be applicable to the same term (the presence condition ϕ *and* its negation $\neg\phi$ would be satisfiable) and the program behavior would be non-deterministic. When variant simulators are analyzed, this non-determinism helps to explore identical execution paths from many variants simultaneously as we discuss in Section 6.

3.4. Generation of Variants and Variant Simulators

Generation of variants. Variant generation derives an FJ program from an CFJ program based on a given configuration Φ. Kästner et al. [28] formalized the generation of variants. In Figure 11 we show the relevant subset of these rules. Function *derive* takes a CFJ program $(CT, AT, MT, \text{init}, \hat{\Phi})$ and a valid configuration Φ. It returns a corresponding variant implemented in FJ (the program is also an FJSIM program without feature choices and super calls). It uses the auxillary functions $\preccurlyeq \cdot \succcurlyeq$, $\ll \cdot \gg$, and $[\![\cdot]\!]$. These functions transform a single CFJ syntax element (class, method, term) into an FJ syntax element. Function $\preccurlyeq \cdot \succcurlyeq$ traverses the elements of the program recursively and invokes the derivation of term variants. Function $\ll \cdot \gg$ chooses between alternative program elements according to the given configuration Φ by iterating alternatives defined in MT. For example, $\ll MT$ (printMulti(f,b)), printDuplex(f,b) $\gg$ selects between

$$\begin{array}{rcl} \mathit{derive} : (\text{CFJ program}, \text{Configuration}) & \rightarrow & \text{FJ program} \\ \mathit{derive}((\mathit{CT}, \mathit{AT}, \mathit{MT}, \mathsf{init}, \hat{\Phi}), \Phi) & = & (\preccurlyeq(\mathit{range}(\mathit{CT}), \mathsf{init})\succcurlyeq) \end{array}$$

$$\begin{array}{rcll} \preccurlyeq\succcurlyeq : \text{CFJ term} & \rightarrow & \text{FJ}\textsc{sim}\text{ term} & \\ \preccurlyeq\mathsf{v}\succcurlyeq & = & \mathsf{v} & \text{(G.1)} \\ \preccurlyeq\mathsf{t.f}\succcurlyeq & = & \preccurlyeq\mathsf{t}\succcurlyeq\mathsf{.f} & \text{(G.2)} \\ \preccurlyeq\mathsf{t.m}(\bar{\mathsf{t}})\succcurlyeq & = & \preccurlyeq\mathsf{t}\succcurlyeq\mathsf{.m}(\preccurlyeq\bar{\mathsf{t}}\succcurlyeq) & \text{(G.3)} \\ \preccurlyeq\mathsf{new\ C}(\bar{\mathsf{t}})\succcurlyeq & = & \mathsf{new\ C}(\preccurlyeq\bar{\mathsf{t}}\succcurlyeq) & \text{(G.4)} \\ \preccurlyeq\mathsf{C\ m}(\overline{\mathsf{C\ x}})\{\,\mathsf{return\ t;}\}\succcurlyeq & = & \mathsf{C\ m}(\overline{\mathsf{C\ x}})\{\,\mathsf{return}\ \ll \mathit{MT}(\mathsf{t}), \mathsf{t} \gg;\} & \text{(G.5)} \\ \preccurlyeq\mathsf{class\ C\ extends\ D}\ \{\,\overline{\mathsf{C\ f}};\ \mathsf{K}\ \overline{\mathsf{M}}\}\succcurlyeq & = & \mathsf{class\ C\ extends\ D}\ \{\,\overline{\mathsf{C\ f}};\ \mathsf{K}\preccurlyeq\llbracket\overline{\mathsf{M}}\rrbracket\succcurlyeq\} & \text{(G.6)} \\ \preccurlyeq(\bar{\mathsf{L}}, \mathsf{t})\succcurlyeq & = & (\preccurlyeq\bar{\mathsf{L}}\succcurlyeq, \mathsf{t}) & \text{(G.7)} \end{array}$$

$$\begin{array}{rcl} \ll\gg & : & \text{CFJ term} \times \text{CFJ term} \rightarrow \text{FJ}\textsc{sim}\text{ term} \\ \ll \mathsf{t}_1, \mathsf{t}_2 \gg & = & \begin{cases} \preccurlyeq\mathsf{t}_1\succcurlyeq & \mathsf{t}_1 \neq \bullet,\quad \mathit{sat}(\Phi \wedge \mathit{AT}(\mathsf{t}_1)) \\ \ll \mathit{MT}(\mathsf{t}_1), \mathsf{t}_2 \gg & \mathsf{t}_1 \neq \bullet,\quad \neg\mathit{sat}(\Phi \wedge \mathit{AT}(\mathsf{t}_1)) \\ \preccurlyeq\mathsf{t}_2\succcurlyeq & \mathsf{t}_1 = \bullet\,(\mathit{otherwise}) \end{cases} \end{array}$$

$$\begin{array}{rcl} \llbracket\rrbracket & : & \text{CFJ term} \rightarrow \text{FJ}\textsc{sim}\text{ term} \\ \llbracket\mathsf{a}\rrbracket & = & \begin{cases} \mathsf{a} & \mathit{sat}(\Phi \wedge \mathit{AT}(\mathsf{a})) \\ \bullet & \mathit{otherwise} \end{cases} \end{array}$$

Figure 11: Variant generation rules, adopted from Kästner et al. [28]; lists are processed element-wise, e.g., $\llbracket \mathsf{t}_1, \mathsf{t}_2, \ldots, \mathsf{t}_n \rrbracket = \llbracket \mathsf{t}_1 \rrbracket, \llbracket \mathsf{t}_2 \rrbracket, \ldots, \llbracket \mathsf{t}_n \rrbracket$; $\bullet$ denotes the empty program element and $\preccurlyeq\bullet\succcurlyeq = \bullet$; $\mathit{range}(\mathit{CT})$ denotes all class definitions in class table CT.

printMulti(f,b) and its alternative printDuplex(f,b) in Figure 7. Function $\llbracket\cdot\rrbracket$ removes program elements if their presence condition is not satisfied. For example, the method definition of printDuplex is eliminated in $\llbracket$void printDuplex(Page f, Page b) {...}$\rrbracket$ iff its presence condition in AT is not satisfied.

Variant-simulator generation. The generation of variant simulators is similar to the generation of variants. The main differences are that, the target language is FJSIM and that instead of removing optional program elements, we encode this variability by means of feature choices. For example, the expression printMulti(f,b) and its alternative printDuplex(f,b) in Figure 7 are encoded as $(\mathit{Duplex}\,?\,\mathsf{printDuplex(f,b)}\!:\!\mathsf{printMulti(f,b)})$ in FJSIM. Figure 12 shows the definition of function *encode*. Function *encode* generates a variant simulator in FJSIM for a given CFJ program $(\mathit{CT}, \mathit{AT}, \mathit{MT}, \mathsf{init}, \hat{\Phi})$. Function *encode* uses the auxillary functions $\preccurlyeq\cdot\succcurlyeq$ and $\ll\cdot\gg$, which we redefine in Figure 12. The figure also defines function $\llbracket\mathsf{t}\rrbracket_{\mathsf{C}}$ where C denotes the class containing the term t. The omitted cases of $\preccurlyeq\cdot\succcurlyeq$ are the same as in Figure 11. Function $\ll \mathsf{t}_1, \mathsf{t}_2 \gg$ iterates through all alternatives of term t_1, introducing feature choices. It uses the default term t_2 as innermost else case. All functions employ the variability model $\hat{\Phi}$ instead of a configuration Φ, so that variable parts are only dropped if their presence condition is not satisfiable in $\hat{\Phi}$. Function $\llbracket\cdot\rrbracket_{\mathsf{C}}$ handles optional methods. The function introduces a call to the same method in a superclass of C if there exists a variant in which the currently generated method is not present. In this case, a call to the current method will execute the superclass method in the variant. We model this behavior in the variant simulator using the keyword super.

$$\begin{array}{rcl} \mathit{encode} : \text{CFJ program} & \rightarrow & \text{FJSIM program} \\ \mathit{encode}(\mathit{CT}, \mathit{AT}, \mathit{MT}, \mathsf{init}, \hat{\Phi}) & = & (\preccurlyeq(\mathit{range}(\mathit{CT}), \mathsf{init})\succcurlyeq, \hat{\Phi}) \end{array}$$

$$\begin{array}{rcll} \preccurlyeq\succcurlyeq : \text{CFJ term} & \rightarrow & \text{FJSIM term} & \\ & \dots & & \\ \preccurlyeq\mathsf{class\ C\ extends\ D}\,\{\,\overline{\mathsf{C\ f}};\ \mathsf{K}\ \overline{\mathsf{M}}\}\succcurlyeq & = & \mathsf{class\ C\ extends\ D}\,\{\,\overline{\mathsf{C\ f}};\ \mathsf{K}\preccurlyeq[\![\overline{\mathsf{M}}]\!]_{\mathsf{C}}\succcurlyeq\} & \text{(G.6)} \\ & \dots & & \end{array}$$

$$\begin{array}{rcl} \ll\gg & : & \text{CFJ term} \times \text{CFJ term} \rightarrow \text{FJSIM term} \\ \ll \mathsf{t}_1, \mathsf{t}_2 \gg & = & \begin{cases} \mathit{AT}(\mathsf{t}_1)\,?\preccurlyeq\mathsf{t}_1\succcurlyeq : \ll \mathit{MT}(\mathsf{t}_1), \mathsf{t}_2 \gg & \mathsf{t}_1 \neq \bullet \wedge \mathit{sat}(\hat{\Phi} \wedge \mathit{AT}(\mathsf{t}_1)) \\ \ll \mathit{MT}(\mathsf{t}_1), \mathsf{t}_2 \gg & \mathsf{t}_1 \neq \bullet \wedge \neg\mathit{sat}(\hat{\Phi} \wedge \mathit{AT}(\mathsf{t}_1)) \\ \preccurlyeq\mathsf{t}_2\succcurlyeq & \mathsf{t}_1 = \bullet\,(\mathit{otherwise}) \end{cases} \end{array}$$

$$\begin{array}{rcl} [\![]\!]_{\mathsf{C}} & : & \text{CFJ method definition} \rightarrow \text{FJSIM method definition} \\ [\![\mathsf{a}]\!]_{\mathsf{C}} & = & \begin{cases} \mathsf{D\ m}(\overline{\mathsf{C\ x}})\,\{\,\mathsf{return}\,(\mathit{AT}\,(\mathsf{a})\,?\preccurlyeq\mathsf{t}\succcurlyeq\ :\mathsf{super.m}(\overline{\mathsf{C\ x}}));\} & \mathsf{a} = \mathsf{D\ m}(\overline{\mathsf{C\ x}})\,\{\,\mathsf{return\,t};\} \\ & \mathit{sat}(\hat{\Phi} \wedge \mathit{AT}(\mathsf{a}))\quad \mathit{hasSuperImpl}(\mathsf{C}, \mathsf{m}) \\ \mathsf{D\ m}(\overline{\mathsf{C\ x}})\,\{\,\mathsf{return\,super.m}(\overline{\mathsf{C\ x}});\} & \mathsf{a} = \mathsf{D\ m}(\overline{\mathsf{C\ x}})\,\{\,\mathsf{return\,t};\} \\ & \neg\mathit{sat}(\hat{\Phi} \wedge \mathit{AT}(\mathsf{a}))\quad \mathit{hasSuperImpl}(\mathsf{C}, \mathsf{m}) \\ \mathsf{D\ m}(\overline{\mathsf{C\ x}})\,\{\,\mathsf{return}\preccurlyeq\mathsf{t}\succcurlyeq;\} & \mathsf{a} = \mathsf{D\ m}(\overline{\mathsf{C\ x}})\,\{\,\mathsf{return\,t};\} \\ & \mathit{sat}(\hat{\Phi} \wedge \mathit{AT}(\mathsf{a}))\quad \neg\mathit{hasSuperImpl}(\mathsf{C}, \mathsf{m}) \\ \mathsf{a} & \mathit{otherwise} \end{cases} \end{array}$$

Figure 12: Variant-simulator generation rules. $\ll \cdot \gg$ introduces *feature choices*, if multiple terms are feasible. For brevity, we omit the propagation of *AT*, *MT*, and $\hat{\Phi}$.

4. Behavior Preservation

Many applications that use variability encoding depend on the fact that variability encoding preserves the behavior of the simulated variants. This includes, in particular, all control-flow sensitive applications such as verification [4, 5, 46] or testing [30]. In this section, we prove the behavior-preservation property for variability encoding based on our model of Section 3. As the key result of this article, our proof guarantees that a variant simulator can be used for behavioral analysis of the variants it simulates. It shows that the execution of a variant simulator and the corresponding variants exhibit the same observable behavior, as illustrated in Figure 4.

In particular, we prove that the behavior of each variant of a configurable system is *weakly bisimilar* to the variant simulator, if the variant simulator is executed with the variant's configuration. We use weak bisimulation [37] as proof technique to show that each execution trace in a variant is represented by a trace in the variant simulator with the configuration that corresponds to the variant. We use the *weak* form of bisimulation to allow the occurence of additional feature-choice transitions in the simulator. Next, we define a trace semantics for FJSIM programs in Section 4.1, which we use to prove behavior preservation in Section 4.2.

4.1. *A Trace Semantics for* FJSIM *Programs*

We introduce a trace semantics for FJSIM that encodes the run-time semantics defined by the evaluation rules (Figures 6 and 9). This way, the behavior of a program

$genTS : (\text{FJ}\textsc{sim}\ \text{CT}, \text{FJ}\textsc{sim}\ \text{term}) \to \text{Transition system}\,(\mathit{States}, \mathit{Transitions}, \mathit{Presence\ conditions})$

$$
genTS(CT,\mathsf{t}) = \left\{\begin{array}{ll}
\textsc{Termination case}\quad (\text{if } \mathsf{t} \text{ is a value}) & \\
(\{\mathsf{v}\}, \emptyset, \emptyset) & \mathsf{t} = \mathsf{v} \\
\hline
\text{TS-}\textsc{ProjNew} & \mathsf{t} = ((\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{f}_j) \text{ and} \\
(\{\mathsf{t}, \mathsf{v}_j\}, \{(\mathsf{t}, \mathit{true}, \mathsf{v}_j)\}, \{\mathit{true}\}) & \mathsf{f}_j \in \mathit{fields}(\mathsf{C}) \text{ and } v_j \in \overline{\mathsf{v}} \\
\hline
\text{TS-}\textsc{CastNew} & \\
(\{\mathsf{t}, \mathsf{t}'\}, \{(\mathsf{t}, \mathit{true}, \mathsf{t}')\}, \{\mathit{true}\}) \Cup genTS(CT, \mathsf{t}') & \mathsf{t} = (\mathsf{D})(\mathsf{new\ C}(\overline{\mathsf{v}})) \text{ and} \\
\text{ with } \mathsf{t}' = \mathsf{new\ C}(\overline{\mathsf{v}}) & \mathsf{C} <: \mathsf{D} \\
\hline
\text{TS-}\textsc{InvkNewSuper} & \\
(\{\mathsf{t}, \mathsf{t}'\}, \{(\mathsf{t}, \mathit{true}, \mathsf{t}')\}, \{\mathit{true}\}) \Cup genTS(CT, \mathsf{t}') & \mathsf{t} = (\mathsf{new\,C}\,(\overline{\mathsf{v}})).@\mathsf{D}.m(\overline{\mathsf{u}}) \text{ and} \\
\text{ with } \mathsf{t}' = [\overline{\mathsf{x} \mapsto \mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}(\overline{\mathsf{v}})]\,\mathsf{t}_0 & \mathit{mbody}(\mathsf{m}, \mathsf{D}) = (\overline{\mathsf{x}}, \mathsf{t}_0) \\
\hline
\text{TS-}\textsc{InvkNew} & \\
(\{\mathsf{t}, \mathsf{t}'\}, \{(\mathsf{t}, \mathit{true}, \mathsf{t}')\}, \{\mathit{true}\}) \Cup genTS(CT, \mathsf{t}') & \mathsf{t} = ((\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{u}})) \text{ and} \\
\text{ with } \mathsf{t}' = [\overline{\mathsf{x} \mapsto \mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}(\overline{\mathsf{v}})]\,\mathsf{t}_0 & \mathit{mbody}(\mathsf{m}, \mathsf{C}) = (\overline{\mathsf{x}}, \mathsf{t}_0) \\
\hline
\text{TS-}\textsc{VarEnc-En} \text{ and TS-}\textsc{VarEnc-Dis} & \\
(\{\mathsf{t}, \mathsf{t}_0\}, \{(\mathsf{t}, \phi, \mathsf{t}_0)\}, \{\phi\}) \Cup & \mathsf{t} = (\phi\ ?\, \mathsf{t}_0 : \mathsf{t}_1) \\
(\{\mathsf{t}, \mathsf{t}_1\}, \{(\mathsf{t}, \neg\phi, \mathsf{t}_1)\}, \{\neg\phi\}) \Cup & \\
genTS(CT, \mathsf{t}_0) \Cup genTS(CT, \mathsf{t}_1) & \\
\hline
\mathit{Cases\ for\ congruence\ rules\ are\ omitted} &
\end{array}\right.
$$

Figure 13: Definition of function *genTS* for the generation of a transition system $(\mathit{States}, \mathit{Transitions}, \mathit{Presence\ Conditions})$ from an FJSIM program. For legibility, we define the join operation $\Cup$ on transition systems (S, T, PC) and (S', T', PC') as follows: $(S, T, PC) \Cup (S', T', PC') = (S \cup S', T \cup T', PC \cup PC')$.

is represented as a transition system, which is better suited for our bisimulation proof than the source code representation. In particular, we model the run-time behavior of a variant π_Φ and a variant simulator σ. Using function $genTS(CT,\mathsf{t})$ of Figure 13, we define the transition system of a program $p = (\mathsf{CT}, \mathsf{init}, \Phi)$ as $genTS(\mathsf{CT}, \mathsf{init})$. The generated system is a labeled transition system (S, T, PC), where S is the set of states, T is the set of transitions ($T \subseteq S \times PC \times S$), and PC is the set of presence conditions. States in the transition system represent FJSIM terms, and transitions represent evaluation steps that rewrite one term into another. Presence conditions are propositional formuls over variables in the set of configuration options. Transitions are labeled with presence conditions that have to hold during evaluation in order to proceed with the respective evaluation step. A trace is a sequence of states of the transition system starting in the initial state (term init). Each trace represents an execution path in the corresponding FJSIM program. Function *genTS* recursively processes all terms of an FJSIM program, and adds states and transitions for each term to the transition system. The generation for transition systems for FJ programs (system variants) is defined analogous.

4.2. Proof of Behavior Preservation

We assume that all valid variants of a given CFJ program with code base Δ are well typed with respect to the variability model $\hat{\Phi}$ [28], as the behavior of ill-typed

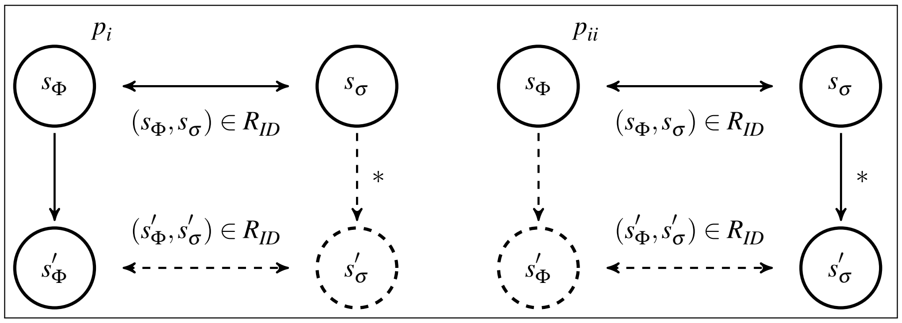

p_i
s_Φ
s_σ
$(s_\Phi, s_\sigma) \in R_{ID}$
$*$
$(s'_\Phi, s'_\sigma) \in R_{ID}$
s'_Φ
s'_σ
p_{ii}
s_Φ
s_σ
$(s_\Phi, s_\sigma) \in R_{ID}$
$*$
$(s'_\Phi, s'_\sigma) \in R_{ID}$
s'_Φ
s'_σ

Figure 14: The weak bisimulation property has two sub-properties p_i and p_{ii}. The sub-properties prove the existence of dashed relations and states assuming solid ones.

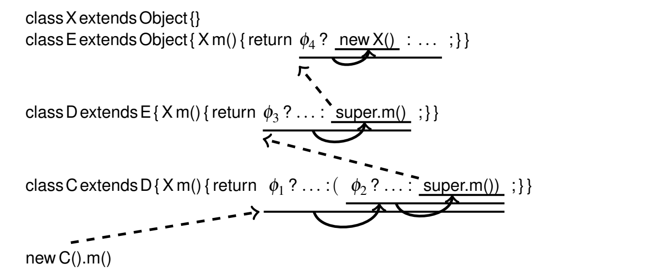
variants is undefined [39]. To prove that the behavior of all variants is preserved by the corresponding variant simulators (i.e., that variability encoding is sound), we create a variant π_Φ with function $derive(\Delta, \Phi)$ for every configuration Φ. The generated variant $\pi_\Phi = (CT_\Phi, \mathsf{init}, true)$ has a class table CT_Φ, which is constructed with the function *encode* of Figure 11 and a start term init. The corresponding variant simulator $\sigma = (CT_\sigma, \mathsf{init}, \hat{\Phi})$ is generated using function $encode(\Delta, \hat{\Phi})$ of Figure 11 and Figure 12. Hence, the variant simulator σ and the variant π_Φ are constructed from the same configurable code base Δ, and the execution of both programs, σ and π_Φ, starts with the term init.

We generate the transition systems for the variant π_Φ and the variant simulator σ as defined in Section 4.1. The transition system for variant π_Φ is denoted with $(S_\Phi, \rightarrow_\Phi, PC)$ and derived by $genTS(CT_\Phi, \mathsf{init})$, where S_Φ is the set of states of the system, PC is the set of labels (i.e. presence conditions) on the transitions, and $\rightarrow_\Phi$ is the set of transitions ($\rightarrow_\Phi \subseteq S_\Phi \times PC \times S_\Phi$).

The transition system for variant simulator σ is denoted with $(S_\sigma, \rightarrow_\sigma, PC)$ and derived by $genTS(CT_\sigma, \mathsf{init})$. When we execute the simulator σ with configuration Φ, we use the corresponding projection of the transitions from $\rightarrow_\sigma$: $\rightarrow_{\sigma|\Phi} = \{(s, pc, s') \in \rightarrow_\sigma | sat(\Phi \wedge AT(pc))\}$. This corresponds to execution of a simulator in a real execution environment that evaluates the presence conditions of feature choices with respect to a configuration. For clarity, we denote states from the variant transition system with s_Φ and s'_Φ, and states from the variant-simulator transition system with s_σ and s'_σ. If it is clear from the context, we omit the subscripts in the transition relations $\rightarrow_\Phi$ and $\rightarrow_{\sigma|\Phi}$.

Based on these definitions, Figure 14 illustrates the weak bisimulation property we want to prove [1]. R_{ID} is the *simulation relation*, which relates states of the variant transition system to states of the variant-simulator transition system. In our proof, we use the syntactic equality of terms in FJSIM as simulation relation. For two states s_Φ and s_σ, $(s_\Phi, s_\sigma) \in R_{ID}$ holds, iff the terms represented by s_Φ and s_σ are equal.

[1] We choose the *weak bisimulation* property over the often weaker *weak-trace-equivialence* property because both properties are equal in our case. Bisimulation and trace equivialence are different iff a state can have two equally-labeled outgoing edges leading to different states. Such behavior often occurs in the context of concurrent execution. In our transition system, an FJSIM term is always evaluated with a specific evaluation rule yielding exactly one term (assuming a configuration Φ).



class X extends Object{}
class E extends Object{ X m(){ return ϕ_4 ? new X() : ... ;}}
class D extends E{ X m(){ return ϕ_3 ? ... : super.m() ;}}
class C extends D{ X m(){ return ϕ_1 ? ... :(ϕ_2 ? ... : super.m()) ;}}
new C().m()

Figure 15: Proof concept in presence of overriding methods

This is possible because each FJ term is by definition also an FJSIM term. The weak bisimulation property has two sub-properties (p_i and p_{ii}), which must both be proved. Property p_i states that, for each direct successor state s'_Φ of s_Φ, there exists a state s'_σ in the variant-simulator transition system that is related to the state s'_Φ of the variant transition system with $(s'_\Phi, s'_\sigma) \in R_{ID}$, and that s'_σ is a successor state of s_σ. Property p_{ii} states that, for each successor state s'_σ of s_σ, there exists a state s'_Φ in the variant-simulator transition system that is related to the state s'_σ of the variant transition system, with $(s'_\Phi, s'_\sigma) \in R_{ID}$, and that s'_Φ is a successor state of s_Φ. In *weak* bisimulation, s'_σ does not need to be a *direct* successor to s_σ; in our case there may be a number of auxillary states in between that evaluate feature choices. We denote such a sequence of states linked by consecutive transitions with $s_\sigma \overset{*}{\rightarrow} s'_\sigma$. A $s_\sigma \overset{*}{\rightarrow} s'_\sigma$ sequence starts with an evaluation rule from normal FJ and continues with zero or more applications of the new FJSIM evaluation rules (E-VARENC-EN, E-VARENC-DIS, or E-INVKNEWSUPER).

A particularly interesting part of the proof is how we prove correctness in the presence of overriding methods. We moved a corresponding part of the proof to Lemma 1, to simplify understanding. Figure 15 shows an example for how the lemma is used. It shows four classes of a variant simulator and a term (new C()).m() that is evaluated. The classes C, D, and E implement method m. The term (new C()).m() is evaluated as shown in the figure if the presence conditions ϕ_1, ϕ_2, and ϕ_3 are not satisfiable, and ϕ_4 is satisfiable in a configuration Φ. In this case, term (new C()).m() evaluates to term new X(), defined in E. As a consequence, the transition system of the variant simulator must contain a path from (new C()).m() to new X(), which may contain auxillary states. We prove the existence of this path in two steps. First, Lemma 1 proves that the intra-method dispatch among alternative implementations is resolved correctly (solid arrows in Figure 15). Second, Case 3 in the proof of Theorem 1 shows that the steps from overriding to overridden methods are evaluated correctly (dashed arrows in Figure 15).

Lemma 1. *Given (1) a configuration Φ, (2) the method body* return t; *of a method* m($\bar{\text{x}}$) *in class* C*, and (3) that a call* (new C($\bar{\text{v}}$)).m($\bar{\text{u}}$) *is type correct in Φ, there exists a chain of consecutive states* t$\overset{*}{\rightarrow}\dots$ *that either evaluate* t (i) *to a term* t_Φ *that has a presence condition satisfied by Φ and is one of* m*'s alternative implementations in* C*, or* (ii) *to* (new C($\bar{\text{v}}$)).*@D.m*($\bar{\text{u}}$) *if no such term* t_Φ *exists and a superclass of* C *implements* m*. This evaluation sequence uses only the rules* TS-VARENC-EN *and* TS-VARENC-DIS.

Proof. We use induction over the number of feature choices n in term t to prove Lemma 1. The induction hypothesis is that each subterm of t evaluates to t_Φ (Case i) or to $(\mathsf{new\,C}(\bar{\mathsf{v}}))@D.m(\bar{\mathsf{u}})$ (Case ii) if it is annotated with a presence condition satisfied in Φ and has n or less feature choices . In both cases, the resulting term does not contain the keyword super. In Case (i), the term is a part of the CFJ program and as such cannot use super. In Case (ii), super has been substituted with a super reference (this.@..) when the method body was loaded with *mbody*.

There are two base cases ($n = 0$) which correspond to cases (i) and (ii) in Lemma 1. In Base Case (i), t is one of the alternative implementations of m. In this case, the presence condition $AT(\mathsf{t})$ must be satisfied by Φ, otherwise a call to m is not well typed. In Base Case (ii), t is an invocation of m in the direct superclass of C. Therefore, t equals $(\mathsf{new\,C}(\bar{\mathsf{v}}))@\mathsf{D.m}(\bar{\mathsf{u}})$, and there exists an implementation of m in some superclass of C, because the call is well typed. The base cases are exclusive; a given term can satisfy either (i) or (ii).

In the inductive step $n \to n+1$, t_1' is a feature choice $\mathsf{t}_1' = (\phi\,?\,\mathsf{t}_\phi : \mathsf{t}_{\neg\phi})$. The subterms t_ϕ and $\mathsf{t}_{\neg\phi}$ have n or less feature-choice terms. From the definition of *genTS* (Figure 13, TS-VarEnc-En and TS-VarEnc-Dis), we know that the variant-simulator transition system contains the transitions *tr* from t_1' to t_ϕ and tr' from t_1' to $\mathsf{t}_{\neg\phi}$ (with the presence conditions ϕ and $\neg\phi$, respectively). Either ϕ is satisfied by configuration Φ (i.e., $sat(\phi \wedge \Phi)$) or the negation $\neg\phi$ is satisfied by Φ. Exactly one of ϕ or $\neg\phi$ is satisfied, because Φ is a configuration, and as such has a fixed value for each configuration option. The code that is invoked if ϕ is satisfied is encoded in t_ϕ and the code that is invoked if $\neg\phi$ is satisfied is encoded in $\mathsf{t}_{\neg\phi}$ (function $\ll \cdot \gg$, Figure 12). First, we consider the case where ϕ is satisfied under configuration Φ. Our version of CFJ enforces that feature choices can only occur as the outermost terms in return statements (alternatives are only allowed for method bodies, cf. Section 3.2). Thus, if the result of this evaluation step t_ϕ is not a feature choice, it does not contain any further variability and one of the base cases applies. Otherwise, the resulting term t_ϕ is a feature choice. This means, t_ϕ has the same syntactic form as t_1' in the beginning of the induction step and it is shorter than t_1' (has one feature choice less). Therefore, we can apply the induction hypothesis. If $\neg\phi$ is satisfied under configuration Φ the proof is analogous. In each step of the evaluation E-VarEnc-En or E-VarEnc-Dis is applied.

The induction shows that there is a sequence of consecutive states $(\mathsf{t}, \mathsf{t}_2, \ldots, \mathsf{t}_n)$ in the variant-simulator transition system that evaluate t to a non-feature-choice term t_n with t_n either being a term with a satisfied presence condition (Case (i)) or a call to an implementation of m in a superclass (Case (ii)). The chain is finite because the term becomes smaller in each iteration as long as the evaluated term is a feature choice. Therefore, the induction hypothesis and the lemma holds. □

Theorem 1. *Given a* CFJ *code base* Δ *with a variability model* $\hat{\Phi}$ *and a start term* init*, a configuration* Φ*, the simulation relation* R_{ID} *(term equality), a variant* $\pi_\Phi = (CT_\Phi, \mathit{init}, true) = derive(\Delta, \Phi)$*, a variant simulator* $\sigma = (CT_\sigma, \mathit{init}, \hat{\Phi}) = encode(\Delta, \hat{\Phi})$*, which is executed with* Φ*, and the corresponding transition systems* $(S_\Phi, \to, PC_\Phi) = genTS(CT_\Phi, \mathit{init})$ *and* $(S_\sigma, \to, PC_\sigma) = genTS(CT_\sigma, \mathit{init})$*, then the weak bisimulation property holds:*

$$\begin{array}{lll} & \forall s_\Phi \in S_\Phi, \forall s_\sigma \in S_\sigma \textit{ with } (s_\Phi, s_\sigma) \in R_{ID}: & \\ (p_i) & \forall s'_\Phi \textit{ with } s_\Phi \rightarrow s'_\Phi: & \exists s'_\sigma \in S_\sigma \textit{ such that } (s_\sigma \xrightarrow{*} s'_\sigma \textit{ and } (s'_\Phi, s'_\sigma) \in R_{ID}) \textit{ and} \\ (p_{ii}) & \forall s'_\sigma \textit{ with } s_\sigma \xrightarrow{*} s'_\sigma: & \exists s'_\Phi \in S_\Phi \textit{ such that } (s_\Phi \rightarrow s'_\Phi \textit{ and } (s'_\Phi, s'_\sigma) \in R_{ID}) \end{array}$$

Proof. We prove the two properties of bisimulation (p_i and p_{ii}) seperately.

Property $\mathbf{p}_i$. We prove p_i with a case distinction over the construction rules used to generate the transition ($s_\Phi \rightarrow s'_\Phi$), according to the definition of the transition systems (Figure 13). Overall, there are eleven cases. However, we omit cases handling congruence rules and focus on the cases of the six evaluation rules shown in Figures 6 and 9. The omitted cases are very similar to Case 1 shown below. For a complete proof, we refer to the supplementary material.

Case 1 (TS-PROJNEW)*:* As the transition ($s_\Phi, \textit{true}, s'_\Phi$) has been generated with TS-PROJNEW (Figure 13), f_j is a field in class C of variant π_Φ ($\mathsf{f}_j \in \textit{fields}(\pi_\Phi, \mathsf{C})$). Thus, there is a valid configuration Φ, in which the field is present in the program. The variant simulator generation rules, in particular G.2 (Figure 11 and Figure 12), ensure that the field is also present in the variant simulator: $\mathsf{f}_j \in \textit{fields}(\sigma, \mathsf{C})$. The definition of *genTS* (Figure 13, TS-PROJNEW) ensures that there is a transition ($s_\sigma \rightarrow s'_\sigma$) with $s'_\sigma = \mathsf{f}_j$. Therefore, s'_Φ and s'_σ represent the same terms and $(s'_\Phi, s'_\sigma) \in R_{ID}$ holds.

Case 3 (TS-INVKNEW)*:* Let $s_\Phi = (\mathsf{new\ C(\bar{v})).m(\bar{u})}$. Method m in π_Φ must have been generated with variant generation rule G.5 of Figure 11. It is important to note that $\bar{\mathsf{v}}$ and $\bar{\mathsf{u}}$ are lists of values (values cannot be evaluated any further). Let t_0 be the body of method m in the variant π_Φ. As the method body is included in π_Φ with Rule G.5, we can conclude that the configuration Φ implies the presence condition $AT(\mathsf{t}_0)$ and $s'_\Phi = \mathsf{t}_0$. As s_Φ is in simulation relation to s_σ, we know that $s_\sigma = (\mathsf{new\ C(\bar{v})).m(\bar{u})}$. From the definition of *genTS* (Figure 13, TS-INVKNEW), we infer (1) that $\rightarrow_{\sigma|\Phi}$ contains a transition $tr_1 = (s_\sigma \rightarrow ([\overline{\mathsf{x} \mapsto \mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C(\bar{v})}]\mathsf{t}'_1))$ with $\textit{mbody}(\sigma, \mathsf{m}, \mathsf{C}) = \mathsf{t}'_1$ and (2) that tr_1 has the presence condition *true*.

We use induction over the number of superclasses of C in simulator σ to prove that s_Φ and s_σ evaluate to the same term. In the base case, class C is a direct subclass of Object. Object does not implement any methods [39]. Because each variant is well typed, term s_Φ is also well typed, and we can apply Lemma 1 with configuration Φ. The application of Lemma 1 shows that there exists a list of consecutive states $(s_\sigma, \mathsf{t}_2, \ldots, \mathsf{t}_n)$ in the variant-simulator transition system that evaluate s_σ to either a term t_n with a presence condition satisfied by Φ or to a call of m in a superclass. As C does not have superclasses (other than Object), we know that s_σ evaluates to t_n with $\mathsf{t}_n = s'_\sigma$ and $(s'_\Phi, s'_\sigma) \in R_{ID}$. The evaluation step $s_\sigma \rightarrow \mathsf{t}_2$ applies evaluation rule E-INVKNEWSUPER. All subsequent evaluation steps, until t_n is reached, apply rules TS-VARENC-EN or TS-VARENC-DIS which concludes the base case of the induction.

In the inductive step, C has $n+1$ superclasses, including Object. The term s_Φ is an invocation of method m on class C and the invocation is well typed in variant π_Φ, because either C has an implementation of m in the variant π_Φ or the method invocation is dispatched to an implementation of m in a superclass. If C itself has an implementation, Lemma 1 shows that there is a chain of states in σ from s_σ to s'_σ. As s'_σ is the first alternative implementation of m that satisfies the configuration Φ, s'_σ is in simulation relation to s'_{π_Φ}. If C does not have an implementation of m in the variant π_Φ, Lemma 1 shows that there is a chain of states in simulator σ from s_σ to (new C($\overline{\text{v}}$))@D.m($\overline{\text{u}}$). This chain uses only TS-VARENC-DIS. This new expression invokes method m in the direct superclass D of C. Because $\overline{\text{v}}$ and $\overline{\text{u}}$ are values, the next evaluation rule must be TS-INVKNEWSUPER. D has n superclasses, so we can apply the induction hypothesis, which states that the call of m on D evaluates to the same term as s_{π_Φ} in variant π_Φ.

Therefore, $s'_\Phi = \text{t}_0$, $s'_\sigma = \text{t}_n$, and $(s'_\Phi, s'_\sigma) \in R_{ID}$. So, there exists a state $s'_\sigma \in S_\sigma$, such that $(s'_\Phi, s'_\sigma) \in R_{ID}$ and a sequence of transitions $(s_\sigma \overset{*}{\rightarrow} s'_\sigma)$. The sequence starts with TS-INVKNEW followed by applications of TS-VARENC-EN, TS-VARENC-DIS, and TS-INVKNEWSUPER, which concludes Case 3.

Case 4 and Case 5 (TS-VARENC-EN *and* TS-VARENC-DIS)*:* None of the rules for variant generation (Figure 11, G.1–G.8) can generate a feature choice. Therefore, the variant π_Φ cannot contain terms to which rules TS-VARENC-EN and TS-VARENC-DIS may apply. Therefore, Case 4 and Case 5 cannot occur.

*Case 6 (*TS-INVKNEWSUPER*):* None of the rules for variant generation (Figure 11, G.1–G.8) can generate a super term. Therefore, the variant π_Φ cannot contain terms to which rule TS-INVKNEWSUPER applies and thus Case 6 cannot occur. This concludes the proof of property p_i of the bisimulation property.

Property p$_{ii}$. To prove the second sub-property of bisimulation (property p_{ii}), we have to do a similar case distinction as for the first property (p_i). Case 1 ((TS-PROJNEW) can be proven analogous to p_i. We focus on Cases 4, 5, and 6. All other cases can be proven analogous to the the cases for p_i given in the supplementary material.

Case 3 (TS-INVKNEW) is interesting because we have to show that the simulator can not evaluate to any states that are not present in a variant, except for states with ternary operators or the @D annotation. Given a term $s_\Phi =$ (new C($\overline{\text{v}}$)).m($\overline{\text{u}}$) with $(s_\Phi, s_\sigma) \in R_{ID}$ and a state s'_σ with $(s_\sigma \overset{*}{\rightarrow} s'_\sigma)$, we must show that the variant contains a state s'_Φ with $(s'_\Phi, s'_\sigma) \in R_{ID}$. Interestingly, s'_σ is the *only* state in the sequence $(s_\sigma \overset{*}{\rightarrow} s'_\sigma)$ that *can* be present in S_Φ. All intermediary states between s_σ and s'_σ are evaluated with the rules TS-VARENC-EN, TS-VARENC-DIS, or TS-INVKNEWSUPER and, therefore, contain ternary operators or the @D annotation, which can not occur in a variant. As a result, evaluation in the simulator deviates exactly as far from the variant evaluation as necessary to simulate the correct variant behavior. s_σ evaluates to s'_σ assuming configuration ϕ, and the argumentation of Case 3 in the proof of property p_i shows that s'_σ is equal to the state that s_Φ evaluates to in the variant. Therefore, variant π_Φ has a state s'_Φ with $(s'_\Phi, s'_\sigma) \in R_{ID}$ and a transition $(s_\Phi \rightarrow s'_\Phi)$, which concludes Case 3.

Cases 4, 5 and 6 (TS-VARENC-EN, TS-VARENC-DIS, and TS-INVKNEWSUPER) are not relevant, as the given states s_Φ and s_σ are in simulation relation R_{ID} and s_Φ cannot contain ternary operators or the @D annotation. This concludes property p_{ii} and the bisimulation proof.

□

5. Variability Encoding Beyond Featherweight Java

For the purpose of developing a formal proof, we limited our model of variability encoding as well as the considered languages to a small subset of JAVA. However, variability encoding is meant to be performed on real programs written in languages such as JAVA or C. Even for such complex languages it is always possible to build variant simulators trivially using duplication. One can just build every valid program variant and, at program start, dispatch between these variants. However, analysis of such variant simulators is inefficient because similarities among variants cannot be exploited (see Section 6). So, the question raises how additional language concepts of JAVA and C affect our model (which enables sharing among variants) and our proof of behavior preservation of variant simulators.

As modelling the full power of JAVA or C is elusive, we discuss a selection of interesting language constructs that are problematic for variability encoding, and we discuss how to deal with them in concrete examples. We express the variability in the examples using #if directives, because they facilitate a compact notation with fine-grained variability. In most examples, we have to resort to local code duplication as a workaround to solve the variability-encoding problems. This results in code fragments that are not shared among variants any more. However, in our experience with real applications, the blowup introduced by these duplications can be kept locally and the majority of the code is still shared, as we discuss in Section 6. In some cases the variability-encoding problems and solutions may seem trivial, but it is important to discuss these basic situations before attempting to implement variability encoding in a more complex language. The overarching goal is still that a variant simulator weakly bisimulates all variants. That is, it executes *only* feature switches and statements that are equivalent to statements of the variant that is currently simulated.

Method overloading. JAVA allows programmers to define multiple methods with the same name in a class hierarchy iff the signatures of the methods differ. Such methods can even have the same number of parameters, iff the parameter types are different. Method overloading is not supported in FJ, and thus neither in CFJ and FJSIM. Adding method overloading to CFJ leads to two problems for variability encoding, which we discuss separately in the following paragraphs:

1. In combination with inheritance, methods may overload methods defined in other classes of the inheritance hierarchy. If the overloading method is optional, a simulator generated with the rules of Section 3.4 could execute the wrong method.

2. Alternative methods in one class can have identical signatures in the simulator if we allow annotation of single parameters with presence conditions. If this situation is not handled, variability encoding can generate simulators that are not well typed.

Figure 16a shows a situation where method overloading and inheritance is combined. Class Y implements method m(B) and inherits an implementation of m(A) from class X. This is a case of overloading because both methods are callable on objects of type Y. The JAVA run-time environment decides which method to choose depending on the type of the given parameter. The given situation is even more complex because the method implemented in Y is optional and B is a subclass of A. This means that (new Y()).m(new B()) resolves to the implementation of m in Y if Opt1 is satisfiable and to the implementation in X otherwise. We have to model this behavior when designing the variant simulator. One crude approach would be to apply method renaming and

```
class A {}
class B extends A {}
class X { A m(A a) {...} }
class Y extends X {
  #if (Opt1)
  A m(B b) {return new A();}
  #endif
}

(new Y()).m(new B());
```

(a) Problem: Optional method overloading with inheritance

```
class A {}
class B extends A {}
class X { A m(A a) {...} }
class Y extends X {
  A m(B b) {
    return Opt1 ? new A() : super.m(b);
  }
}
```

(b) Solution: Introduce super also for overloaded methods.

Figure 16: Optional method overloading with inheritance in JAVA

```
1   class A {
2     int x = 0;
3     #if (Opt1 && Opt2)
4     int m() {return 0;}
5     #endif
6     int m(
7       #if (Opt1)
8       int x
9       #endif
10    ) { return x+1; }
11  }
```

(a) Problem: Method overloading with optional parameters

```
class A {
  int x=0;
  int m() { return Opt1 && Opt2 ? 0 :
            (Opt1 ? x+1 : intErr()); }
  int m (int x) {
    return Opt1 ? x+1 : intErr();
  }
  int intErr() { throw new Error(); }
}
```

(b) Solution: Duplication of the expression Opt1? x+1 : intErr()

Figure 17: Method overloading with optional parameters in JAVA

generate feature switches at all call sites. However, using super, we have a more elegant solution for this example. Figure 16b shows a simulator where we alternatively execute the code from Y or use super to call the method from X. This solution can be implemented in our formalism by altering the premise $\mathsf{m} \in \overline{\mathsf{M}}$ in the *hasSuperImpl*(C, m) rule from Figure 9 such that it also considers methods that overload m.

If we allow overloading and optional parameters, as shown in Figure 17a (Line 8), signatures are not unique any more, even within a single class. The code in Figure 17a contains two implementations of method m. The first method implementation (Line 4) is present only if the options Opt1 and Opt2 are selected. The second method implementation (Lines 6–10) is always present, but has two alternative variants. In one of the variants, an optional parameter is present; in the other variant, the parameter is missing. So, there is a variant of the second method definition in which it has the same signature as the first method. All configurations of the configurable program are well typed, and all variants of the method are used in some configurations, so we must include both methods in the variant simulator. However, as the methods have the same signature, we have a signature conflict if we just copy them to the simulator. To solve this signature conflict, we *duplicate* the code of the second method and insert it into both methods where it might be needed at run time (Figure 17b). Both locations are guarded with corresponding presence conditions. Although this solution leads to duplicated code fragments, we avoid modification of all call sites of the methods. Alternatively, we

```
struct str {
  int x;
#if (Opt1)
  int y;
#endif
} str;
int f() {
  return sizeof (str);
}
```

(a) Problem: Run-time system

```
struct str_noOpt1 {
  int x;
} str_noOpt1;
struct str_Opt1 {
  int x;
  int y;
} str_Opt1;
int f() {
  return Opt1 ? sizeof (str_Opt1) : sizeof (str_noOpt1);
}
```

(b) Solution: Duplication of the struct str

Figure 18: Interaction of variability encoding and environment functions in C

could introduce new methods encapsulating these fragments and rewrite all call sites.

Alternative types. We do not allow static variability of return types of functions or types of variables or parameters. However, this is possible in languages such as JAVA or C, with **#if** annotations. We can, for example, declare a variable with alternative types, such as **#if** (Opt1) **int #else double #endif** x;. The variable x is either of type integer or of type double. Depending on how the variable is used in the program, all valid variants of the configurable program can be well typed [2].

When building a variant simulator, we cannot statically determine the type of the variable. As the different types may not have a common supertype, we have to include both possibilities in the variant simulator. Therefore, we have to duplicate the variable declaration. Each location, at which the variable is used, must be modified such that the used variable variant depends on the selected configuration option. In extreme cases, if each variant has a different type for the variable, we have to introduce a variable for each configuration. However, according to our experience, this situation is very unlikely. Similar problems occur at other program locations where signatures are variable (for example optional modifiers) and no conditional statements are allowed. Examples are struct or enum definitions in C and generics, annotations, exceptions in method declarations, and class declarations (e.g., variable inheritance with alternative extends clauses) in JAVA.

Optional program variables. We do not support optional fields in our model. However, program variables (fields and local variables) may be optional in real-world applications.

Figure 18a shows a program that contains a struct with one or two integer variables, depending on the configuration. We need to include both variables in the variant simulator, because both are used in at least one configuration. The program also contains a function that returns the size of this struct. In one configuration (where Opt1 is not selected), the function returns the size of one integer; in the other configuration (where Opt1 is selected), it returns the size of two integers. However, as we have to include both integers in the variant simulator, the function always returns the size of two integers and, therefore the simulator may not preserve the behavior in some variants.

There are two possible solutions of this problem. One solution (shown in Figure 18b) is to rename and duplicate the struct definition. We also have to modify

```
int m() {
    int x = 0;
    for (int i = 0; i < 10; i++) {
      #if Opt1
        int x = 1;
      #endif
      x++;
    }
    return x;
}
```

(a) Problem: Field shadowing

```
int m() {
    int x = 0;
    for (int i = 0; i < 10; i++) {
        int y = 1;
        (Opt1 ? y++ : x++);
    }
    return x;
}
```

(b) Solution: Variable renaming

Figure 19: Optional field shadowing in C

references to the struct and add feature choices, such that the use always refers to the correct struct implementation. For more complex instances of the problem, this leads to an exponential blowup and complex variant simulators. An alternative solution is to transform all expressions that are affected by system functions that cannot be changed (such as sizeof). The transformed sizeof expression for our example would be (Opt1 ? 2∗sizeof(**int**) : sizeof(**int**)). In complex scenarios, the transformed expression may grow exponentially, similar to the previous solution. Both presented solutions (struct duplication and transformation of struct usages) have limitations in practice have to be applied depending on the case at hand. The same problem occurs with direct memory access in C via pointer arithmetic and with reflection in JAVA. In both cases, either a case-specific solution has to be found or code has to be duplicated.

Field shadowing. A further problem with optional program variables is shadowing [46]. As an example, assume that we define two local variables in different scopes with the same name, but one of the variables is optional depending on configuration option Opt1 (Figure 19a). Consequently, Opt1 influences to which program variable an identifier in the inner scope refers to. In our example, method m either returns 0 or 11. Hence, optional shadowing needs to be handled in variability encoding; one solution is to rename one of the program variables and duplicate all statements that contain the identifier (see Figure 19b). Shadowing may also co-occur with other language constructs such as inner classes. Instead of renaming, we can consider all these cases (other than variable shadowing) as code smells and suggest to forbid them, because code with optional shadowing may be hard-to-understand and may cause faults in configurable programs anyway.

Concurrency. In our model of variability encoding, we ensure that each statement and therefore each access to potentially shared data of the configurable program is either guarded with presence conditions or duplicated. There are two problems that occur in variability encoding of concurrent programs: (1) code executed during class initialization can sometimes not be enclosed in if statements (e.g., field initializations) and (2) feature-choice statements might slow down threads with much variable code more than others. To handle the first problem, we move code that initializes optional data structures (e.g., fields) to constructors and guard them with presence conditions there. This way, only code of one variant (and feature choices) is executed and other variants cannot interfere. The configuration of all feature variables is selected and fixed from the beginning of the execution of the variant simulator. All threads are executed with the same configuration and execute the behavior of the variant, including possible interactions between the threads. These interactions also include synchronizations on shared variables. The second problem can be solved by using an execution engine that explores all possible thread interleavings such as a model checker for concurrent programs. If multiple variants in a variant simulator with concurrency are analyzed at the same time (e.g., with a model checker), the analysis tool has to ensure that program states from different variants do not get mixed up and produce formerly unreachable states. Other common pitfalls of concurrent programs, such as breaking atomicity, shared mutable states, transactions, livelocks, and deadlocks, can be dismissed because a thread execution path in a simulator accesses the same variables (in the same order) as the corresponding path in the variant plus immutable feature variables.

Non-functional properties. Concerning non-functional properties (e.g., performance, memory consumption, or response time), a variant simulator behaves differently than a variant. The implementation of variability encoding ensures that the variant simulator executes statements that would be executed in the variant and additionally executes guard statements. Therefore, in theory, the only difference in executed statements should be the evaluation of presence conditions and the loading of classes which do not occur in the variant but are necessary in the simulator. Code for the instantiation of such classes can also be guarded. Depending on the system and its granularity, there may be many feature choices and the evaluation of presence conditions may also be expensive. The code overhead of variant simulators naturally influences aspects such as binary size and time for program setup. In the end, one has to take such differences into account when analyzing the non-functional performance of the variant simulator [44].

6. Experience with Variability Encoding

In recent years, variability encoding has been used in several projects for analyzing configurable systems. In this section, we give an overview of our attempts in this direction, and we summarize our experience with implementing variability encoding for real languages and systems[2]. Our formal proof of behavior preservation strengthens the results of these projects. It raises confidence that the implementations of variability encoding used in these projects also preserve behavior (even though the implementations are much more complex).

Variability encoding in JAVA. Variability encoding has been used for testing, verification, and performance modeling of configurable JAVA programs.

First, variability encoding was used for model checking configurable systems by Apel et al. [4, 5]. For the experiments, we selected three configurable JAVA programs that served as benchmarks for the community before and that provide functional specifications violated by some variants. To identify the violations, we constructed a variant

[2]When referring to "our" work, this means that at least one of the authors was involved in the respective project.

simulator per configurable program using FEATUREHOUSE [3] and verified the variant simulator using the software model checker JAVA PATHFINDER. The experiments showed that analyzing the variant simulator is substantially faster than checking all variants individually [5]. Also, analyzing the variant simulator was as precise for correctness checking as analysis of all variants.

Second, for testing of configurable systems, Kästner et al. [30] built an interpreter for JAVA-based systems with #ifdef variability on top of the tool TYPECHEF [29]. The idea is to parse the #ifdef variability as if it was run-time variability (similar to variant simulators). Then the program is executed with a fixed test input, but without fixing the feature variables; once a feature choice is reached, the interpreter executes both branches. The interpreter aims at visiting execution paths as few times as possible, still covering the executions of all variants, which can reduce testing time substantially.

Third, in a work on deductive verification of configurable programs, Thüm et al. [46, 47] applied variability encoding not only to the configurable JAVA code, but also to the corresponding specifications, which have been written in an extension of the JAVA MODELING LANGUAGE (JML). Much like for programs, a configurable JML specification may give rise to different variants, depending on the configuration, which needs to be taken into account during verification. Technically, we transform a configurable JML specification to a corresponding *specification simulator* (a.k.a. meta-specification [46]), which is similar to creating a variant simulator for a configurable JAVA program. Verifying the variant simulator using the theorem prover KEY, we observed considerable speedups compared to the verification of all variants, and the resulting proofs for the variant simulator have a similar complexity as the proofs for variants [46].

Fourth, an interesting property of variant simulators is that existing verifiers for run-time variability can be reused as-is for compile-time variability. We exploited this property by verifying compile-time configurable programs with the theorem-prover KEY and the software model checker JAVA PATHFINDER. We found that the combination imporves efficiency and effectiveness at the same time [47].

Finally, Siegmund et al. [44] used variability encoding to quantify the effects of individual features on the run time of the variants of a configurable program. Using FEATUREHOUSE, the authors constructed variant simulators for a set of configurable JAVA programs, with (non-variable) test cases. Based on these tests, they executed the variant simulators with a normal JAVA run-time environment, and they measured how much time was spent in each method and in which context the method was called. Using this information, they built a performance model per configurable program that allowed them to estimate the performance contribution of individual features and feature interactions, without creating and measuring each program variant individually.

In all of these approaches, variability encoding plays a central role in reducing analysis or measurement effort. As in all cases variability has been implemented with feature modules, most of the variability stems from method refinements resulting in alternative method bodies, which are easy to encode in variant simulators. Notably, none of the problems discussed in Section 5 occurred in the respective case studies, which suggests that coarse-grained variability mechanisms, such as feature modules, sufficiently facilitate variability encoding.

Variability Encoding in C. In a parallel line of research, we extended the tool TYPE-CHEF with functionality for variability encoding of large-scale configurable C programs that use the C preprocessor to implement compile-time variability. Due to the ability to express variability at a very fine grain, the process of constructing variant simulators for C programs with preprocessor directives is much more challenging than for feature modules in JAVA. Preprocessor-induced variability in C programs occurs frequently at the level of statements and expressions [32, 33]. Since variability at this granularity level cannot be expressed in C directly, one has to duplicate statements and expressions and embed them into conditional statements (i.e., feature choices), as we have illustrated in various examples in Section 5. As a further challenge, C does not support method overloading, so a variant simulator for a C program, similar to the example in Figure 17a, requires more code duplication to represent all configurations !Opt1, Opt1 && !Opt2, and Opt1 && Opt2. As a result, one has to introduce new method names for each duplicate and modify all calls to the method correspondingly. Furthermore, since the C preprocessor is often used to implement portable code (e.g., for different hardware architectures or operating systems), #if directives often implement variability at the level of type definitions (e.g., choosing between types u32 and u64). Similar to the handling of alternative method types (Section 5), one has to create multiple variants of type definitions that are included in different configurations. All other variability patterns that we encountered in our work could also be encoded by using code duplication.

```
char params[] = "−a\0" + "−o\0"
#if (FIND_NOT)
"!\0"
#endif
#if (DESKTOP)
"−and\0"+ "−or\0"
#if (FIND_NOT)
"−not\0"
#endif
#endif
"−print\0"
#if (FIND_PRINT0)
"−print0\0"
#endif
... // 49 additonal lines of
... // code with string literals
;
```

(a) Exponential explosion in BUSYBOX

	Before Variability Encoding	Overhead
Typedefs	128 617	0
Structs/Unions	85 096	612
Enums	24 891	0
Global Variables	29 614	101
Methods	14 825	548
Forward Declarations	726 115	2 120

(b) Statistics of variability encoding in BUSYBOX; program elements before variability encoding and overhead introduced by duplications; summarized over all files

Figure 20: Variability encoding in C: example and statistics

Apart from the abstract patterns discussed in Section 5, we would like to illustrate an extreme variability pattern that we encountered in systems such as the LINUX kernel and the BUSYBOX tool suite [34]. In Figure 20a, we show a variable declaration (taken from BUSYBOX) storing a string for the evaluation of command-line parameters. The string itself is variable, as its substrings are annotated with 21 features. In the worst case, this implementation requires the generation of 2^{21} different string variants to be included in the corresponding variant simulator, which is infeasible in practice. The only way to solve this problem is to compute the desired string variant at run or load time using a function that resembles the compile-time computation for generating the string variants. However, this extreme pattern occurs infrequently compared to other patterns that can be handled without duplication by variability encoding.

Overall, we found that variability encoding is feasible for real-world C programs. The relatively low amount of code duplication indicates the feasibility of the overall approach, because a variant simulator without sharing cannot be analyzed faster than analyzing all of its variants. To give an impression, we measured how often code duplication is necessary when building variant simulators for BUSYBOX. In particular, we measured the number of program elements before variability encoding and how many additional program elements (overhead) arise from expressing this variability in the variant simulator, as shown in Figure 20b. The statistic represents 518 C files and the included header files.[3] 5 of the 518 transformed files still contain patterns of extreme variability such as the one shown in Figure 20a which need to be handled manually. As a key result, we found that the number of additional elements that we need to generate in code duplications is always below 4%. Hence, the generated overhead is negligible.

7. Related Work

The importance of formal foundations for configurable programs has been recognized before [23, 42, 45]. There are several publications that define and refine formal definitions of systems with variability [6, 17, 19, 22, 25] and also prove behavioral equivialence [20, 35] between different configurable program representations. However, to the best of our knowledge, we are the first to define variability encoding based on a canonical representation of the syntax of the configurable program, and not starting from abstract representations such as transition systems. This is an important aspect because, in practice, configurable programs are not implemented with transition systems or state machines, but in programming languages, such as C using #if directives. Therefore, it is important to investigate the effect of different language constructs on the correctness of variability encoding.

There are several publications in which formal representations for configurable programs are defined and discussed. Gruler et al. [22] propose PL-CCS based on the process algebra CCS. Kapsus [25] proposes the language TLA+ based on featured transition systems [12]. Fischbein et al. [20], Asirelli et al. [6], and Fantechi et al. [19] discuss formal languages based on modal transition systems. These formalisms (especially the modal transition systems) are similar to the transition systems we used in Section 4.1. Fischbein et al. [20] also use a variant of bisimulation to prove that their modal transition system correctly models the behavior of the system variants. In contrast to our work, all these formalisms rely on a graph-based representation of configurable programs, and they do not discuss programming language constructs.

Gnesi and Petrocchi [21] define the Conrolled Language for configurable programs (CL4SPL). It is designed to be used by engineers of configurable programs and to be translated to executable languages for automated verification. This work is similar

[3]We skipped 1 C-file due to type errors in the transformation result.

to ours in that they offer a language in which developers can express variability and verify that the implementation satisfies its specifications. However, this language is quite far from common programming languages and no proof for the correctness of the transformation is given.

Erwig and Walkingshaw propose the *choice calculus* [18] as fundamental representation of software variation, which has been extended and used for various scenarios such as type inference [11]. In our work, we depend on formal rules that model the behavior of JAVA. Unlike CFJ, the choice calculus is not bound to any mainstream language, and therefore does not encode variability in the host language, as we do.

Mitgaard et al. [36] developed a formal framework to derive variability-aware static analyses from standard static analyses. They prove that the variability-aware part of analyses adapted with their framework is correct by construction. The overall goal of their research is the same as ours, but our proof for variability encoding enables the reuse of existing analyses without further adaption.

There are several other publications [2, 8, 16] in which researchers define frameworks for compile-time variability and variant generation based on JAVA. They formally define syntax and typing rules as well as how variants are derived from a configurable program. They also show that the derivation process preserves type correctness. However, they do not use variability encoding or other load-time variability mechanisms. As type correctness is a requirement for behavioral analysis, we see this work on type checking and others (e.g., [28]) as premise to variability encoding.

Classen et al. [13] and Post and Sinz [41] used approaches that are similar to variability encoding. Classen et al. developed a verification engine based on *featured transition systems* (FTS), which represent control-flow graphs with presence conditions on edges. As input language, they use the language fSMV [40], which is based on PROMELA, and encodes variability with the conditional language constructs of PROMELA. fSMV programs are automatically loaded as FTS and verified. Post and Sinz *manually* encoded variable statements of LINUX device drivers with guarding conditional statements and verified the resulting variant simulator. Both approaches rely on manual encoding of variability; correctness proofs are not available.

8. Conclusion

Variant simulators for configurable programs have various applications. However, the details and properties of variability encoding have not been defined and reasoned about formally so far. Specifically, there was no formal evidence as to whether efficient variant simulators exactly simulate the behavior of all valid variants of the configurable program or not. To fill this gap, we formally define the process of variability encoding, which starts from the code base of the configurable program and generates a variant simulator. We formally define syntax, typing rules, and evaluation rules of variants and of the variant simulator. We prove that the variant simulator subsumes the behavior of all valid variants of the configurable program and that the simulator does not simulate more than the valid variants. Due to the complexity of formally reasoning on mainstream programming languages, we base our model and proof on FEATHERWEIGHT JAVA (FJ), a subset of JAVA. We also discuss implications of common programming language constructs that were not included in the formalism.

As a key result, the proof of behavior preservation of variability encoding strengthens the validity of several projects that rely on variability encoding. It also promotes the use of variability encoding for future research projects on configurable programs implemented in mainstream programming languages. Our model of variability encoding can be used for future work by adding new language constructs, such as variability in method parameters. Another interesting avenue of future work would be to replace the simulation relation with a more sophisticated relation that models additional run-time properties such as memory assignment in an extended version of FJ.

9. Acknowledgements

We thank Christian Kästner and Jens Meinicke for discussions about early ideas on proving correctness of variability encoding. We also thank Florian Garbe for helping with the variability-encoding experiments for BUSYBOX. Our work is supported by the German Research Foundation grants AP 206/4, AP 206/5, and AP 206/6.

10. References

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines – Concepts and Implementation*. Springer, October 2013.

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

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

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

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

[6] P. Asirelli, M. Beek, A. Fantechi, and S. Gnesi. A Logical Framework to Deal with Variability. In *Proc. IFM*, pages 43–58. Springer, 2010.

[7] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

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

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

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

[11] S. Chen, M. Erwig, and E. Walkingshaw. An Error-Tolerant Type System for Variational Lambda Calculus. In *Proc. ICFP*, pages 29–40. ACM, 2012.

[12] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and their Application to LTL Model Checking. *IEEE TSE*, 39(8):1069–1089, 2013.

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

[14] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. GPCE*, pages 422–437. Springer, 2005.

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

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

[17] C. Dubslaff, S. Klüppelholz, and C. Baier. Probabilistic Model Checking for Energy Analysis in Software Product Lines. In *Proc. MODULARITY*, pages 169–180. ACM, 2014.

[18] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM TOSEM*, 21(1):1–27, 2011.

[19] A. Fantechi and S. Gnesi. A Behavioural Model for Product Families. In *Proc. ESEC/FSE: Companion Papers*, pages 521–524. ACM, 2007.

[20] D. Fischbein, S. Uchitel, and V. Braberman. A Foundation for Behavioural Conformance in Software Product Line Architectures. In *Proc. ROSATEA*, pages 39–48. ACM, 2006.

[21] S. Gnesi and M. Petrocchi. Towards an Executable Algebra for Product Lines. In *Proc. SPLC*, pages 66–73. ACM, 2012.

[22] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and Model Checking Software Product Lines. In *Proc. FMOODS*, pages 113–131. Springer, 2008.

[23] P. Heymans. Formal Methods for the Masses. In *Proc. SPLC*, page 4. ACM, 2012.

[24] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[25] T. Kapus. Specifying System Families with TLA+. In *Proc. WSEAS*, pages 98–103. World Scientific and Engineering Academy and Society, 2012.

[26] C. Kästner and S. Apel. Feature-Oriented Software Development. In *Proc. GTTSE*, pages 346–382. Springer, 2013.

[27] C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proc. GPCE*, pages 157–166. ACM, 2009.

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

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

[30] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward Variability-Aware Testing. In *Proc. FOSD*, pages 1–8. ACM, 2012.

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

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

[33] J. Liebig, C. Kästner, and S. Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proc. AOSD*, pages 191–202. ACM, 2011.

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

[35] M. Lochau. *Model-Based Conformance Testing of Software Product Lines*. PhD thesis, TU Braunschweig, 2012.

[36] J. Midtgaard, C. Brabrand, and A. Wąsowski. Systematic Derivation of Static Analyses for Software Product Lines. In *Proc. MODULARITY*, pages 181–192. ACM, 2014.

[37] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[38] H. Nguyen, C. Kästner, and T. Nguyen. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proc. ICSE*, pages 907–918, 2014.

[39] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

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

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

[42] I. Schaefer and R. Hähnle. Formal Methods in Software Product Line Engineering. *Computer*, 44(2):82–85, 2011.

[43] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software Diversity: State of the Art and Perspectives. *Software Tools for Technology Transfer*, 14(5):477–495, 2012.

[44] N. Siegmund, A. von Rhein, and S. Apel. Family-Based Performance Measurement. In *Proc. GPCE*, pages 95–104. ACM, 2013.

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

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

[47] Thomas Thüm, Jens Meinicke, Fabian Benduhn, Martin Hentschel, Alexander von Rhein, and Gunter Saake. Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In *Proc. SPLC*. ACM, 2014.

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

[49] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proc. ONWARD*, pages 213–226, 2014.