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

Alexander von Rhein\textsuperscript{a,\ast}, Thomas Thüm\textsuperscript{b}, Ina Schaefer\textsuperscript{b}, Jörg Liebig\textsuperscript{a}, Sven Apel\textsuperscript{a,\ast}

\textsuperscript{a}University of Passau, Innstraße 33, D-94032, Germany
\textsuperscript{b}TU Braunschweig, Mühlenpfordstraße 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 \textsc{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
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 JAV
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 cre-
ate 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 repli-
cating 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 con-
trol 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 config-
uration, 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 behav-
ior 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 en-
coding 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 experi-
ence 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 correspond-
  ing configurable system in CFJ.
- A discussion of the implementation and correctness of variability encoding in
  the presence of language constructs common in mainstream programming lan-
  guages.
- A report of our experience with generating variant simulators for practical appli-
cations.
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.
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 11, 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 $\phi$ 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 [11]). 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 $\Phi$ 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 $\Phi$, and presence conditions, denoted with $\phi$, are also propositional formulas over options. For example, $\phi_1 = \text{Color} \lor \text{Duplex}$ is a presence condition. Each valid configuration has exactly one satisfying assignment (i.e., each configuration option is either selected or deselected). As $\Phi$ 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 sat$(\phi \land \Phi)$ returns true iff there is a variable assignment $A$ under which $\phi$ and $\Phi$ hold (i.e., $A : A \models \phi \land \Phi$). In terms of configurations, this means that there exists a configuration that is valid under the constraints $\phi$ and $\Phi$.
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).

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
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 ProgramState_1). Early joining merges the traces as soon as they reach equal states (ProgramState_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.
between variants and the corresponding variant simulator in Figure 4. Given a configurable program \( p \) with variability model \( \Phi \) and code base \( \Delta \), we use weak bisimulation to prove that the execution of any variant \( \pi_\Phi \) (generated with configuration \( \Phi \)) and the variant simulator \( \sigma \), restricted to \( \Phi \), yields the same observable behavior (Section 4). This proof shows the soundness and completeness of variability encoding. In particular, we derive the variant \( \pi_\Phi \) with \( \text{derive}(\Delta, \Phi) \) and encode the variant simulator \( \sigma \) with \( \text{encode}(\Delta, \hat{\Phi}) \) (both functions are defined in Section 3.4). With \( \text{exec}(\pi_\Phi) \) and \( \text{exec}(\sigma|\Phi) \) we denote the execution of variant and variant simulator assuming configuration \( \Phi \), 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 overriding]. 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
are assumed to contain no duplicate names. For example, the parameter list of a method definition is denoted as \((C \times), 3 \Rightarrow 3\), which expands to \((C, x_1, \ldots, C, x_n)\).

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 \((new C(x)).m(y)\) is evaluated using E-INVKNEW. This rule applies alpha-equivaluent substitution \([(x \mapsto y)]t_0\) replacing occurences of \(x\) in \(t_0\) with \(y\). The term \((new C(\emptyset)).m(\emptyset)\) 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 \((new C(\emptyset))\), 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 \(C <: D\).

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 con-
ditions 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 \(\Phi\), 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 \(\phi_e\) are satisfied \((\text{sat}(\phi_e \land \Phi))\) and removing all elements with unsatisfied conditions. The function \text{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 \((CT, AT, MT, \text{init})\) 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 \(\text{init}\). 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 \(a_i\) 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 \(a_1 = MT(a)\) can again have alternatives \(a_2 = MT(a_1)\). During variant derivation, for each program element \(a\), \(MT\) is used to recursively search an alternative \(a_i\) for which \(AT(a_i)\) 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 \text{printMulti}(f,b)\ in Line 3 is replaced by \text{printDuplex}(f, b)\ iff feature Duplex is selected. In this case, the declaration of method \text{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
```java
class Printer {
    void print(Page f, Page b) {
        return printMulti(f, b);
    }
    void printDuplex(Page f, Page b) { ... }
    // other methods omitted
}
```

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:
   (\phi \ ? t : t)  feature choice
   t.@C.m(t)  method-lookup annotation
```

Figure 8: The syntax FJSIM adds to FJ. \( \phi \) 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 \(\Phi\). 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 \(\Phi\) which has been fixed at load time.

**Syntax.** At the syntax level, we make two extensions. First, we introduce the ternary operator presence condition \(\phi \ ? t : t\) with a similar semantics as in JAVA. During evaluation of a ternary operator, its presence condition \(\phi\) is evaluated with respect to the given configuration \(\Phi\) (i.e., \(\text{sat}(\phi \land \Phi)\) means that the presence condition is satisfiable in configuration \(\Phi\)). Second, we introduce the syntax construct `t.@C.m(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(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, auxiliary functions and evaluation rules we introduce for FJSIM. The upper part of the figure shows typing rules for typing feature
Typing

\[
\begin{align*}
\Gamma \vdash t_0 : C & \quad \Gamma \vdash t_1 : C \\
\Gamma \vdash (\theta \ ? \ t_0 ; t_1) : C
\end{align*}
\]  

\text{(T-VARENC)}

\[
CT(E) = \text{class } E \ldots \{ \ldots \text{mbody}(\text{G}) \{ \ldots \} \}
\]

\[
D < < E \quad \Gamma \vdash t : C \quad C < : D \quad \Gamma \vdash \Pi \top H \quad H < : G
\]

\[
\Gamma \vdash t @ D.m[\overline{n}] : F
\]  

\text{(T-SUPERREF)}

\[
\overline{x} \in C \; \text{this} : C_0 \; \text{super} : D \vdash t_0 : E_1 \quad E_1 < : C_1
\]

\[
CT(C_0) = \text{class } C \text{ extends } D \{ \ldots \}
\]

\[
\text{override}(m, D, C \rightarrow C_1)
\]

\[
\Gamma \vdash C_1.m(\overline{x}) \{ \text{return } t_0 ; \text{OK in } C_0 \}
\]  

\text{(METHOD TYPING replaces rule from [39])}

Auxiliary functions

\[
CT(C_0) = \text{class } C_0 \text{ extends } D \{ CT; KM \}
\]

\[
B(m, \{ \text{return } t_0 ; \} \in M)
\]

\[
mbody(m, C_0) = (\overline{x}, \text{super } \rightarrow \text{this } @ D | t)
\]

\[
\text{hasSuperImpl}(C, M)
\]  

\[
\text{sat}(\phi \land \Phi)\\n(\phi \land t_0 : t_1) \rightarrow t_0
\]  

\text{(E-VARENC-EN)}

\[
mbody(m, D) = (\overline{x}, t_0)
\]

\[
\text{hasSuperImpl}(C, M)
\]  

\text{(E-INVKNEWSUPER)}

\[
\text{sat}(\neg \phi \land \Phi)\\n(\phi \ ? t_0 : t_1) \rightarrow t_1
\]  

\text{(E-VARENC-DIS)}

Figure 9: The typing rules, auxiliary functions, and evaluation rules FJSIM adds to FJ; \(\phi\) denotes a presence condition, \(\Phi\) a configuration.

choices, as well as rules for typing the superclass lookup. The expression \(\Gamma \vdash t : C\) denotes that the term \(t\) is of type \(C\) in context \(\Gamma\), which maps bound variables to types. \(\text{T-VARENC}\) enforces that the terms in the then and else branches of a feature choice have the same type. \(\text{T-SUPERREF}\) 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 \(\text{E-INVKNEWSUPER}\) 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 auxiliary function \(\text{mbody}\), which we redefined in Figure 9 compared to FJ. The redefined \(\text{mbody}\) 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 \(\text{E-INVKNEW}\) [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 C()).m()

(b) Evaluation with method-lookup annotation

E@NEWSUPER with mbody(m, E)
E@NEWSUPER with mbody(m, D)
E@NEWSUPER with mbody(m, C)

(c) Incorrect evaluation without method-lookup annotation

E@NEWSUPER with (naive) super handling and mbody(m, D)
E@NEWSUPER with mbody(m, C)

The resulting term (new C(this)) is again substituted 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-VAR-EN and E-VAR-ENC-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-VAR-EN or E-VAR-ENC-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 ¬φ 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 a 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, init, Φ) 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 auxiliary functions ◀ ◀ ◀, ◀ ◀ ◀ ◀, and ◀ ◀ ◀ ◀. These functions transform a single CFJ syntax element (class, method, term) into an FJ syntax element. Function ◀ ◀ ◀ traverses the elements of the program recursively and invokes the derivation of term variants. Function ◀ ◀ ◀ ◀ chooses between alternative program elements according to the given configuration Φ by iterating alternatives defined in MT. For example, ◀ MT (printMulti(t,b)), printDuplex(t,b) ▶ selects between
printMulti(f, b) and its alternative printDuplex(f, b) in Figure 7. Function \([\cdot]\) removes program elements if their presence condition is not satisfied. For example, the method definition of printDuplex is eliminated in \([\text{void printDuplex(Page f, Page b) \{ \ldots \}}]\) 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 \((\text{Duplex ? printDuplex(f, b); printMulti(f, b)})\) in FJSIM. Figure 12 shows the definition of function encode. Function encode generates a variant simulator in FJSIM for a given CFJ program \((CT, AT, MT, \text{init}, \Phi)\). Function encode uses the auxiliary functions \(<:\cdot>\) and \(<:\cdot,\cdot>\), which we redefine in Figure 12. The figure also defines function \([\cdot]_C\) where \(C\) denotes the class containing the term \(t\).

The omitted cases of \(<:\cdot,\cdot>\) are the same as in Figure 11. Function \(<t_1, t_2>\) 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 \(\Phi\) instead of a configuration \(\Phi\), so that variable parts are only dropped if their presence condition is not satisfiable in \(\Phi\). Function \([\cdot]_C\) 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.
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 occurrence 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...
4.2. Proof of Behavior Preservation

For transition systems for FJ programs (system variants) is defined analogous.

The generation $\text{genTS}(CT, t)$ of a variant is defined recursively, adding states and transitions for each term to the transition system. The generation step that rewrite one term into another. Presence conditions are propositional formulas 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 $\text{genTS}$ 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.

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

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 $\pi_\Phi$ and a variant simulator $\sigma$. Using function $\text{genTS}(CT, t)$ of Figure 13, we define the transition system of a program $p = (CT, \text{init}, \Phi)$ as $\text{genTS}(CT, \text{init})$. 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 formulas 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 $\text{genTS}$ 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 $\Delta$ are well typed with respect to the variability model $\Phi$ [28], as the behavior of ill-typed
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 $\pi_\Phi$ with function $\text{derive}(\Delta, \Phi)$ for every configuration $\Phi$. The generated variant $\pi_\Phi = (CT_\Phi, \text{init}, \text{true})$ has a class table $CT_\Phi$, which is constructed with the function $\text{encode}$ of Figure 11, and a start term $\text{init}$. The corresponding variant simulator $\sigma = (CT_\sigma, \text{init}, \Phi)$ is generated using function $\text{encode}(\Delta, \Phi)$ of Figure 11 and Figure 12.

Hence, the variant simulator $\sigma$ and the generated variant $\pi_\Phi$ are constructed from the same configurable code base $\Delta$, and the execution of both programs, $\sigma$ and $\pi_\Phi$, starts with the same term.

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

The transition system for variant simulator $\sigma$ is denoted with $(S_\sigma, \rightarrow_\sigma \times PC)$ and derived by $\text{genTS}(CT_\sigma, \text{init})$. When we execute the simulator $\sigma$ with configuration $\Phi$, we use the corresponding projection of the transitions from $\rightarrow_{\pi_\Phi}$ to $\rightarrow_\Phi = \{(s,pc,s') \in \rightarrow_{\pi_\Phi} | \Phi \land AT(pc)\}$. This corresponds to executing 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 $s_{\pi_\Phi}$ and $s'_{\pi_\Phi}$, and states from the variant-simulator transition system with $s_\sigma$ and $s'_{\sigma}$. If it is clear from the context, we omit the subscripts in the transition relations $\rightarrow_{\pi_\Phi}$ and $\rightarrow_\Phi$.

Based on these definitions, Figure 14 illustrates the weak bisimulation property we want to prove. $R_{ID}$ is the simulation relation, which relates states of the 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 $s_{\pi_\Phi}$ and $s_\sigma$, $(s_{\pi_\Phi}, s_\sigma) \in R_{ID}$ holds, if the terms represented by $s_{\pi_\Phi}$ and $s_\sigma$ are equal.

---

1 We choose the weak bisimulation property over the often weaker weak-trace-equivalence property because both properties are equal in our case. Bisimulation and trace equivalence 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 $\Phi$).
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 $s'_q$ of $s_q$, there exists a state $s'_\sigma$ in the variant-simulator transition system that is related to the state $s'_q$ of the variant transition system with $(s'_q, s'_\sigma) \in R_{ID}$, and that $s'_\sigma$ is a successor state of $s'_{\sigma}$. Property $p_{ii}$ states that, for each successor state $s'_\sigma$ of $s_{\sigma}$, there exists a state $s'_q$ in the variant-simulator transition system that is related to the state $s'_\sigma$ of the variant transition system, with $(s'_{\sigma}, s'_q) \in R_{ID}$, and that $s'_q$ is a successor state of $s_q$. In weak bisimulation, $s'_\sigma$ does not need to be a direct successor to $s_{\sigma}$; in our case there may be a number of auxiliary states in between that evaluate feature choices. We denote such a sequence of states linked by consecutive transitions with $s_{\sigma} \xrightarrow{\ast} s'_\sigma$. A $s_{\sigma} \xrightarrow{\ast} s'_\sigma$ sequence starts with an evaluation rule from normal FJ and continues with zero or more applications of the new FJSIM evaluation rules (E-VAR-E-NC-E, E-VAR-E-NC-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 $(\text{new } C()).m()$ that is evaluated. The classes C, D, and E implement method $m$. The term $(\text{new } C()).m()$ is evaluated as shown in the figure if the presence conditions $\phi_1$, $\phi_2$, and $\phi_3$ are not satisfiable, and $\phi_4$ is satisfiable in a configuration $\Phi$. In this case, term $(\text{new } C()).m()$ evaluates to term $(\text{new } X())$, defined in E. As a consequence, the transition system of the variant simulator must contain a path from $(\text{new } C()).m()$ to $(\text{new } X())$, which may contain auxiliary 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 $\Phi$, (2) the method body return $t$ of a method $m(\bar{x})$ in class $C$, and (3) that a call (new $C(\bar{v}))$.m($\bar{u}$) is type correct in $\Phi$, there exists a chain of consecutive states $t \xrightarrow{\ast} \ldots$ that either evaluate $t$ (i) to a term $t_{\Phi}$ that has a presence condition satisfied by $\Phi$ and is one of $m$'s alternative implementations in $C$, or (ii) to (new $C(\bar{v}))$.&&$D$.m($\bar{u}$) if no such term $t_{\Phi}$ exists and a superclass of $C$ implements $m$. This evaluation sequence uses only the rules TS-VAR-E-NC-EN and TS-VAR-E-NC-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 \( t_\phi \) (Case i) or to \( (\text{new} \Phi) \circ D.m(\bar{v}) \) (Case ii) if it is annotated with a presence condition satisfied in \( \Phi \) 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 \( AT(t) \) must be satisfied by \( \Phi \), 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 \( (\text{new} \Phi) \circ D.m(\bar{v}) \), 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 \( t_{1}' = (\phi ? t_\phi : t_{\neg \phi}) \). The subterms \( t_\phi \) and \( t_{\neg \phi} \) have \( n \) or less feature-choice terms. From the definition of \( \text{genTS} \) (Figure 13) \( \text{TS-\text{VAR}ENC-EN} \) and \( \text{TS-\text{VAR}ENC-DIS} \), we know that the variant-simulator transition system contains the transitions \( tr \) from \( t_1 \) to \( t_\phi \) and \( tr' \) from \( t_1 \) to \( t_{\neg \phi} \) (with the presence conditions \( \phi \) and \( \neg \phi \), respectively). Either \( \phi \) is satisfied by configuration \( \Phi \) (i.e., \( \text{sat}(\phi \land \Phi) \)) or the negation \( \neg \phi \) is satisfied by \( \Phi \). Exactly one of \( \phi \) or \( \neg \phi \) is satisfied, because \( \Phi \) is a configuration, and as such has a fixed value for each configuration option. The code that is invoked if \( \phi \) is satisfied is encoded in \( t_\phi \) and the code that is invoked if \( \neg \phi \) is satisfied is encoded in \( t_{\neg \phi} \) (function \( \ll \cdot \gg \)， Figure 12). First, we consider the case where \( \phi \) is satisfied under configuration \( \Phi \). 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 \( t_\phi \) is not a feature choice, it does not contain any further variability and one of the base cases applies. Otherwise, the resulting term \( t_\phi \) is a feature choice. This means, \( t_\phi \) has the same syntactic form as \( t_{1}' \) in the beginning of the inductive 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 \( \Phi \) the proof is analogous. In each step of the evaluation \( \text{E-\text{VAR}ENC-EN} \) or \( \text{E-\text{VAR}ENC-DIS} \) is applied.

The induction shows that there is a sequence of consecutive states \((t,t_2,\ldots,t_n)\) 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.

\( \blacksquare \)

**Theorem 1.** Given a CFJ code base \( \Delta \) with a variability model \( \Phi \) and a start term \( \text{init} \), a configuration \( \Phi \), the simulation relation \( R_{ID} \) (term equality), a variant \( \pi_\Phi = (CT_\Phi, \text{init}, \text{true}) = \text{derive}(\Delta, \Phi) \), a variant simulator \( \sigma = (CT_\sigma, \text{init}, \Phi) = \text{encode}(\Delta, \Phi) \), which is executed with \( \Phi \), and the corresponding transition systems \((S_\Phi, \to, PC_\Phi) = \text{genTS}(CT_\Phi, \text{init})\) and \((S_\sigma, \to, PC_\sigma) = \text{genTS}(CT_\sigma, \text{init})\), then the weak bisimulation property holds:
\[ \forall s_\Phi \in S_\Phi, \forall s_\sigma \in S_\sigma \text{ with } (s_\Phi, s_\sigma) \in R_{ID} : \\
(p_1) \quad \forall s'_\Phi \text{ with } s_\Phi \rightarrow s'_\Phi : \quad \exists s'_\sigma \in S_\sigma \text{ such that } (s_\sigma \rightarrow s'_\sigma \text{ and } (s'_\Phi, s'_\sigma) \in R_{ID}) \text{ and } \\
(p_{ii}) \quad \forall s'_\sigma \text{ with } s_\sigma \rightarrow s'_\sigma : \quad \exists s'_\Phi \in S_\Phi \text{ such that } (s_\Phi \rightarrow s'_\Phi \text{ and } (s'_\Phi, s'_\sigma) \in R_{ID}) \]

**Proof.** We prove the two properties of bisimulation \((p_1, p_{ii})\) separately.

**Property \(p_1\).** We prove \(p_1\), 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 \((\text{Figure 13})\). 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 below. For a complete proof, we refer to the supplementary material.

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

**Case 3 (TS-INVKNEW):** Let \(s_\Phi = (\text{new } C(\overline{v})).m(\overline{v})\). Method \(m\) in \(\pi_\Phi\) must have been generated with variant generation rule \(G.5\) of Figure \(11\). It is important to note that \(\overline{v}\) and \(\overline{u}\) are lists of values (values cannot be evaluated any further). Let \(t_0\) be the body of method \(m\) in the variant \(\pi_\Phi\). As the method body is included in \(\pi_\Phi\) with Rule \(G.5\), we can conclude that the configuration \(\Phi\) implies the presence condition \(AT(t_0)\) and \(s'_\Phi = t_0\). As \(s_\Phi\) is in simulation relation to \(s_\sigma\), we know that \(s_\sigma = (\text{new } C(\overline{v})).m(\overline{v})\). From the definition of \(\text{genTS}\) \((\text{Figure 13})\) TS-INVKNEW \(\), we infer \(1\) that \(-\rightarrow_{\sigma, \Phi}\) contains a transition \(\tau_1 = (s_\sigma \rightarrow ([\overline{x} \mapsto \overline{u}, \text{this } \rightarrow \text{new } C(\overline{v})]t'_j))\) \(\) with \(\text{mbody}(\sigma, \overline{m}, C) = t'_j\) and \(2\) that \(\tau_1\) has the presence condition \text{true}.

We use induction over the number of superclasses of \(C\) in simulator \(\sigma\) to prove that \(s_\Phi\) and \(s_\sigma\) evaluate to the same term. In the base case, class \(C\) is a direct subclass of \text{Object}. \text{Object} does not implement any methods \([39]\). Because each variant is well typed, term \(s_\Phi\) is also well typed, and we can apply Lemma \([\text{L1}]\) with configuration \(\Phi\). The application of Lemma \([\text{L1}]\) shows that there exists a list of consecutive states \((s_\sigma, t_1, \ldots, t_n)\) in the variant-simulator transition system that evaluate \(s_\Phi\) to either a term \(t_n\) with a presence condition satisfied by \(\Phi\) or to a call of \(m\) in a superclass. As \(C\) does not have superclasses (other than \text{Object}), we know that \(s_\sigma\) evaluates to \(t_n\) with \(t_n = s'_\sigma\) and \((s'_\Phi, s'_\sigma) \in R_{ID}\). The evaluation step \(s_\sigma \rightarrow t_2\) applies evaluation rule \(\text{E-INVKNEWSUPER}\). 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 \text{Object}. The term \(s_\Phi\) is an invocation of method \(m\) on class \(C\) and the invocation is well typed in variant \(\pi_\Phi\), because either \(C\) has an implementation of \(m\) in the variant \(\pi_\Phi\) or the method invocation is dispatched to an implementation of \(m\) in a superclass. If \(C\) itself has an implementation, Lemma \([\text{L1}]\) shows that there is a chain of states in \(\sigma\) from \(s_\Phi\) to \(s'_\Phi\). As \(s'_\Phi\) is the first alternative implementation of \(m\) that satisfies the configuration \(\Phi\), \(s'_\Phi\) is in simulation relation to \(s'_\Phi\). If \(C\) does not have an implementation of \(m\) in the variant \(\pi_\Phi\), Lemma \([\text{L1}]\) shows that there is a chain of states in simulator \(\sigma\) from \(s_\sigma\) to
(new C(φ))@D.m(u). This chain uses only TS-VARENCE-DIS. This new expression invokes method m in the direct superclass D of C. Because v and u are values, the next evaluation rule must be TS-INVKNESUPPER. 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 $s_{Φ_D}$ in variant $π_D$.

Therefore, $s'_Φ = l_0$, $s'_ρ = t_n$, and $(s'_Φ, s'_ρ) ∈ R_{ID}$. So, there exists a state $s'_ρ ∈ S_{ρ}$, such that $(s'_Φ, s'_ρ) ∈ R_{ID}$ and a sequence of transitions $(s_ρ \rightarrow s'_ρ)$. The sequence starts with TS-INVKNESUPPER followed by applications of TS-VARENCE-EN, TS-VARENCE-DIS, and TS-INVKNESUPPER, which concludes Case 3.

Case 4 and Case 5 (TS-VARENCE-EN and TS-VARENCE-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-VARENCE-EN and TS-VARENCE-DIS may apply. Therefore, Case 4 and Case 5 cannot occur.

Case 6 (TS-INVKNESUPPER): 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-INVKNESUPPER 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-INVKNESUPPER) 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 $s_Φ = (new C(φ)).m(u)$ with $(s_Φ, s_{Φ}) ∈ R_{ID}$ and a state $s_{Φ}$ with $(s_Φ \rightarrow s_{Φ}')$, we must show that the variant contains a state $s_Φ'$ with $(s_Φ', s_{Φ}') ∈ R_{ID}$. Interestingly, $s_{Φ}'$ is the only state in the sequence $(s_Φ \rightarrow s_{Φ}')$ that can be present in $S_{Φ}$. All intermediary states between $s_{Φ}$ and $s_{Φ}'$ are evaluated with the rules TS-VARENCE-EN, TS-VARENCE-DIS, or TS-INVKNESUPPER 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. $s_Φ$ evaluates to $s_{Φ}'$ assuming configuration $φ$, and the argumentation of Case 3 in the proof of property $p_i$ shows that $s_{Φ}'$ is equal to the state that $s_Φ$ evaluates to in the variant. Therefore, variant $π_Φ$ has a state $s_Φ'$ with $(s_Φ', s_{Φ}') ∈ R_{ID}$ and a transition $(s_Φ \rightarrow s_Φ')$, which concludes Case 3.

Cases 4, 5 and 6 (TS-VARENCE-EN, TS-VARENCE-DIS, and TS-INVKNESUPPER) are not relevant, as the given states $s_Φ$ and $s_{Φ}$ 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.

\[\square\]

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 sim-
ulator 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 com-
bined. 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 design-
ing the variant simulator. One crude approach would be to apply method renaming and
(a) Problem: Optional method overloading with inheritance

```java
class A {}
class B extends A {}
class X { A m(A a) {...} }
class Y extends X {
  A m(B b) { return new A(); }
}
(new Y()).m(new B());
```

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

Figure 16: Optional method overloading with inheritance in JAVA

---

(a) Problem: Method overloading with optional parameters

```java
class A {
  int x = 0;
  if (Opt1 & Opt2) { return 0; }
  int m();
  int m(int x) {
    int m();
    if (Opt1) { return x+1; }
    return x+1; }
}
```

(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 `m ∈ 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);
}

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);
}

(a) Problem: Run-time system

(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:

```c
int #if (Opt1) 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
can 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, op-
tional 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 vari-
able 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 initia-
ization 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 vari-
ants cannot interfere. The configuration of all feature variables is selected and fixed

25
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. 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] 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 [45]), 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 improves 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 TYPECHEF 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 \texttt{!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 \texttt{u32} and \texttt{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"
#else (FIND_NOT)
"−not\0"
#endif
#else (FIND_PRINT0)
"−print0\0"
#endif
... // 49 additional lines of
... // code with string literals
```

(a) Exponential explosion in BUSYBOX

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

<table>
<thead>
<tr>
<th></th>
<th>Before Variability Encoding</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typedefs</td>
<td>128617</td>
<td>0</td>
</tr>
<tr>
<td>Structs/Unions</td>
<td>85096</td>
<td>612</td>
</tr>
<tr>
<td>Enums</td>
<td>24891</td>
<td>0</td>
</tr>
<tr>
<td>Global Variables</td>
<td>29614</td>
<td>101</td>
</tr>
<tr>
<td>Methods</td>
<td>14825</td>
<td>548</td>
</tr>
<tr>
<td>Forward Declarations</td>
<td>726115</td>
<td>2120</td>
</tr>
</tbody>
</table>

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. 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 equivalence [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 Controlled 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

---

3We 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


