A Variability-Aware Module System

Christian Kästner, Klaus Ostermann, and Sebastian Erdweg
Philipps University Marburg, Germany

Module systems enable a divide and conquer strategy to software development. To implement compile-time variability in software product lines, modules can be composed in different combinations. However, this way variability dictates a dominant decomposition. Instead, we introduce a variability-aware module system that supports compile-time variability *inside* a module and its interface. This way, each module can be considered a product line that can be type checked in isolation. Variability can crosscut multiple modules. The module system breaks with the antimodular tradition of a global variability model in product-line development and provides a path toward software ecosystems and product lines of product lines developed in an open fashion. We discuss the design and implementation of such a module system on a core calculus and provide an implementation for C, which we use to type check the open source product line Busybox with 811 compile-time options.

1 Introduction

A *module system* allows developers to decompose a large system into manageable subsystems, which can be developed and checked in isolation [13]. A *module* hides information about internal implementations and exports only a well-defined and often machine-enforced *interface*. This enables an *open-world* development style, in which software can be composed from modular self-contained parts.

The need for *compile-time variability*, for example in software product lines [6, 17, 10], challenges existing module systems. To tailor a software system, stakeholders may want to select from compile-time *configuration options* (or features) and derive a specific *configuration* (or variant, or product) of the system. At compile-time, a user selects of which configuration options code should be compiled into the system. In a modular scenario, we can derive different configurations by composing different subsets of modules. However, to encode variability only at composition level, variability must *align* with the modular structure, and each compile-time configuration option must be expressed as a separate module. When variability *crosscuts* the dominant decomposition, a modular implementation becomes tricky: A configuration option, such as transaction support in a database, may affect multiple modules and may even change their interfaces [30].

In fact, state of the art product-line implementations often use antimodular concepts: *Conditional compilation*, typically with #ifdef directives of the C preprocessor, is common and crosscuts entire implementations [34]. Intended variability of the product line is commonly described in a single global *variability model* in a closed-world fashion. As long as product-lines are developed entirely by a small team inside a single company, this closed-world view may suffice. But, for larger product lines developed by multiple teams, for product lines that should be reused in other contexts, and for product lines that span organizational units, a modular solution is needed.

An additional challenge comes from the *combinatorial explosion* of configuration options. There are $\mathcal{O}(2^n)$ compile-time configurations of a product line with n configuration options. Checking all configurations one by one in a brute-force fashion is infeasible in practice. Likewise, checking only specific configurations at module-composition-time defies the purpose of modularity, since conflicts are detected only late.

To enable *modular product-line development*, we introduce a variability-aware module system that supports both *inner variability* inside a module and *crosscutting variability* that affects multiple modules. In the module system, each module can be considered as a product line in itself. Module composition becomes the composition of entire product lines including their variability.

We *formalize* our variability-aware module system as calculus. The distinguishing feature of this module calculus is that interfaces and implementations are *variable* depending on the selection of configuration options. Furthermore, each module defines its own local variability model – the constraints on its environment. The formalization is based on Cardelli's seminal formalization of separate type checking and linking [13] and a more recent generalization of this work towards propositions in interfaces [31]. We show that the calculus is sound in two ways: (1) well-typedness of a module implies well-typedness of all configurations of the module, and (2) module composition preserves well-typedness.

We *implement* a variant of our variability-aware module system for the C programming language with #ifdef variability. Taking every translation unit (.c file) as a module with inner variability (from #ifdef directives), we *efficiently and modularly check all configurations of a module* and infer an interface with variability. Subsequently, we perform composition checks for all configurations, equivalent to linker checks in C. We encode expensive compatibility checks and type checks as Boolean satisfiability problems, building on prior work on variability-aware analysis [18, 50, 28, 3, 15]. We provide a full open-source implementation as part of our *TypeChef* project and type check the entire Busybox product line with 811 compile-time configuration options (and, thus, more potential configurations than the estimated number of atoms in the universe [32]) to demonstrate practicality. Modularity allows us to type check each of Busybox's 522 files in parallel. We show that TypeChef is able to find actual type errors in Busybox.

In summary, our central contributions are the following: (1) We *motivate* the need for inner and crosscutting variability. (2) We *design* a novel module system for product lines and discuss design decisions. (3) We *formalize* the module system as formal calculus and prove its soundness. (4) To demonstrate *practicality*, we present a practical implementation strategy, we implement the module system for C and the C preprocessor, and we find type errors with this implementation in a medium-size real-world product line. To

the best of our knowledge, this is the first implementation of modular type-checking for practical product-line implementions in C with #ifdef variability.

2 Modules and variability

Modularity as enforced by most module systems serves a simple means: It allows splitting a system into smaller subsystems (modules), each of which are divided into an internal implementation and an external interface. The module's interface describes a contract with the rest of the system in terms of imports and exports. Ideally, a developer (or compiler) can understand (or type-check and compile) a module separately, by looking only at its internal implementation and interface, but not at implementations of other modules. Internals of the module can be changed without affecting (and even knowing) any other module. This separation into modules with interfaces enables modular reasoning and reuse of modules in unplanned contexts. For uniformity, we adopt Cardelli's notion that modules have explicit imports and are closed under composition [13]. That is, two modules can be composed (or linked) to form a larger module, in which imports that are exported by the other module are removed. When composing two compatible modules, module composition should preserve well-typedness.

In practical software development, frequently a demand for variation arises. Different configurations of a system should be compiled for different platforms, customers, and use cases. Especially in software product lines, such variation is planned and used as strategic advantage. Instead of developing a software system only for a single customer, product lines cover related systems in a whole domain. Such a product-line approach promises lower costs, better quality, shorter time to market, and flexibility to react to market changes, due to strategic reuse [6, 17, 10]. For illustration and demonstration, we use two examples of product lines. First, we introduce a tailorable embedded database management system that can be configured with two different storage mechanisms – persistent and in-memory, – an optional XML layer, and other options. Second, we analyze Busybox, a real-world resource-efficient product line of UNIX utilities. In both scenarios, resource constraints of embedded systems demand compile-time reduction and specialization to the necessary core; hence, for different scenarios, different tailor-made solutions should be provided.

In the following, we outline how to implement variability with conventional module systems, we discuss their limitations, we outline our concept of a variability-aware module system, and we survey how variability is implemented in the real-world product line Busybox.

2.1 Variable module composition

To implement product lines modularly, developers usually develop a module for each configuration option and express variability by composing different sets of modules. This style of programming, in which modules *align* with configuration options, is also known as *feature-oriented programming* [42, 7] and popular in the form of plug-in systems [2, 6].

```
core = (
  import write: Key→Table→Bool;
  import read: Key→Table;

  fun log(msg: String): Unit = ...;
  export fun select(q: String): Table = ... read(...)  ...;
  export fun update(q: String): Bool  = ... write(...) ...;
  export fun main(p: String): Int = ...;
)
```

```
xml = (
  import update: String→Bool;

  fun parse(s: String): XML = ...;
  fun unparse(x: XML): String = ...;
  export fun storeXML(x: XML): Bool  = ... update(...) ...;
)
```

```
inmem = (
  export fun write(k: Key, t: Table): Bool = ...;
  export fun read(k: Key): Table  = ...;
)
```

```
persist = (
  fun fopen(f: Int): Handle = ...;
  export fun write(k: Key, t: Table): Bool = ...;
  export fun read(k: Key): Table  = ...;
)
```

Figure 1: Simple database example without inner variability.

In our database example, we could decompose a system into four modules as illustrated in Figure 1: a core database module *core*, an in-memory-storage module *inmem*, a persistent-storage module *persist*, and an XML module *xml*. Now, we derive different systems by composing (•) modules in different combinations: *core* • *inmem* yields an in-memory database, *core* • *persist* a persistent database, *core* • *inmem* • *xml* an in-memory XML database, and so forth.

Composing two modules merges their definitions. Imports of one module are matched by exports of the other module as far as possible; nonexported (private) functions are renamed or inlined if necessary. Two modules exporting the same function, such as *inmem* and *persist* in our example, are incompatible and cannot be composed.

If desired, we can *automate* the generation of tailored systems for a given selection of configuration options with a build system. Build systems range from simple shell scripts to sophisticated compilation managers [11, 43, 9]. Build systems typically introduces explicit configuration options (and possibly dependencies between them in a variability model). The configuration options are then mapped to modules [17]. For a given selection of configuration options, the build system compiles and composes the corresponding modules. In such setting, variability is expressed globally for a *fixed set* of modules at *composition level*: The modules themselves have no notion of variability, especially no variability in interfaces.

2.2 A case against variability-induced decomposition

When variability is expressed only at composition level, modules align with configuration options. On one hand, this alignment enforces separation of concerns regarding configuration options; but, on the other hand, then, variability dictates a dominant decomposition [48] of the system, which might not be the desired one. There are at least three problems of a variability-induced decomposition:

- Variability is known as a crosscutting concern [34, 25, 49, 42, 36]. In our database example, configuration options such as READONLY affect many modules and concerns. Even with advanced and controversial module constructs, such as aspects, it is not clear whether the implementation of crosscutting configuration options can

be specified in a single module [30].

- Configuration options are not independent. In our example, configuration option READONLY would affect both in-memory and persistent storage. A typical solution of the module-per-configuration-option approach is creating more modules (e.g., *inmem-common*, *inmem-write*, and *inmem-readonly*) [36], leading to an explosion of micro-modules.

- Configuration options (and interactions between configuration options) that affect only few lines of code must be extracted into their own function and module, even if they are just a minor concern in a larger context. Such small additional modules reduce the benefit of an open-world module system, because they are typically hard to reuse and tightly coupled to the rest of the system.

We argue that variability should not necessarily dictate the dominant decomposition. Although tool support could potentially address the problem of many small modules, we explore a language-based solution. In the remainder of the paper, we introduce a variability-aware module system that enforces modular checks in the presence of inner and crosscutting variability.

2.3 Variability inside modules

Instead of encoding variability using module composition, we propose to encode variability *inside* modules, such that configuration options and modules do not need to align. Each module can be interpreted as a product line that can be configured. For example, we implement a module *storage* for the storage subsystem that can be configured to use in-memory or persistent storage and to provide read-only or read-and-write access. However, with variability in interfaces, the composition process and the role of the variability model (previously part of the build system) changes.

We introduce variability with *presence conditions*. A presence condition on a code element is a formula over configuration options that specifies in which configurations the element should be included. For example, we say module *storage* defines function *fopen* only if configuration option PERSIST is selected (`fopen if PERSIST`). In the simplest case, presence conditions can be implemented by conditional compilation with #ifdef directives; we discuss alternatives in Section 4.3.

To reason about configuration options inside a module, we declare them explicitly or import them like functions. Hence, for every configuration option there is a unique module that declares the configuration option (and possibly related configuration information, such as description, defaults, costs, and interested stakeholders). Hence, there is a well-defined distinction between configuration option definition and configuration option usage, which yields a well-defined scoping concept for configuration options and enables standard techniques such as α-renaming of configuration options. Declared configuration options are always part of the interface (they cannot be hidden, because users must be able to configure the module). Finally, each module can have a local variability model that constraints possible combinations of configuration options; for this purpose,

```
storage = (
  config PERSIST  "persistent storage" default;
  config INMEM    "in-memory storage"
  config READONLY "read-only access only (faster, smaller)";
  variability (PERSIST∨INMEM)∧¬(PERSIST∧INMEM);

  fun fopen(f: Int): Handle     if PERSIST = ...;
  export fun write(x: Rec): Tid if ¬READONLY∧PERSIST = ...;
  export fun write(x: Rec): Tid if ¬READONLY∧INMEM   = ...;
  export fun read(x: Tid): Rec  if PERSIST = ... fopen ...;
  export fun read(x: Tid): Rec  if INMEM   = ...;
)
```

```
query = (
  import config READONLY;
  import write: Rec → Tid if ¬READONLY,
         read: Tid → Rec;

  config TXN, INDEX;

  fun createIndex(t: T): I if INDEX = ...;
  fun log(m: String): Bool          = ...;
  export fun select(q: String): Tab = ...;
  export fun update(q: String): Bool if ¬READONLY = ...;
  export fun main(p: String): Int   = ...;
)
```

```
xml = (
  import config READONLY, TXN, INDEX;
  import select: String → Table,
         update: String → Table if ¬READONLY,
         read: Tid → Rec;

  config XQUERY;
  variability XQUERY ⇒ INDEX;

  fun parse(e: String): Xml = ...;
  fun toXML(x: Xml): String = ...;
  export fun storeXML(x: Xml): Tid if ¬READONLY = ...;
  export fun query(q: String): Xml if XQUERY = ...;
)
```

```
storage • xml = (
  import config TXN, INDEX;
  import select: String → Table,
         update: String → Table if ¬READONLY;

  config PERSIST, INMEM, READONLY, XQUERY;
  variability (XQUERY ⇒ INDEX) ∧
        ((PERSIST∨INMEM)∧¬(PERSIST∧INMEM));

  fun fopen(f: Int): Handle     if PERSIST = ...;
  export fun write(x: Rec): Tid if ¬READONLY∧PERSIST = ...;
  export fun write(x: Rec): Tid if ¬READONLY∧INMEM   = ...;
  export fun read(x: Tid): Rec  if PERSIST = ...;
  export fun read(x: Tid): Rec  if INMEM   = ...;
  fun parse(e: String): Xml = ...;
  fun toXML(x: Xml): String = ...;
  export fun storeXML(x: Xml): Tid if ¬READONLY = ...;
  export fun query(q: String): Xml if XQUERY = ...;
)
```

Figure 2: Extended database example with inner variability.

we specify a formula, but other notations, including graphical feature diagrams, are possible [17, 51, 9, 8].

Composing two modules in our variability-aware module system is similar to composing two modules in a traditional module system. Composing two modules with variability (i.e., two product lines) yields another module that combines the variability of both (i.e., another product line). Imports are matched by exports as far as possible, and configuration options and functions are merged. Local variability models are combined, requiring now the constraints of both models. Furthermore, we will explore additional constraints on the variability model in case of function conflicts later.

In Figure 2, we illustrate the concepts with an extended database example. The system is divided into three modules *storage*, *query*, and *xml*, not aligning with variability. Each of these modules has inner variability. Furthermore, we exemplify the result of the composition $storage \bullet xml$.

2.4 Crosscutting and inner-module variability in Busybox

Before we get to a formal description, we want to emphasize the need for a proper variability-aware module system once more with a look into practice. We report data from the open-source product line Busybox. Like in many product-lines, the Busybox developers did not pursue a strictly modular approach and used the C preprocessor to encode variability inside and across modules.

We selected Busybox (release 0.18.5, available at `http://busybox.net/`) as a paradigmatic case, representing many other conditional-compilation-based product-line imple-

mentations [34]. Busybox has 522 .c files and 260 000 lines of unpreprocessed C code. BusyBox combines custom implementations of many common UNIX utilities into a single small executable for small or embedded devices. Targeted at resource-constraint environments, BusyBox is highly customizable with 811 explicitly declared Boolean compile-time configuration options, allowing selecting which utilities to include and with which facilities.

As common in C, we regard every *translation unit* (.c file with inlined header files) as a module, which a compiler can translate independently and which a linker can compose with other modules. Variability in Busybox uses both variability at composition level, automated by the build system, and variability at inner-module source-code level, encoded with #ifdef directives.

We illustrate variability in Busybox from different perspectives with metrics:

- *Variable module composition.* There is a high amount of variability at composition level. Of 522 modules, 413 modules (79 %) are composed only under some condition. Of 811 configuration options, 386 (48 %) influence variable module composition; 270 configuration options (33 %) exclusively control composition, but not inner-module variability. Variability at composition level mostly reflects the selection of entire tools to be linked into the Busybox executable, such as *grep*, *find*, or *chmod*, and compression libraries.

- *Variability inside modules.* Of 811 configuration options, 499 (62 %) control variability at source-code level with #ifdef directives. All 522 translation units contain source-code level variability. Many translation units provide configuration options that are *local* to that module and occur in no other module. For example, *find.c* has 22 local inner configuration options, *hush.c* has 13, and *httpd.c* has 11. There are 69 translation units with at least two local inner configuration options. Between configuration options inside a translation unit, there are often dependencies in the variability model, such as HUSH_JOB $\Rightarrow$ HUSH_INTERACTIVE. Hence, Busybox shows potential for local declarations of configuration options and local variability models.

- *Crosscutting variability.* In addition to 391 (48 %) configuration options local to a single translation unit (usually configuration options of individual tools, such as MODPROBE_BLACKLIST), there are 109 (13 %) configuration options that crosscut multiple translation units. Crosscutting is mostly moderate with 46 configuration options affecting between two and ten translation units, and 15 between 11 and 50 translation units. However, 47 configuration options affect over 500 (essentially all) translation units. Configuration options crosscut when several translation units together implement the same concepts, such as UNICODE_SUPPORT and SHADOWPASSWDS. Heavy crosscutting comes mostly from variability in header files that are included in most translation units, independent of whether the functionality is used. These metrics show that crosscutting configuration options are common and should be addressed by an implementation approach.

- *Variability in module interfaces.* Source-level variability does not only affect module implementations, but also their interfaces. As interface of a translation unit in C, we regard imported and exported functions (for details, see Section 5.2). Conditional compilation that controls only statements, expressions, or unused declarations do not cause variability in module interfaces.

  Overall, 11 % of all exports and 7 % of all imports are variable. Of all 811 configuration options, 303 (37 %) affect imports or exports in at least one translation unit. While again variability is mostly local to the interface of a single translation unit, 45 configuration options affect interfaces in up to ten translation units (e.g., HUMAN_READABLE, SHADOWPASSWDS), and 11 configuration options affect more than ten interfaces, with the maximum of 41 interfaces affected by IOCTL_HEX2STR_ERROR (a configuration option adjusting how errors are reported). The metrics indicate that a large amount of source-level variability is hidden inside modules and does not influence interfaces, but still handling variability in interfaces is crucial.

In summary, Busybox illustrates that variability is used both at composition level and source-code level inside modules. There is potential for local definitions of configuration options and for hiding variability implementations inside a module. Many translation units can be considered as small product lines. At the same time, crosscutting is also common. In our experience with other open-source product lines implemented in C [34], we judge Busybox as a paradigmatic case.

Traditional module systems cannot handle implementations with inner-module source-level variability. Enforcing decomposition by variability would require many additional modules and rewrites, which we regard as impractical for Busybox. On the other hand, in the current form with inner-module variability, a C compiler only determines imported and exported symbols after running the preprocessor to remove all variability from the code. Modules are composed only selecting configuration options. There is no means to check module compatibility for *all* configurations, other than applying a brute-force strategy. We conclude that our motivation for variability-aware modules is also supported by current software practice.

3 Formalization

In our module system, a module has a well-defined interface that describes the names and types of imported and exported functions. A type system checks each module in isolation against its interface and a composition engine ensures that composed modules have type-compatible interfaces and that no name clashes occur. The formalization can be seen as the specification of the desired behavior, quite distinct from our implementation. We use the formalization to prove that our module system is sound. Although the formal definitions are simple, the soundness properties are not obvious; in fact, it took several iterations of proving and fixing to get the definitions right.

Notation:

$x \in X$	function names
$e \in E$	expressions
$t \in T$	types
$\Gamma \in X \to T$	contexts / function imports
$\Delta \in X \to E \times T$	function definitions
$m = (\Gamma, \Delta) \in M$	module

Auxiliary functions:

$sig : (X \to E \times T) \to (X \to T)$
$sig(\Delta)(x) = t \quad \text{where } \Delta(x) = (e, t)$

$$\frac{\forall x \in dom(\Gamma_1) \cap dom(\Gamma_2).\ \Gamma_1(x) = \Gamma_2(x)}{typecompatible(\Gamma_1, \Gamma_2)}$$

Module typing:

$$\frac{dom(\Gamma) \cap dom(\Delta) = \emptyset \quad \Gamma \vdash \Delta}{(\Gamma, \Delta)\ \mathsf{OK}}$$

$$\frac{\forall x \in dom(\Delta).\ \Gamma \cup sig(\Delta) \vdash e : t \quad \text{where } \Delta(x) = (e, t)}{\Gamma \vdash \Delta}$$

Module compatibility and composition:

$$\frac{\begin{array}{c} dom(\Delta_1) \cap dom(\Delta_2) = \emptyset \\ typecompatible(\Gamma_1, \Gamma_2) \\ typecompatible(\Gamma_1, sig(\Delta_2)) \\ typecompatible(sig(\Delta_1), \Gamma_2) \end{array}}{(\Gamma_1, \Delta_1) \div (\Gamma_2, \Delta_2)} \qquad \frac{\begin{array}{c} \Gamma' = \Gamma_1 \cup \Gamma_2 \setminus (sig(\Delta_1) \cup sig(\Delta_2)) \\ \Delta' = \Delta_1 \cup \Delta_2 \end{array}}{(\Gamma_1, \Delta_1) \bullet (\Gamma_2, \Delta_2) = (\Gamma', \Delta')}$$

Figure 3: Module system M without variability.

3.1 A base module system M without variability

To illustrate the basic concepts, let us start with a small calculus of a module system without variability inside modules in Figure 3. The calculus follows the spirit of Cardelli's module system formalization [13].

A module consists of a set of imported function declarations with their associated type, and a list of typed function definitions with a body. Imports are modeled as partial finite maps (we overload the function arrow $\to$ to denote partial maps) from names to types, definitions are maps from names to types and expressions. The interface of a module consists of all imported declarations and the signatures of all locally defined functions. We could easily model a distinction between private definitions and exported definitions, but except for the need of renaming during composition, this adds little to our discussion; we simply assume that private functions have been inlined. Translating the example from Figure 1 into this calculus is straightforward. We leave the exact form of expressions and types open; we just assume that there is a type system for the expression language that can perform a type-check of the form $\Gamma \vdash e : t$. The only requirement on the typing relation is that it must be monotonic (if Γ' is an extension of Γ and $\Gamma \vdash e : t$, then

$\Gamma' \vdash e : t$), otherwise module composition would not preserve well-typedness. Almost all type systems used in practical programming languages have this property.[1]

We type check each module in isolation. A module is well-typed ($m\ \mathsf{OK}$) if all function bodies are well-typed in the context of imported and defined functions and if a function is not both imported and defined.

Two modules are compatible ($m_1 \div m_2$) unless they contain a *function conflict*. There are three kinds of possible function conflicts: (1) both modules export a function with the same name, (2) both modules import a function with the same name but with different types, and (3) one module imports a function defined in the other with a different type. Composing two modules essentially merges imports and exports, and imports provided by the other module are removed. To compose two modules with a function conflict, first the conflict must be resolved; for example, developers can rename the function in one module to make both modules compatible (see also the rename operator of the composition language in Section 3.4).

Module system M has the following desirable properties:

(P1) The module system is closed under composition, that is, composing two modules yields a new module ($\bullet : \mathsf{M} \times \mathsf{M} \to \mathsf{M}$).

(P2) We can type check each module in isolation (against its own interface), independent of other modules ($m\ \mathsf{OK}$).

(P3) To determine whether two modules are compatible ($m_1 \div m_2$), we only need to investigate their interfaces, not their internal implementations.

(P4) When composing two well-typed compatible modules, and the typing relation is monotonic, then the composed module is well-typed as well ($m_1\ \mathsf{OK} \wedge m_2\ \mathsf{OK} \wedge m_1 \div m_2 \Rightarrow m_1 \bullet m_2\ \mathsf{OK}$).

(P5) Composition is associative and commutative.

(P6) Module compatibility is closed under module composition ($m_1 \div m_2 \wedge m_1 \div m_3 \wedge m_2 \div m_3 \Rightarrow m_1 \div (m_2 \bullet m_3)$), as proved in the appendix.

We want to preserve these properties when we move to a variability-aware module system.

3.2 A variability-aware module system M^ν

Now, let us introduce variability into the module system M. For clarity, we proceed in two steps: First, we add variability with a global name space for configuration options in M^ν, as specified in Figure 4. Subsequently, in the next subsection, we add a scoping concept for configuration options in $\mathsf{M}^{\nu\mathsf{l}}$. The calculus models semantics and is hence rather abstract: We leave open how sets of configuration options are represented (usually with propositional formulas) and just represent them semantically as sets of (or mappings

[1] Often called *weakening*; *substructural* type systems [53] that violate this property (such as linear types) are uncommon in practice.

Additional notation:

$f \in F$	configuration options
$c \in C = 2^F$	configurations
$\nu \in V = 2^C$	variability models
$\Gamma \in C \to X \to T$	variable contexts
$\Delta \in C \to X \to E \times T$	variable definitions
$m = (\nu, \Gamma, \Delta) \in M^\nu$	module

Auxiliary functions:

$Sig : (C \to X \to E \times T) \to (C \to X \to T)$
$Sig(\Delta)(c)(x) = t \quad \text{where } \Delta(c)(x) = (e, t)$

Module typing:

$$\frac{\nu \subseteq dom(\Gamma) \qquad \nu \subseteq dom(\Delta) \qquad \forall c \in \nu.\ (\Gamma(c), \Delta(c))\ \mathsf{OK} \qquad \nu \neq \emptyset}{(\nu, \Gamma, \Delta)\ \mathsf{OK}}$$

$$\begin{aligned}
conflictpresence(\Gamma_1, \Gamma_2) &= \{c \in dom(\Gamma_1) \cap dom(\Gamma_2) \mid dom(\Gamma_1(c)) \cap dom(\Gamma_2(c)) \neq \emptyset\} \\
conflicttype(\Gamma_1, \Gamma_2) &= \{c \in dom(\Gamma_1) \cap dom(\Gamma_2) \mid \\
&\qquad \exists x \in dom(\Gamma_1(c)) \cap dom(\Gamma_2(c)).\ \Gamma_1(c)(x) \neq \Gamma_2(c)(x)\} \\
conflict(\Gamma_1, \Delta_1, \Gamma_2, \Delta_2) &= \bigcup \left\{ \begin{array}{l} conflictpresence(Sig(\Delta_1), Sig(\Delta_2)), conflicttype(\Gamma_1, \Gamma_2), \\ conflicttype(\Gamma_1, Sig(\Delta_2)), conflicttype(Sig(\Delta_1), \Gamma_2) \end{array} \right\}
\end{aligned}$$

Module compatibility and composition:

$$\frac{\nu' = \bigcup_{x \neq y} conflict(\Gamma_x, \Delta_x, \Gamma_y, \Delta_y) \qquad \nu = \nu_1 \cap \ldots \cap \nu_n \quad \nu \backslash \nu' \neq \emptyset}{\div\{(\nu_1, \Gamma_1, \Delta_1), \ldots, (\nu_n, \Gamma_n, \Delta_n)\}}$$

$$\frac{\begin{array}{c} \nu' = \nu_1 \cap \nu_2 \setminus conflict(\Gamma_1, \Delta_1, \Gamma_2, \Delta_2) \\ \Gamma'(c) = \Gamma_1(c) \cup \Gamma_2(c) \setminus (sig(\Delta_1(c)) \cup sig(\Delta_2(c))) \\ \Delta'(c) = \Delta_1(c) \cup \Delta_2(c) \end{array}}{(\nu_1, \Gamma_1, \Delta_1) \bullet (\nu_2, \Gamma_2, \Delta_2) = (\nu', \Gamma', \Delta')}$$

Figure 4: Module system M^ν with inner-module variability.

from) configurations. It also leaves open the question of an efficient implementation, since the formal definitions quantify over (possibly infinite) sets of configurations. We will describe an efficient implementation strategy later in Section 5.1.

From the (countably infinite) set of names of configuration options F, we can derive all possible configurations ($c \in 2^F$). Of those, a variability model describes the subset of intended valid configurations ($\nu \subseteq 2^F$). A module is a 3-tuple (ν, Γ, Δ) that consists of a variability model ν, imported function signatures Γ, and defined functions Δ. When considering variability, a function may be imported only in a subset of all configurations or may even be imported with different types in different configurations. Hence, we model imports as a partial map from configurations and function names to types ($\Gamma \in C \to X \to T$). This model ensures the invariant that, in each configuration, each name is mapped to at most one type. Similarly, we model function definitions as map from configurations and function names to expressions with corresponding type declarations.

Despite variability, type checking ($m\ \mathsf{OK}$) is still modular. Reusing the formalism of the module system M without variability, we check that function definitions are well-typed

and do not overlap with imports *in all valid configurations* described by the variability model ν. Furthermore, we assert that for each valid configuration the partial map of imports and definitions is well-defined ($\nu \subseteq \mathit{dom}(\Gamma)$). Finally, we expect that the variability model describes at least a single valid configuration ($\nu \neq \emptyset$, called 'model consistency' in [17, 37]) – otherwise module compatibility would be trivial.[2]

Based on well-typed modules, we define module compatibility and module composition. There are actually different designs of compatibility and composition possible. Here, we first introduce a notion with some resemblance to type inference: We infer a variability model that describes valid configurations and only report an error when no valid configurations remain. In Section 4.1, we discuss alternative designs and their benefits and drawbacks.

Modules are incompatible if their variability models do not share a single configuration ($\nu_1 \cap \ldots \cap \nu_n = \emptyset$). In addition, modules are incompatible if *all shared configurations* contain a function conflict (as in M, two modules define the same function, two modules import the same function but with different types, or one module defines a function imported by another module but with different types). Auxiliary function *conflict* returns the set of configurations containing a function conflict. In this design, we allow conflicts in some configurations, as long as not all configurations are affected. Furthermore, pairwise checking of compatibility is not sufficient to guarantee preservation of compatibility under composition (P6), because incompatibilities, say, due to a mutual exclusion property asserted by one module, only show up when considering the compatibility of all modules to be composed (see discussion in Section 4.1). Hence, we model compatibility as predicate on a set of modules ($\div\{m_1, \ldots, m_n\}$).

Composing two compatible modules yields a new module. The new module contains the common configurations of both modules, excluding configurations that contain function conflicts ($\nu' = \nu_1 \cap \nu_2 \setminus \mathit{conflict}(\Gamma_1, \Delta_1, \Gamma_2, \Delta_2)$; we essentially add additional constraints to the variability model for function conflicts). Imports are merged but reduced by the corresponding function definitions for each valid configuration separately. The exclusion of conflicting configurations from the new variability model ν' ensures that the partial mappings of imports and function definitions are well-defined on the full variability model ν'.

Our module system M^ν with variability preserves properties (P1)–(P5) of the module system M. Since pairwise compatibility is not sufficient to preserve compatibility, as argued above, we relax (P6) to (P6'):

(P6') Module compatibility is closed under module composition
($\div\{m_1, m_2, \ldots, m_n\} \Rightarrow \div\{m_1 \bullet m_2, \ldots, m_n\}$).

Note that (P6') is compatible with an open-world assumption because a composed module can still be composed with arbitrary other modules (provided that they are compatible with the composed module).

In addition, M^ν satisfies a new property *configuration preserves typing*:

[2] Requiring a single valid configuration is merely a consistency check. Asserting that a module provides specific configurations can be checked at composition-language level, see Section 3.4.

Additional notation:

$\mathsf{i} \subseteq \mathsf{F}$ configuration-option imports
$\mathsf{j} \subseteq \mathsf{F}$ configuration-option definition
$\mathsf{m} = (\nu, \mathsf{i}, \mathsf{j}, \Gamma, \Delta) \in \mathsf{M}^{\nu\mathsf{l}}$ module

Module typing:

$$\frac{\nu \subseteq \mathit{dom}(\Gamma) \qquad \nu \subseteq \mathit{dom}(\Delta) \qquad \forall \mathsf{c} \in \nu.\ (\Gamma(\mathsf{c}), \Delta(\mathsf{c}))\ \mathsf{OK} \qquad \nu \neq \emptyset \qquad \mathsf{i} \cap \mathsf{j} = \emptyset \qquad \mathit{varmodel}(\nu) \subseteq \mathsf{i} \cup \mathsf{j} \qquad \mathit{varmap}(\nu, \Gamma) \cup \mathit{varmap}(\nu, \Delta) \subseteq \mathsf{i} \cup \mathsf{j}}{(\nu, \mathsf{i}, \mathsf{j}, \Gamma, \Delta)\ \mathsf{OK}}$$

$$\mathit{varmodel}(\nu) = \left\{\mathsf{f} \in \mathsf{F} \mid \exists \mathsf{c} \in \nu.\ (\mathsf{c} \setminus \{\mathsf{f}\}) \notin \nu \ \vee\ (\mathsf{c} \cup \{\mathsf{f}\}) \notin \nu\right\}$$

$$\mathit{varmap}(\nu, \Delta) = \left\{\mathsf{f} \in \mathsf{F} \mid \mathsf{f} \notin \mathit{varmodel}(\nu) \wedge \exists \mathsf{c} \in \nu.\ \Delta(\mathsf{c} \setminus \{\mathsf{f}\}) \neq \Delta(\mathsf{c} \cup \{\mathsf{f}\})\right\}$$

Module compatibility and composition:

$$\frac{\mathsf{m}_\mathsf{x} = (\nu_\mathsf{x}, \mathsf{i}_\mathsf{x}, \mathsf{j}_\mathsf{x}, \Gamma_\mathsf{x}, \Delta_\mathsf{x}) \qquad \nu' = \bigcup_{\mathsf{x} \neq \mathsf{y}} \mathit{conflict}(\Gamma_\mathsf{x}, \Delta_\mathsf{x}, \Gamma_\mathsf{y}, \Delta_\mathsf{y}) \qquad \nu = \nu_1 \cap \ldots \cap \nu_\mathsf{n} \qquad \nu \backslash \nu' \neq \emptyset \qquad \forall \mathsf{x} \neq \mathsf{y}.\ \mathsf{j}_\mathsf{x} \cap \mathsf{j}_\mathsf{y} = \emptyset}{\div\{\mathsf{m}_1, \ldots, \mathsf{m}_\mathsf{n}\}}$$

$$\frac{\mathsf{m}' = (\nu', \mathsf{i}', \mathsf{j}', \Gamma', \Delta') \qquad \nu' = \nu_1 \cap \nu_2 \setminus \mathit{conflict}(\Gamma_1, \Delta_1, \Gamma_2, \Delta_2) \qquad \Gamma'(\mathsf{c}) = \Gamma_1(\mathsf{c}) \cup \Gamma_2(\mathsf{c}) \setminus (\mathit{sig}(\Delta_1(\mathsf{c})) \cup \mathit{sig}(\Delta_2(\mathsf{c}))) \qquad \Delta'(\mathsf{c}) = \Delta_1(\mathsf{c}) \cup \Delta_2(\mathsf{c}) \qquad \mathsf{i}' = \mathsf{i}_1 \cup \mathsf{i}_2 \setminus (\mathsf{j}_1 \cup \mathsf{j}_2) \qquad \mathsf{j}' = \mathsf{j}_1 \cup \mathsf{j}_2}{(\nu_1, \mathsf{i}_1, \mathsf{j}_1, \Gamma_1, \Delta_1) \bullet (\nu_2, \mathsf{i}_2, \mathsf{j}_2, \Gamma_2, \Delta_2) = \mathsf{m}'}$$

Figure 5: Module system $\mathsf{M}^{\nu\mathsf{l}}$ extends M^{ν} with scoped configuration options.

(P7) All module configurations derivable from a well-typed module are well-typed, that is, $\forall(\nu, \Gamma, \Delta) \in \mathsf{M}^{\nu}.\ \ (\nu, \Gamma, \Delta)\ \mathsf{OK} \Rightarrow \forall \mathsf{c} \in \nu.\ (\Gamma(\mathsf{c}), \Delta(\mathsf{c}))\ \mathsf{OK}$.

Properties (P1)–(P3) and (P7) follow directly from the definition of M^{ν}. Proofs of the remaining properties can be found in the appendix.

3.3 Locality of configuration options in $\mathsf{M}^{\nu\mathsf{l}}$

So far, our module system has global configuration options. In a final step, we introduce a scoping concept that allows declaring and explicitly importing configuration options, as illustrated in our motivating example in Figure 2.

We model configuration options in direct analogy to functions: A configuration option is defined in a module. Equivalent to a function body, a configuration option can provide additional specifications, such as descriptions and defaults. Other modules can import a configuration option to use it, and referencing a configuration option as part of a presence condition is the equivalent of a function call. In line with functions, we check, in each

module separately, that only defined or imported configuration options are referenced. Similar to name clashes between functions, name clashes between configuration options can be resolved with α-renaming (see Section 3.4). As function names in our basic module system, configuration options share a global namespace; however, declarations and imports provide a means to enforce scoping of names, so modules that declare the same name are incompatible. As with functions, it does not technically matter which of two modules defines and which imports a configuration option; selecting where to place the definition is a design choice. For example, similar to bundling function definitions in separate modules as libraries, designers may decide to bundle many configuration options in one separate module.

We extend our module system to M^{vl} as specified in Figure 5. A module now additionally contains imports ($\mathsf{i} \subseteq \mathsf{F}$) and definitions ($\mathsf{j} \subseteq \mathsf{F}$) of names of configuration options. Since additional description or defaults of configuration options are relevant only for external concerns, we omit them from our formalization. Locally, we check that a configuration option is not both imported and declared ($\mathsf{i} \cap \mathsf{j} = \emptyset$). Furthermore, variability in the variability model, in function imports, and in function declarations must be expressed only in terms of declared or imported configuration options. Auxiliary function *varmodel* yields all configuration options that affect the variability model and *varmap* yields configuration options that make a difference in the definition of a variable mapping (a configuration option f makes a difference if and only if two otherwise equal configurations with and without f are distinguished by a model or mapping). This way, we enforce well-defined scoping of configuration options. In a practical implementation, in which sets of configuration options are represented by propositional formulas, these checks can be conservatively approximated by considering the set of configuration options that occur syntactically in presence conditions in the module.

Compatibility and composition require only minimal, straightforward extensions: Two modules are incompatible if they declare the same configuration option. During composition, declarations and imports of configuration options are matched and merged like functions. M^{vl} also preserves properties (P1)–(P5), (P6'), and (P7), as proved in the appendix.

3.4 Composition language

So far, we have discussed the module-composition operator ($\bullet : \mathsf{M} \times \mathsf{M} \to \mathsf{M}$) and module compatibility. There are additional useful operators at the level of the composition language, such as renaming, hiding, partial configuration, and variability checking. Here, we outline useful operators to give a more complete picture of typical and flexible module composition. We have the following syntax of module expressions in the module-composition language on top of M^{vl}:

```
Z ::= M^vl                          atomic modules
    | Z • Z                         composed modules
    | closed Z                      completeness check
    | rename X → X in Z             function renaming
    | renameC F → F in Z            configuration-option renaming
    | hide X in Z                   function hiding
    | configure F → {⊤,⊥} in Z      partial configuration
```

```
| assert V in Z              variability check
```

Since the operators and their formalization are straightforward, we provide only an intuition of how they work.

A module is *closed* if it has no remaining imports (function imports or configuration-option imports) in any configuration. The operation *closed* returns a closed module unmodified and gets stuck on modules that are not closed. A module check is easy to specify and implement by inspecting Γ and $\mathtt{i}$ of the module.

Operation *rename* takes a module and produces a new module in which all occurrences of a function name (in function imports, function definitions, and function calls in all configurations) are replaced by a different name. As precondition, we expect a well-typed module in which the new function name is not already imported or defined. For example to compose *inmem* and *persist* in Figure 1, we could rename functions *write* to *writemem* and *read* to *readmem*: `inmem • (rename read→readmem in (rename write→writemem in persist))`.

Similarly, operation *renameC* renames all occurrences of a configuration option. The operation assumes a well-typed module and a target name that is not yet imported or defined as configuration option inside the module. Technically, we simply exchange the names in configurations during lookups; in a more syntactic implementation, we would rewrite variables in presence conditions. For example, in the source code in Figure 2, we could simply replace all syntactic occurrences of READONLY by DB_READONLY_ACCESS to avoid possible name clashes with other modules that also have a configuration option READONLY.

Operation *hide* hides a function inside a module so that it is no longer exported. The notion of hiding is especially useful in hierarchical module systems [11]. We can either explicitly model private functions, or we implement hiding by inlining the function. For example, after composing the modules *core* and *inmem* in Figure 1, we could hide functions *read* and *write* to clean the namespace before further compositions.

Operation *configure* removes a configuration option from a well-typed module, by selecting or deselecting it. As discussed previously, there are no private configuration options, but every configuration option must be exported to enable a choice. Therefore, we cannot hide a configuration option without deciding whether the corresponding code should be included or not. Syntactically, this operation replaces all occurrences of the configuration option in presence conditions by *true* or *false* and removes the corresponding declaration. For example, we could decide to select feature READONLY of module *storage* in Figure 2 (note that on subsequent composition of that module with *query*, the imported feature READONLY would no longer be matched by a corresponding definition, that is, the resulting module is not closed).

Finally, we provide a *variability check* for a module that asserts that the given module provides expected variability (somewhat similar to a type cast). The operation simply returns the module if the expected variability (provided as a variability model) is a subset of the module's variability model, or gets stuck otherwise. So, we can compare a composed module with a separately defined specification, as we discuss in Section 4.2.

Based on this composition language and the formalization of M^{vl}, we could define a type system that statically checks that a composition does not get stuck. However, such

a type system adds little new to our discussion of variability, so it is outside the scope of this paper.

3.5 Formalization summary

We have shown that it is possible to make modules variability-aware while preserving the basic properties of traditional module systems. To do so, we replaced the globals of traditional feature-oriented programming – variability model and scope of configuration options – by modular counterparts and enriched the interface language with variability, such that separate checking becomes possible.

4 Design decisions

The variability-aware module system we defined in the previous section makes several design decisions that deserve discussion.

4.1 Constraint inference

The most controversial design decision of our calculus is to *infer* constraints during composition when function conflicts are detected. In our calculus, two modules are compatible even if they contain function conflicts, as long as at least one configuration is without conflict.

As alternative design, we could regard two modules as incompatible, if they have function conflicts in *any configuration*. We would define compatibility as follows:

$$\frac{\nu_1 \cap \nu_2 \cap \mathit{conflict}(\Gamma_1, \Delta_1, \Gamma_2, \Delta_2) = \emptyset \qquad \mathfrak{j}_1 \cap \mathfrak{j}_2 = \emptyset}{(\nu_1, \mathfrak{i}_1, \mathfrak{j}_1, \Gamma_1, \Delta_1) \div (\nu_2, \mathfrak{i}_2, \mathfrak{j}_2, \Gamma_2, \Delta_2)}$$

$$\frac{\bigwedge_{x \neq y} \mathfrak{m}_x \div \mathfrak{m}_y}{\div\{\mathfrak{m}_1, \ldots, \mathfrak{m}_n\}}$$

A main difference between both designs is associativity of module composition (P5). Consider the following modules:

```
a = (                          b = (                          vm = (
  config A;                      config B;                      import config A,B;
  fun foo(): Int                 fun foo(): Int                 variability
    if A = ...;                    if B = ...;                    ¬(A∧B);
)                              )                              )
```

Modules a and b export the same function in overlapping configuration sets, but module vm excludes all overlapping configurations. In an open-world scenario, modules a and b do not know about each other or their configuration options. Nevertheless, the inference-based design decision allows us to compose modules a and b without knowing about a dependency between configuration options A and B; the composition operator infers that A and B must be mutually exclusive. In the alternative design, in which we do not allow any function conflicts, we can compose a with b only *after* composing one of it with vm (i.e., $a \bullet (b \bullet vm)$ is a valid composition, whereas $(a \bullet b) \bullet vm$ is undefined).

In the inference-based design, the composition operator infers additional constraints on the feature model. If needed, we can use the *assert* operator of the composition language to ensure that we do not accidentally restrict the module's variability model too much (see Section 3.4). In contrast, in the alternative design, a developer is forced to compose one module with a glue-code module *before* composing it with another module with partial function conflicts. Along those lines, the variability model to be used as glue code can be integrated into the composition operator, such as $\mathfrak{m}_1 \bullet_\nu \mathfrak{m}_2$ as shorthand for $(\mathfrak{m}_1 \bullet (\nu, \emptyset, \emptyset, \emptyset, \emptyset)) \bullet \mathfrak{m}_2$.

There are trade-offs between both designs:

- *Associativity vs. pairwise compatibility:* On the one hand, the inference-based design enables associativity of module composition (P5). On the other hand, in the alternative design, already *pairwise* module compatibility is closed under module composition; thus it satisfies the stronger property (P6) in addition to (P6').

- *Local errors vs. specification effort:* In the alternative design, function conflicts are always reported locally when composing two modules. When these function conflicts do not matter due to additional constraints, the developer must provide additional specifications at composition time. Conceptually, the alternative design roughly relates to *explicit type annotations* for type checking, where precise local error messages are possible at the expense of additional specification effort. In contrast, the inference-based design roughly aligns with *type inference*, because we infer which compositions are correct but only report an error when we actually use one of the excluded configurations. As in languages based on type inference, errors are reported less immediate and less local, but less specifications are required.

Both designs have their merits. For us, flexible, associative composition (P5, P6') was the more important goal, so we decided to present and implement the inference-based design as main mechanisms. Furthermore, our experimental evaluation suggests that compatibility in the inference-based design is not trivial and can find type errors in real-world code. Nevertheless, the alternative design is straightforward to formalize (actually, the different compatibility rule above is the only necessary change) and to implement.

Finally, there is a third alternative that would allow both associativity (P5) and pairwise compatibility checks (P6): We could restrict variability models such that only positive constraints can be expressed, for example, by restricting constraints to Horn clauses. Unfortunately, that design choice reduces expressiveness beyond what is acceptable in product-line practice: We could not even express mutual exclusion as in module *vm*.

4.2 Local variability models and configuration-option imports

One of our design goals was to eliminate the inherently antimodular global variability model, which is common in product line engineering [17]. A global variability model does not align with the open-world design of our module system. Instead, we allow specifying the relevant constraints distributedly in different modules. Thereby, our variability-aware module system allows decomposing large global variability models into small local

variability models. We believe that modules with local variability models can be more easily reused, because local variability models make weaker assumptions on the context.

If desired, a global variability model can still be encoded as just another module. This module would declare all configuration options and their constraints. The pattern of having a separate variability model may be useful for the common case that a domain expert models constraints not reflected or detected by the type system, such as "a read-only database does not require transactions". Such a additional variability model can simply be linked into any other module to restrict valid configurations. Optionally, we could extend our module system such that a module can specify *expected* variability in form of a minimal configuration space that should not be restricted by other modules; this can also be encoded with the *assumes* operator of the composition language (cf. Section 3.4).

We could criticize that we still have a global namespace for configuration options. While this is true, the same holds for the namespace of function names. In both cases, we enforce scoping with explicit imports and compatibility checks detect accidental re-definitions. Furthermore, renaming operations of the composition language can be used to resolve naming conflicts.

An arguable aspect of our design is that we need to locally redeclare constraints between crosscutting configuration options in every module that needs those constraints for modular type checking. This could easily be addressed by adding named imports in which lists of constraints (and function signatures) or entire modules can be imported with a single import statement. In principle, we could also infer a local variability model that describes exactly all well-typed configurations (similar to how we infer constraints during linking), but we prefer immediate modular error reporting. Arguably, repeating constraints for modular type checking provides even useful documentation.

4.3 Abstraction from variability implementation

Our calculus abstracts from a concrete language and type system at expression level. We intentionally focus on module interfaces to allow different inner implementation approaches and different strategies to type check all configurations of a module's implementation.

There are many examples of how variability inside a module can be implemented and type-checked.

- Conditional compilation as introduced in Section 2, even though usually frowned upon from the research community [47], is a perfect match for our calculus: Developers can encode presence conditions on code fragments with #ifdef directives. Even variability at expression level is not uncommon in practice [34]. Variability-aware type checking [28] can be used to type check all configurations efficiently as we will show. Our implementation for C, discussed in Section 5.2, is entirely based conditional compilation in C code. Although we do not want to encourage using lexical preprocessors, we acknowledge their widespread use and the huge amount of legacy code and provide corresponding tool support.

- We can use a module-per-configuration-option approach (without variability inside modules; cf. Section 2.1) to encode variability *inside* a composite module [52]. Several mechanisms, called safe composition, can be used to efficiently check whether all compositions of a fixed set of inner modules allowed by a local variability model are well-typed [50, 20, 3, 15]. In our module system, we can nest even variable modules and guarantee a common interface.

- Finally, we can use any other implementation strategy, including runtime variability [44], sophisticated metaprogramming systems [26, 31], and configuration management systems [10, 32]. For most of these implementation mechanisms, no efficient means to type check all configurations is available yet. However, if variability in each module is sufficiently restricted, a brute-force approach of checking each distinct implementation of the module's configurations may sufficiently scale. Using our module system, we can apply brute-force type checking to each module in isolation, whereas, once we determined that a module is well-typed, there is no need to recheck it for composition (P3).

4.4 Product lines of product lines

Developing product lines of product lines (also known as nested product lines [32], multi product lines [45], or product populations [52]) has received increasing attention as the size of industrial product lines has grown and the need for a divide-and-conquer strategy arose again. Since each module can be considered as a product line of its own, composing multiple product lines and reusing product lines in different (even unplanned) contexts is a natural use case of our module system. For example, we could reuse the storage-subsystem product line from Figure 2 in a product line of consumer electronics. Our module system offers a clean solution to decompose a product line into smaller subproduct lines, including a suitable decomposition of the variability model, enforcing information hiding with variable interfaces.

In this context, it is useful to adopt the notion of hierarchical modularity [11] and provide a richer composition language as outlined in Section 3.4. Supporting composition in hierarchical form allows resolving possible composition conflicts locally, at lower levels of the hierarchy. At each level, developers control what functions and variability the composed module exposes. To that end, renaming, hiding exported functions after composition and partially configuring a module by selecting or deselecting configuration options become essential operations to prepare modules for composition with independently developed product lines. We believe that most concepts of the SML/NJ compilation manager [11] can be adopted for product lines in our module system as well; but an in-depth analysis is outside the scope of this work.

5 Implementation and practical scenario

We demonstrate a practical application of our variability-aware module system as follows. First, we outline an implementation strategy that is sufficiently efficient for real-world

product-line implementations. Second, we actually implemented a variant of the module system for C code with #ifdef variability. Third, we apply our implementation to the medium-size product line Busybox. We do not intend to perform rigorous benchmarks; instead, we demonstrate that it is possible to implement such a module system *efficiently*, and we illustrate *practical* potential of the module system in a realistic setting.

5.1 Implementation strategy

The calculus leaves open how to represent variability and describes checks by quantifying over large configuration spaces (e.g., $\forall \mathsf{c} \in \nu.\ \ldots$). In our implementation, we encode sets of configurations as propositional formulas ($\mathsf{p} \in \mathsf{P}$), in which variables are names of configuration options, as exemplified already in Section 2.3. Each model of the formula corresponds to a configuration. This allows us to encode module well-formedness and compatibility as Boolean satisfiability problem. We describe an encoding in line with a long tradition of prior work on variability-aware analysis [5, 18, 50, 3, 28, 29, 49, 15].

Despite exponential worst-case time, reasoning about all configurations induced by a formula is sufficiently efficient in practice with modern Boolean satisfiability solvers [37]. An empty configuration set corresponds to an unsatisfiable formula ($[\![\mathit{false}]\!] = \emptyset$), the intersection of two configuration sets is equivalent to the conjunction of the corresponding formula ($[\![\mathsf{p}_1 \wedge \mathsf{p}_2]\!] = \nu_1 \cap \nu_2$), and so forth.

We encode the map from configurations and names to types ($\Gamma \in \mathsf{C} \to \mathsf{X} \to \mathsf{T}$) as a map from names and formulas to types ($\Upsilon \in \mathsf{X} \to \mathsf{P} \to \mathsf{T}$). This has two benefits: We can iterate over a typically small set of formulas describing only distinct types, and, due to the reversed mapping order, we do not need to copy the entire environment when changing a single function. In this encoding we need to enforce the invariant that all formulas for a name are mutually exclusive with a SAT solver. As optimization, two entries with the same name pointing to the same type can be combined by disjuncting their formulas.

Module compatibility. To determine compatibility between module interfaces, we check whether there is at least one satisfiable configuration that satisfies both variability models and is not a function conflict: $\mathsf{SAT}(\mathsf{p}_1 \wedge \mathsf{p}_2 \wedge \neg \mathit{conflict}(\ldots))$. To determine function conflicts, we derive a formula that describes all conflicting configurations. Let us illustrate this encoding with *conflictpresence*: For a name x, we determine the condition when x is exported with any type ($\bigvee \mathit{dom}(\Upsilon_\mathsf{i}(\mathsf{x}))$); subsequently, we require exports from both modules must be mutually exclusive ($\neg\big(\bigvee \mathit{dom}(\Upsilon_1(\mathsf{x})) \ \wedge \ \bigvee \mathit{dom}(\Upsilon_2(\mathsf{x}))\big)$); finally, we return the disjunction of these mutually-exclusive constraints for all x defined in both modules. That is, to determine *conflictpresence*, we iterate over a small set of names and with a small set of formulas per name to create a single formula describing all configurations with conflicts. We encode *conflicttype* similarly, but additionally compare (a usually small number of) types. All checks are performed solely on interfaces (P3).

Module composition. During module composition, we create a new variability model as conjunction of the original ones without conflicts ($\mathsf{p}_1 \wedge \mathsf{p}_2 \wedge \neg \mathit{conflict}(\ldots)$). When

both modules import the same function with the same type, we import it only once using the disjunction of the respective presence conditions. To remove a function import with formula $\mathtt{a}$ by an export with formula $\mathtt{b}$, the resulting module imports the function with formula $\mathtt{a} \wedge \neg \mathtt{b}$. Finally, all entries with formulas $\mathtt{a}$ that are unsatisfiable in the resulting variability model $\mathtt{p}'$ (i.e., $\neg \mathsf{SAT}(\mathtt{p}' \wedge \mathtt{a_i})$) can be removed.

Type checking all configurations of a module. As discussed in Section 4.3, many different implementation mechanisms can be used inside a module; even a brute-force approach to type check all configurations may be feasible in some cases. Still, more sophisticated checks have been developed for certain variability-implementation approaches [5, 18, 50, 3, 28, 20, 15]. Here, we briefly outline how to type check code with conditional compilation.

With variability, each expression can have alternative types, just as each name in the current context can have alternative types. All expressions are type checked in a variability context $\mathtt{p}_{ctx}$ (a formula describing the subset of configurations that are checked, for example, the presence condition of the function that contains the expression). When looking up a function call, we find all declared types $\mathtt{t_i}$ with the corresponding formulas $\mathtt{p_i}$. We can discard types with formulas never satisfiable in the current context ($\neg \mathsf{SAT}(\mathtt{p}_{ctx} \wedge \mathtt{p_i})$). We raise an error if, in any configuration in the context, there is no type ($\mathsf{SAT}(\mathtt{p}_{ctx} \wedge \neg \mathtt{p}_1 \wedge \ldots \wedge \neg \mathtt{p_n})$) and hence no function; a violation of a property that we call *reachability* [28]. Again, the key idea is using propositional formulas to reason about (typically few) alternative types instead of iterating over all configurations.

For operations that involve comparing two types, such as function application (e.g., $\mathtt{e}_1\mathtt{(e}_2\mathtt{)}$), we look up alternative types for both subexpressions. We check all combinations of both alternative types, if the conjunction of their formulas is satisfiable in the current context. Worst-case effort is exponential and the number of types can explode, but, in practice, both expressions have only few alternative types [3, 5, 15]. For more details on variability-aware type checking see the rich body of prior work [5, 50, 3, 28, 20, 15].

We determine relevant configuration options (*varmodel*, *varmap* in M^{vl}) syntactically with a sound and conservative approximation (all properties still hold): We collect all variable names in formulas, including the variability model.

Due to SAT solving, determining compatibility and well-typedness of modules is NP-complete. However, with modern SAT solvers, the complexity of SAT solving is not of practical concern even for large product lines [37, 28, 50].

5.2 Implementation for C

We provide the first approach that can type check all configurations of realistic C code, beyond actually preprocessing and checking all configurations in isolation in a brute-force fashion (prior work focused on dialects of Java [50, 3, 28, 20], the lambda calculus [15], and UML [18], or only sketched a possible strategy [5]). Our implementation of the variability-aware module system supports both modular checking of all configurations of a translation unit and variability-aware compatibility checks. The implementation is part of the *TypeChef* project, which pursues variability-aware analysis of real-world C code.

TypeChef is open source and available at `http://ckaestne.github.com/TypeChef/`. For experimentation, a simple interactive online version is available at that site as well.

Based on the outlined implementation strategy, we implemented the variability-aware module system M^{vl} for C. Instead of modifying a C compiler and linker, we wrote the module system as a separate analysis tool in Scala. It separately detects errors that the normal compiler and linker would find when compiling and composing files in a specific configuration. The implementation consists of four main parts: parsing, modular variability-aware type checking of translation units, interface inference, and composition checks between interfaces.

Variability-aware parsing. A challenge in analyzing #ifdef variability in C code, which hampered prior approaches, is preserving variability during parsing. Conventional C parsers only parse a single configuration after the preprocessor has inlined includes, expanded macros, and evaluated conditional-compilation directives. Instead, we parse C code without evaluating #ifdef directives and produce an abstract syntax tree that contains variability information (including information from header files). In case of #ifdef directives, we parse both branches and encode variability in the abstract syntax tree. Typically only explicitly declared configuration options are considered for variability, whereas other macros, such as included guards, are processed as in a traditional preprocessor. The actual process is precise but much more complex, due to lexical use of #ifdef directives on arbitrary tokens and because of interactions between macros, includes, and conditional compilation. The parser has been discussed in detail in prior work [29]; here, we use it as black-box component.

Modular variability-aware type checking. For each translation unit, we perform modular variability-aware type checking ($\mathfrak{m}$ OK of M^{vl}) on the abstract syntax tree with variability, as outlined above in Section 5.1. The type system determines (alternative) types for all expressions. In C, this means it checks reachability (as described in Section 5.1) of function calls and variables access, reachability of field access of structures, and compatibility of types. In principle, a sound and complete variability-aware type system is possible [28, 3, 20, 15]. However, due to the size and the informal description of the C standard, our prototype covers only a large subset of the standard but is incomplete and unsound regarding, for example, goto labels, unreachable-code removal, and several GNU C extensions. The type system incorporates a local variability model (defined in a separate file for each translation unit) and reports errors only within valid configurations. It checks each translation unit in isolation (P2).

Interface inference. Based on the type system's result, we infer an interface for each translation unit. C fits particularly well to our module-system design, because it distinguishes between function declarations without bodies (prototypes; typically defined in header files) and function definitions with bodies. Function definitions are exported unless marked *static*, whereas called functions that are declared but not defined are imported. From the presence conditions of function definitions and function calls, we

derive presence conditions for the interface; for imports, we derive a presence condition as disjunction of all presence conditions of calls of this function within the translation unit.[3] Types of imports and exports are directly recognized from function declarations and function definitions, respectively (i.e., no type inference and no investigation of other modules is necessary). Furthermore, the interface contains imports for all configuration options used within presence conditions in the translation unit. We do not automatically infer declarations of configuration options or local variability models, but users can define them manually if desired.

We decided to infer interfaces instead of writing them explicitly, because, except for declarations of configuration options and variability model, all information is already available in the C code. Developers can decide when to import a function by adding an #ifdef around the prototype declaration and can explicitly decide when to export a function with the *static* specifier, which can also be guarded by an #ifdef. Maintaining a separate manual interface specification and checking it against the implementation is possible, but does not provide additional benefits: An interface cannot be more restrictive than the implementation, unless we change the C compiler to enforce interfaces as well.

Compatibility checks. Compatibility checks between inferred interfaces ($\div\{\mathfrak{m}_1, \ldots, \mathfrak{m}_n\}$) implement $M^{\nu l}$ as outlined in Section 5.1. Our implementation works on inferred interfaces in separate files, not directly on C code.

5.3 Type checking Busybox

Finally, we used our variability-aware module system to parse all translation units in Busybox, type check each translation unit in isolation, infer interfaces, and check the composition of these interfaces.

Errors. In the analyzed release 0.18.5, all 522 translation units are syntactically correct and well-typed in all valid configurations. The development process of Busybox, which includes some random-configuration testing before releases, seems to catch most type errors already. However, in recent development revisions, we found and reported three compiler errors specific to certain configurations.[4] In addition, occasionally compiler errors in development revisions are reported on the mailing list; we reproduced some known (and now fixed) compiler errors throughout the revision history. Here, we exemplify a type problem and a linker problem.

In September 2011, a user reported compiler error of undeclared variables *now* and *info* in file *procps/ps.c*.[5] After some investigation, the user eventually traced down the

[3] In fact, the GNU C compiler creates symbols only for functions called after the optimizer removed unreachable code. We do not yet perform such optimizations; so, functions called only from unreachable code are part of the inferred interface. Adopting constant folding and static analysis to detect unreachable code in all configurations is an interesting avenue for future work.

[4] Bug reports `https://bugs.busybox.net/show_bug.cgi?id=4994` and `http://lists.busybox.net/pipermail/busybox/2012-April/077683.html`; fixed in subsequent commits.

[5] See `http://lists.busybox.net/pipermail/busybox/2011-September/076730.html` for the full discussion and patch.

problem to a configuration without feature FEATURE_PS_LONG and posted a configuration that would reproduce the error. The patch that fixed the problem adds an additional #ifdef directive around the problematic code fragment. Running TypeChef on the revision at the time of the bug report (git commit *b64bd16459*) yields two type errors in file *procps/ps.c*. In contrast to the manual investigation, TypeChef pinpoints the problem precisely to a set of configurations with the following constraint: $\neg$DESKTOP $\wedge$ PS $\wedge \neg$FEATURE_PS_LONG $\wedge$ (SELINUX $\vee$ FEATURE_SHOW_THREADS $\vee$ FEATURE_PS_WIDE).

In the same month, another user provided a patch for a linker error.[6] An incorrectly placed #ifdef (introduced in git commit *128543721*) caused that library function *match_fstype* was no longer exported (instead of being exported when feature PLATFORM_LINUX is selected). At the same time, the function was still imported in modules *mount* and *umount*, when the corresponding features MOUNT or UMOUNT were selected. In that revision, TypeChef reports that the composed module still conditionally imports function *match_fstype*; that is, the module is not closed in configurations with MOUNT $\vee$ UMOUNT.

For other kinds of linker errors, such as conflicting types of imports, multiple function exports with the same name, and type mismatch between imports and exports, we have not found actual instances in Busybox. For testing purposes, we deliberately introduced and detected several of them. Overall, our experiments confirm that TypeChef finds type errors and linker errors in real-world product lines, which is especially helpful as rapid feedback during the development process, for instance as part of an automated build in a continuous-integration process.

Local and crosscutting variability. Our module system supports both local and crosscutting variability. In Section 2.4, we presented several metrics from Busybox that were gathered ex post from our infrastructure. When taking the module-per-configuration-option approach, we would have been forced to decompose translation units with local inner variability into smaller modules, just for technical reasons. Furthermore, we would have been forced to create many additional modules for configuration options that crosscut the entire implementation. Such encoding appears cumbersome and unpractical, whereas our module system allows modular checks without restructuring the code.

Our module system explicitly supports encapsulating local inner variability, enables variability to crosscut multiple modules, and supports variability in interfaces. Whereas previously every configuration had to be checked in isolation in a brute-force fashion, we can type check all configurations of each module in isolation and we can check compatibility of all modules with their variability.

Performance. The advantage of modular checks shows most prominently regarding performance. In total, we need 57 minutes to type check all modules.[7] On average

[6] `http://lists.busybox.net/pipermail/busybox/2011-September/076576.html`

[7] We measured performance on a normal lab computer (Intel quad-core 3.4 GHz with 8 GB RAM; Linux; Java 1.6, OpenJDK). We did not perform low-level optimizations and still compute debug

we need 5 seconds to parse a single translation unit with all its headers and with its variability, 0.7 seconds to type check all configurations in a variability-aware fashion, and 0.03 seconds to infer its interface. Compared to a brute-force approach of checking all configurations in isolation, our analysis is extremely fast. The slow down compared to conventional compilers is a tribute to the inherent complexity (we parse and type check all, potentially billions of possible configurations) and the necessity to solve many Boolean satisfiability problems. With our module system, we easily parallelize type checking with multiple machines. Furthermore, after a change, we only need to recheck affected files and corresponding compatibility checks instead of reperforming whole-program analysis.

Checking compatibility and composing all interfaces incrementally (in alphabetical order) requires 29 seconds. Composing them in a divide and conquer fashion (pairwise composition, then pairwise composition of the results, and so on) reduces effort to 4 seconds, with additional potential for parallelization. Overall, the opportunity for quick compatibility checks, for parallelization, and for incremental checking allows us to scale variability-aware analysis to real-world C code.

6 Related work

Variable module composition. Product-line implementations that target some notion of modularity (e.g., components, plug-ins, feature modules, functors, or aspects) typically follow an approach in which compile-time variability is expressed during composition, not inside modules [14, 42, 7, 26, 2, 46, 25, 17, 43]. Several of these approaches offer notions of function refinement, orthogonal to our discussion, for which composition is not commutative [4]. When constructing product lines from modules without inner variability, type checking each module in isolation is usually straightforward; for languages without explicit module interfaces, such as AHEAD, AspectJ, and DeltaJ, corresponding interfaces can be inferred [20, 46, 33]. However, since variability is encoded as variable composition, there is still an exponential number of possible configurations. Checking them all is usually infeasible; modular type checking reduces the costs for each composition check but does not reduce their number.

For a *fixed set* of modules, *safe composition* explores all configurations against a *global* variability model using an encoding as Boolean satisfiability problem [50, 20, 3]. The same technique was explored also as *variability-aware type checking* for closed-world nonmodular implementations [18, 28, 5, 15] and for other analysis approaches [16, 12]. Although following a different technical route, the implementation of our type-checking mechanism inside modules with alternative types was particularly inspired by the structures of the choice calculus [23, 15]. Overall, in our module system, we use similar algorithms, but a closed world assumption is never required; an existing (composed) module can always be composed with more modules while retaining the soundness guarantees.

information and statistics. Measured times provide only rough indicators about what performance to expect and that variability-aware analysis is feasible; they are not meant as rigorous benchmarks.

Variability inside modules. Variability *inside* modules has been explored in different contexts. Our work was initially inspired by prior work on modular logic metaprogramming [31]. In logic metaprogramming, programs are derived from a deductive database; by using a logic to describe the effect of metaprograms in interfaces, sound modular type checking can be achieved. We adopted the underlying idea of logic formulas in interfaces, but restricted and specialized it to a level that is practical for large-scale product line development and efficient to check with automated provers. From the perspective of logic metaprograming, we reduce metaprogramming to propositional presence conditions over configuration options and local variability models.

Several programming languages support some form of type-conditional methods, a form of parametric polymorphism in which the applicability of a method call can depend on the type parameter of the enclosing class [27, 35, 40, 21]. Invoking a conditional method is only well-typed when the condition is satisfied in the context of the invocation. For example, in a collection class, such as List, clients should only be allowed to invoke a method print if the class is parameterized with a type that can be printed. The purpose of type-conditional methods is to improve static type-safety; the operational semantics of the language does not change. For instance, it is not possible to define several alternative variants of a method (with different implementations or types) or to define dependencies between configuration options. This means that the applicability of these approaches to variability management for software product lines is rather limited.

Approaches to increase the flexibility of method dispatch, such as multi-methods [38], predicate dispatch [22], or dependent classes [24] could be used for modules with inner variability, but since the dispatch only depends on method arguments, it is not obvious how to encode variability that can not be deduced from the dynamic arguments of a method call. Furthermore, a set of methods with the same name typically needs a default implementation, which is called if none of the other methods is applicable, which is less safe than the checks in our approach, which do not need default implementations but can detect statically when no applicable function exists. Also, modular type checking of these approaches is quite hard [38, 39]. On the other hand, these approaches are much more powerful with regard to expressing dynamic variability, which is not in the scope of this work.

Compile-time metaprogramming, such as C++ templates or the C preprocessor, is often used to express inner compile-time variability [17], but these approaches suffer from the problem that type checking can only take place after specialization to a specific instance of the product line.

Also the product-line community has explored components with inner variability. Most prominently, the Koala component system has mechanisms for run-time and compile-time variability inside a module, exposed through a diversity interface [52]. A Koala module can express variable module composition of inner modules; the condition for the composition can be exposed in the diversity interface. If a configuration parameter is known at compile-time, only the corresponding inner module is included (a specialized form of partial evaluation), otherwise all modules are included and function calls are dispatched at run-time. In contrast to our module system, Koala does not support variability in the functional interface: Diversity interfaces may change the behavior (and which inner

component is used to provide the behavior) but not the interface. Dependencies between configuration options and crosscutting variability are not explicitly supported, but can be encoded. Since compile-time variability is expressed with variable module composition (possibly nested inside another module), Koala enforces a variability-induced dominant decomposition.

Along similar lines, de Jonge [19] introduced configuration interfaces into a package mechanism: Each package can declare configuration options and bind them in imported packages. Similarly, plastic partial components [41] introduce variability interfaces for architectural components and realize variability internally with aspect-oriented programming. Van der Storm [51] subsequently extended this approach with local variability models and configuration checks by encoding variability information as Boolean satisfiability problem. However, all these approaches do not enforce modularity of the host language modules with code-level interfaces; at most they check consistency between packages.

Composing variability models. Finally, there are many mechanisms to specify and compose variability models and to reason about them. In practice, some flavor of graphical feature diagrams are typically used [17], which represent configuration options in a hierarchical form and have a straightforward translation to propositional logic [51]. Busybox uses the textual feature-modeling language KConfig with a similar concept and translation [9]. Advanced composition mechanisms attempt to retain the hierarchical form of variability models [1]; they are orthogonal to our discussion. If variability model and reasoning should include non-Boolean configuration options, other logics and solvers can be used [8]. For our calculus and our implementation, composing propositional formulas was sufficient.

7 Conclusion

We introduced a variability-aware module system for software product lines that overcomes the variability-induced dominant decomposition of traditional module systems, by allowing variability inside modules and in module interfaces. Each module can be type checked in isolation, covering all configurations allowed by the module's local variability model. Composing two compatible well-typed modules with variability yields another well-typed module with the combined variability. The module system breaks with the product-line tradition of closed-world implementations with a global variability model and takes it into an open environment, toward software ecosystems and product lines of product lines. We defined the module system formally in a calculus, outlined a general implementation strategy, and presented an implementation for C, which we applied to the open source product line *Busybox*. Our next step is to type check the entire Linux kernel with 10 000 configuration options, a task for which the module system is an important foundation, but for which various engineering problems still have to be solved.

Acknowledgments. The work was inspired by Karl Klose's work on modular metaprogramming. We thank Tillmann Rendel, Sven Apel, Yannis Smarakdakis, Don Batory, Chung-chieh Shan, Martin Erwig, and Paolo G. Giarrusso for their valuable discussions on early presentations of this work. This work is supported by ERC grant #203099.

References

[1] M. Acher, P. Collet, P. Lahire, and R. France. Comparing approaches to implement feature model composition. In *Proc. European Conf. Modelling Foundations and Applications (ECMFA)*, volume 6138 of *LNCS*, pages 3–19. Springer, 2010.

[2] M. Anastasopoules and C. Gacek. Implementing product line variabilities. In *Proc. Symposium on Software Reusability (SSR)*, pages 109–117. ACM, 2001.

[3] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[4] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming (SCP)*, 75(11):1022–1047, 2010.

[5] L. Aversano, M. D. Penta, and I. D. Baxter. Handling preprocessor-conditioned declarations. In *Proc. Int'l Workshop Source Code Analysis and Manipulation (SCAM)*, pages 83–92. IEEE CS, 2002.

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, MA, 1998.

[7] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Softw. Eng. (TSE)*, 30(6):355–371, 2004.

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

[9] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 73–82. ACM, 2010.

[10] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Sci. Comput. Program.*, 53(3):333–352, 2004.

[11] M. Blume and A. W. Appel. Hierarchical modularity. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 21(4):813–847, 1999.

[12] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural dataflow analysis for software product lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, 2012. to appear.

[13] L. Cardelli. Program fragments, linking, and modularization. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 266–277. ACM, 1997.

[14] W. Chae and M. Blume. Building a family of compilers. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 307–316. IEEE CS, 2008.

[15] S. Chen, M. Erwig, and E. Walkingshaw. Extending type inference to variational programs. Technical report (draft), School of EECS, Oregon State University, 2012.

[16] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 321–330. ACM, 2011.

[17] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, New York, 2000.

[18] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM, 2006.

[19] M. de Jonge. Source tree composition. In *Proc. Int'l Conf. Software Reuse (ICSR)*, volume 2319 of *LNCS*, pages 261–282. Springer, 2002.

[20] B. Delaware, W. R. Cook, and D. Batory. Fitting the pieces together: A machine-checked model of safe composition. In *Proc. Foundations of Software Engineering (ESEC/FSE)*, pages 243–252. ACM, 2009.

[21] B. Emir, A. Kennedy, C. Russo, and D. Yu. Variance and generalized constraints for C$^{\#}$ generics. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 4067 of *LNCS*, pages 279–303. Springer, 2006.

[22] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 186–211. Springer, 1998.

[23] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(1):6:1–6:27, 2011.

[24] V. Gasiunas, M. Mezini, and K. Ostermann. Dependent classes. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, OOPSLA '07, pages 133–152. ACM, 2007.

[25] M. Griss. Implementing product-line features by composing aspects. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 271–288. Kluwer Academic Publishers, 2000.

[26] S. S. Huang and Y. Smaragdakis. Expressive and safe static reflection with MorphJ. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 79–89. ACM, 2008.

[27] S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with safe type conditions. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 185–198. ACM, 2007.

[28] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(3), 2012. in press.

[29] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM, Oct. 2011.

[30] J. Kienzle and R. Guerraoui. AOP: Does it make sense? The case of concurrency and failures. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 37–61. Springer, 2002.

[31] K. Klose and K. Ostermann. Modular logic metaprogramming. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 484–503. ACM, 2010.

[32] C. W. Krueger. New methods in software product line development. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 95–102. IEEE CS, 2006.

[33] H. C. Li, S. Krishnamurthi, and K. Fisler. Interfaces for modular feature verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 195–204. IEEE CS, 2002.

[34] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114. ACM, 2010.

[35] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*, volume 114 of *LNCS*. Springer, Berlin/Heidelberg, 1981.

[36] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 112–121. ACM, 2006.

[37] M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 231–240. ACM, 2009.

[38] T. Millstein, M. Reay, and C. Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, OOPSLA '03, pages 224–240. ACM, 2003.

[39] T. D. Millstein, C. Frost, J. Ryder, and A. Warth. Expressive and modular predicate dispatch for Java. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 31(2):7.1–7.54, 2009.

[40] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 132–145. ACM, 1997.

[41] J. Pérez, J. Díaz, C. Costa-Soria, and J. Garbajosa. Plastic partial components: A solution to support variability in architectural components. In *Proc. European Conf. Software Architecture (ECSA)*, pages 221–230. IEEE CS, 2009.

[42] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 419–443. Springer, 1997.

[43] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6:307–334, November 1986.

[44] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 445–454. ACM, 2010.

[45] M. Rosenmüller and N. Siegmund. Automating the configuration of multi software product lines. In *Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130, 2010.

[46] I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking for delta-oriented programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 43–56. ACM, 2011.

[47] H. Spencer and G. Collyer. #ifdef considered harmful or portability experience with C news. In *Proc. USENIX Conf.*, pages 185–198. USENIX Association, 1992.

[48] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 107–119. IEEE CS, 1999.

[49] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 47–60. ACM, 2011.

[50] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM, 2007.

[51] T. van der Storm. Variability and component composition. In *Proc. Int'l Conf. Software Reuse (ICSR)*, pages 157–166. Springer, 2004.

[52] R. van Ommering. Building product populations with software components. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 255–265. ACM, 2002.

[53] D. Walker. Substructural type systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1, pages 3–43. MIT Press, 2005.

A Proofs

A.1 Properties of M

Lemma 1. *Let* $\vdash$ *be a monotonic relation,* $(\Gamma_1, \Delta_1) \in \mathsf{M}$*,* $(\Gamma_2, \Delta_2) \in \mathsf{M}$*,* $(\Gamma_1, \Delta_1)\ \mathit{OK}$*,* $(\Gamma_2, \Delta_2)\ \mathit{OK}$*, and* $dom(\Delta_1) \cap dom(\Delta_2) = \emptyset$*. Then* $\Gamma_1 \cup \Gamma_2 \setminus (sig(\Delta_1) \cup sig(\Delta_2)) \vdash \Delta_1 \cup \Delta_2$*.*

Proof. $\Gamma_1 \cup \Gamma_2 \setminus (sig(\Delta_1) \cup sig(\Delta_2)) \vdash \Delta_1 \cup \Delta_2$ if and only if $\forall \mathtt{x} \in dom(\Delta_1 \cup \Delta_2).\ \Gamma_1 \cup \Gamma_2 \cup sig(\Delta_1) \cup sig(\Delta_2) \vdash \mathtt{e} : \mathtt{t}$ where $(\Delta_1 \cup \Delta_2)(\mathtt{x}) = (\mathtt{e}, \mathtt{t})$. Since $dom(\Delta_1) \cap dom(\Delta_2) = \emptyset$, assume $\mathtt{x} \in dom(\Delta_1)$ with $\Delta_1(\mathtt{x}) = (\mathtt{e}, \mathtt{t})$. Then, from $(\Gamma_1, \Delta_1)\ \mathsf{OK}$ it follows that $\Gamma_1 \cup sig(\Delta_1) \vdash \mathtt{e} : \mathtt{t}$. Thus, by monotonicity, $\Gamma_1 \cup \Gamma_2 \cup sig(\Delta_1) \cup sig(\Delta_2) \vdash \mathtt{e} : \mathtt{t}$. Analogous for $\mathtt{x} \in dom(\Delta_2)$. □

Theorem 1 (P4: Composition preserves typing in M)**.** *Given a monotonic relation* $\vdash$*, module composition of well-typed, compatible modules preserves typing, that is,* $\forall \mathtt{m}_1, \mathtt{m}_2 \in \mathsf{M}.\ \mathtt{m}_1\ \mathit{OK} \wedge \mathtt{m}_2\ \mathit{OK} \wedge \mathtt{m}_1 \div \mathtt{m}_2 \Rightarrow \mathtt{m}_1 \bullet \mathtt{m}_2\ \mathit{OK}$*.*

Proof. Let $\mathtt{m}_1 = (\Gamma_1, \Delta_1)$, $\mathtt{m}_2 = (\Gamma_2, \Delta_2)$, and $\mathtt{m}_1 \bullet \mathtt{m}_2 = (\Gamma', \Delta')$. $(\Gamma', \Delta')\ \mathsf{OK}$ if and only if $(dom(\Gamma') \cap dom(\Delta') = \emptyset) \wedge \Gamma' \vdash \Delta'$. We inline the definitions of $\Gamma' = \Gamma_1 \cup \Gamma_2 \setminus (sig(\Delta_1) \cup sig(\Delta_2))$ and $\Delta' = \Delta_1 \cup \Delta_2$. The first conjunct then follows from the equation $dom(sig(\Delta)) = dom(\Delta)$ for all Δ. The second conjunct follows by Lemma 1. □

A.2 Properties of M^ν

Lemma 2. *Let* $\mathtt{m}_1 = (\nu_1, \Gamma_1, \Delta_1) \in \mathsf{M}^\nu$ *and* $\mathtt{m}_2 = (\nu_2, \Gamma_2, \Delta_2) \in \mathsf{M}^\nu$ *with* $\mathtt{m}_1\ \mathit{OK}$*,* $\mathtt{m}_2\ \mathit{OK}$*, and* $(\nu', \Gamma', \Delta') = \mathtt{m}_1 \bullet \mathtt{m}_2$*. Then* $\nu' \subseteq dom(\Gamma')$ *and* $\nu' \subseteq dom(\Delta')$*.*

Proof. By $\mathtt{m}_1\ \mathsf{OK}$ and $\mathtt{m}_2\ \mathsf{OK}$, we deduce $\nu_1 \subseteq \Gamma_1$, $\nu_1 \subseteq \Delta_1$, $\nu_2 \subseteq \Gamma_2$, and $\nu_2 \subseteq \Delta_2$. $\nu' \subseteq dom(\Gamma')$ and $\nu' \subseteq dom(\Delta')$ then follow from the definition of Γ' and Δ'. □

Theorem 2 (P4: Composition preserves typing in M^ν)**.** *Given a monotonic relation* $\vdash$*, module composition of well-typed, compatible modules preserves typing, that is,* $\forall \mathtt{m}_1, \mathtt{m}_2 \in \mathsf{M}^\nu.\ \mathtt{m}_1\ \mathit{OK} \wedge \mathtt{m}_2\ \mathit{OK} \wedge \mathtt{m}_1 \div \mathtt{m}_2 \Rightarrow \mathtt{m}_1 \bullet \mathtt{m}_2\ \mathit{OK}$*.*

Proof. Let $\mathfrak{m}_1 = (\nu_1, \Gamma_1, \Delta_1)$, $\mathfrak{m}_2 = (\nu_2, \Gamma_2, \Delta_2)$, and $\mathfrak{m}_1 \bullet \mathfrak{m}_2 = (\nu', \Gamma', \Delta')$. (ν', Γ', Δ') OK if and only if $\nu' \neq \emptyset$, $\nu' \subseteq \mathit{dom}(\Gamma')$, $\nu' \subseteq \mathit{dom}(\Delta')$, and $\forall \mathfrak{c} \in \nu'.\ (\Gamma'(\mathfrak{c}), \Delta'(\mathfrak{c}))$ OK. The first constraint follows from $\mathfrak{m}_1 \div \mathfrak{m}_2$. The second and third constraint follow from Lemma 2. For the final constraint, note that $(\Gamma'(\mathfrak{c}), \Delta'(\mathfrak{c})) = (\Gamma_1(\mathfrak{c}) \cup \Gamma_2(\mathfrak{c}) \setminus (\Delta_1(\mathfrak{c}) \cup \Delta_2(\mathfrak{c})),\ \Delta_1(\mathfrak{c}) \cup \Delta_2(\mathfrak{c}))$, which equals $(\Gamma_1(\mathfrak{c}), \Delta_1(\mathfrak{c})) \bullet (\Gamma_2(\mathfrak{c}), \Delta_2(\mathfrak{c}))$. Thus, the final constraint follows from the type-preservation Theorem 1 of module system M. □

Lemma 3. *For the computation of conflicts the following properties hold.*

(i) *conflictpresence and conflicttype are commutative*

(ii) $\mathit{conflict}(\Gamma_1, \Delta_1, \Gamma_2, \Delta_2) = \mathit{conflict}(\Gamma_2, \Delta_2, \Gamma_1, \Delta_1)$ *for all* $\Gamma_1, \Gamma_2, \Delta_1, \Delta_2$

(iii) $\mathit{conflictpresence}(\Gamma', \Gamma) = \mathit{conflictpresence}(\Gamma_1, \Gamma) \cup \mathit{conflictpresence}(\Gamma_2, \Gamma)$, *where* $\Gamma'(\mathfrak{c}) = \Gamma_1(\mathfrak{c}) \cup \Gamma_2(\mathfrak{c})$

(iv) $\mathit{conflicttype}(\Gamma', \Gamma) = \mathit{conflicttype}(\Gamma_1, \Gamma) \cup \mathit{conflicttype}(\Gamma_2, \Gamma)$, *where* $\Gamma'(\mathfrak{c}) = \Gamma_1(\mathfrak{c}) \cup \Gamma_2(\mathfrak{c})$

(v) $\mathit{conflicttype}(\Gamma', \Gamma) = \mathit{conflicttype}(\Gamma_1, \Gamma) \setminus \mathit{conflicttype}(\Gamma_2, \Gamma)$, *where* $\Gamma'(\mathfrak{c}) = \Gamma_1(\mathfrak{c}) \setminus \Gamma_2(\mathfrak{c})$

(vi) $\mathit{conflicttype}(\Gamma_1, \Gamma_2) \subseteq \mathit{conflictpresence}(\Gamma_1, \Gamma_2)$

(vii) $\mathit{conflict}(\Gamma', \Delta', \Gamma, \Delta) = \mathit{conflict}(\Gamma_1, \Delta_1, \Gamma, \Delta) \cup \mathit{conflict}(\Gamma_2, \Delta_2, \Gamma, \Delta)$, *where* $\Gamma'(\mathfrak{c}) = \Gamma_1(\mathfrak{c}) \cup \Gamma_2(\mathfrak{c}) \setminus (\mathit{Sig}(\Delta_1(\mathfrak{c})) \cup \mathit{Sig}(\Delta_2(\mathfrak{c})))$ *and* $\Delta'(\mathfrak{c}) = \Delta_1(\mathfrak{c}) \cup \Delta_2(\mathfrak{c})$.

Proof. (i)–(vi) follow directly from the definition of *conflictpresence*, *conflicttype*, and *conflict*. (vii) follows from the definition of *conflict* and properties (i)–(vi). □

Theorem 3 (P6': Composition preserves compatibility)**.** *The partial composition of compatible modules only yields compatible modules, that is,*
$\div\{\mathfrak{m}_1, \mathfrak{m}_2, \ldots, \mathfrak{m}_n\} \Rightarrow \div\{\mathfrak{m}_1 \bullet \mathfrak{m}_2, \ldots, \mathfrak{m}_n\}$.

Proof. Let $\mathfrak{m}_i = (\nu_i, \Gamma_i, \Delta_i)$ and $\mathfrak{m}_1 \bullet \mathfrak{m}_2 = (\nu', \Gamma', \Delta')$. $\div\{\mathfrak{m}_1 \bullet \mathfrak{m}_2, \ldots, \mathfrak{m}_n\}$ if and only if $va \setminus vb \neq \emptyset$ where $va = \nu' \cap (\bigcap_{x>2} \nu_x) = \bigcap_x \nu_x \setminus \mathit{conflict}(\Gamma_1, \Delta_1, \Gamma_2, \Delta_2)$ and $vb = \big(\bigcup_{y>2} \mathit{conflict}(\Gamma', \Delta', \Gamma_y, \Delta_y) \cup \bigcup_{2<x<y} \mathit{conflict}(\Gamma_x, \Delta_x, \Gamma_y, \Delta_y)\big)$. By Lemma 3 (vii), we can simplify vb to $\big(\bigcup_{x \neq y} \mathit{conflict}(\Gamma_x, \Delta_x, \Gamma_y, \Delta_y)\big)$. Since also $\mathit{conflict}(\Gamma_1, \Delta_1, \Gamma_2, \Delta_2) \subseteq vb$, we get $va \setminus vb = \big(\bigcap_x \nu_x\big) \setminus \big(\bigcup_{x \neq y} \mathit{conflict}(\Gamma_x, \Delta_x, \Gamma_y, \Delta_y)\big)$, which is non-empty due to the assumption $\div\{\mathfrak{m}_1, \mathfrak{m}_2, \ldots, \mathfrak{m}_n\}$. □

Theorem 4 (P5: Commutativity and associativity of module composition)**.** *Module composition is commutative* $(\mathfrak{m}_1 \div \mathfrak{m}_2 \Rightarrow \mathfrak{m}_1 \bullet \mathfrak{m}_2 = \mathfrak{m}_2 \bullet \mathfrak{m}_1)$ *and associative* $(\div\{\mathfrak{m}_1, \mathfrak{m}_2, \mathfrak{m}_3\} \Rightarrow \mathfrak{m}_1 \bullet (\mathfrak{m}_2 \bullet \mathfrak{m}_3) = (\mathfrak{m}_1 \bullet \mathfrak{m}_2) \bullet \mathfrak{m}_3)$.

Proof. Commutativity is obvious from the definition of module composition. Associativity follows from Lemma 3 (ii), Lemma 3 (vii), and Theorem 3 by inlining the definition of module composition.

□

A.3 Properties of $\mathsf{M}^{\nu\mathfrak{l}}$

Lemma 4. *For varmodel and varmap, the following properties hold.*

(i) $\mathit{varmodel}(\nu_1 \cap \nu_2) \subseteq \mathit{varmodel}(\nu_1) \cup \mathit{varmodel}(\nu_2)$

(ii) $\mathit{varmodel}(\nu_1 \setminus \nu_2) \subseteq \mathit{varmodel}(\nu_1)$

(iii) $\mathit{varmap}(\nu_1 \cap \nu_2, \Delta) \subseteq \mathit{varmap}(\nu_1, \Delta) \cup \mathit{varmap}(\nu_2, \Delta)$

(iv) $\mathit{varmap}(\nu_1 \setminus \nu_2, \Delta) \subseteq \mathit{varmap}(\nu_1, \Delta)$

(v) $\mathit{varmap}(\nu, \Delta') \subseteq \mathit{varmap}(\nu, \Delta_1) \cup \mathit{varmap}(\nu, \Delta_2)$
where $\Delta'(\mathtt{c}) = \Delta_1(\mathtt{c}) \cup \Delta_2(\mathtt{c})$ *well-defined on* ν

(vi) $\mathit{varmap}(\nu, \Delta') \subseteq \mathit{varmap}(\nu, \Delta_1)$
where $\Delta'(\mathtt{c}) = \Delta_1(\mathtt{c}) \setminus \Delta_2(\mathtt{c})$ *well-defined on* ν

Proof. (i)–(iv) follow directly from the definition of *varmodel* and *varmap*. For (v), there are four cases in which $\Delta_1(\mathtt{c}\setminus\{\mathtt{f}\}) \cup \Delta_2(\mathtt{c}\setminus\{\mathtt{f}\}) \neq \Delta_1(\mathtt{c}\cup\{\mathtt{f}\}) \cup \Delta_2(\mathtt{c}\cup\{\mathtt{f}\})$: For $\mathtt{i} \in \{1, 2\}$, either $\mathit{dom}(\Delta_\mathtt{i}(\mathtt{c} \setminus \{\mathtt{f}\})) \neq \mathit{dom}(\Delta_\mathtt{i}(\mathtt{c} \cup \{\mathtt{f}\}))$ or for some variable $\mathtt{x}$ $\Delta_\mathtt{i}(\mathtt{c} \setminus \{\mathtt{f}\})(\mathtt{x}) \neq \Delta_\mathtt{i}(\mathtt{c} \cup \{\mathtt{f}\})(\mathtt{x})$. Either case is subsumed by $\mathit{varmap}(\nu, \Delta_\mathtt{i})$. Similarly for the proof of (vi). □

Theorem 5 (Composition preserves locality)**.** *Module composition of well-typed, compatible modules preserves the locality of configuration options, that is,*

$$\begin{array}{lll} \forall \mathtt{m}_1, \mathtt{m}_2 \in \mathsf{M}^{\nu\mathfrak{l}}. & & \mathtt{m}_1\ \mathit{OK} \wedge \mathtt{m}_2\ \mathit{OK} \wedge \mathtt{m}_1 \div \mathtt{m}_2 \wedge \\ & & \mathtt{m}_1 \bullet \mathtt{m}_2 = (\nu', \mathtt{i}', \mathtt{j}', \Gamma', \Delta') \\ & \Rightarrow & \mathit{varmodel}(\nu') \cup \mathit{varmap}(\nu', \Gamma') \cup \\ & & \mathit{varmap}(\nu', \Delta') \subseteq \mathtt{i}' \cup \mathtt{j}' \end{array}$$

Proof. By Lemma 4, $\mathit{varmodel}(\nu') \subseteq \mathit{varmodel}(\nu_1) \cup \mathit{varmodel}(\nu_2)$, $\mathit{varmap}(\nu', \Gamma') \subseteq \mathit{varmap}(\nu_1, \Gamma_1) \cup \mathit{varmap}(\nu_2, \Gamma_2)$, and $\mathit{varmap}(\nu', \Delta') \subseteq \mathit{varmap}(\nu_1, \Delta_1) \cup \mathit{varmap}(\nu_2, \Delta_2)$, which respectively are subsets of $\mathtt{i}' \cup \mathtt{j}' = (\mathtt{i}_1 \cup \mathtt{j}_1) \cup (\mathtt{i}_2 \cup \mathtt{j}_2)$ by $\mathtt{m}_1\ \mathsf{OK}$ and $\mathtt{m}_2\ \mathsf{OK}$. □

Theorem 6 (P4: Composition preserves typing in $\mathsf{M}^{\nu\mathfrak{l}}$)**.** *Given a monotonic relation* $\vdash$*, module composition of well-typed, compatible modules preserves typing, that is,* $\forall \mathtt{m}_1, \mathtt{m}_2 \in \mathsf{M}^{\nu\mathfrak{l}}.\ \ \mathtt{m}_1\ \mathit{OK} \wedge \mathtt{m}_2\ \mathit{OK} \wedge \mathtt{m}_1 \div \mathtt{m}_2\ \Rightarrow\ \mathtt{m}_1 \bullet \mathtt{m}_2\ \mathit{OK}$.

Proof. Follows directly from Theorem 2 and Theorem 5. □