*Universität Passau*
Fakultät für Informatik und Mathematik

# A Specification Language for Observer Automata in Feature-Oriented Verification

***Master Thesis***

Hendrik Speidel

**Advisors:**
Prof. Christian Lengauer Ph.D.
Prof. Dr. Dirk Beyer
Dr. Sven Apel

22.1.2011

# Abstract

This thesis presents SPLVERIFIER a toolchain aimed at verification of feature-oriented software product lines. A software product line (SPL) is located in a specific problem space and defines a set of variants i.e. related programs. Feature-oriented software development is a paradigm to develop software product lines aimed at the reuse of code across variants. To this end, the problem space is decomposed into features. A feature represents a design decision and presents a potential configuration option. Individual variants of the SPL can be composed automatically from the set of available features. This circumstance and the fact that the number of possible variants can increase exponentially with the number of features makes it hard to verify software product lines. Therefore, SPLVERIFIER needs to check that all possible variants of the SPL are correct. SPLVERIFIER is targeting a specific class of correctness problems that is inherent to software product lines called feature interactions: A feature interaction is a situation in which the combination of multiple features —even though each individual feature works as specified— leads to emergent and possibly critical behavior.

The central part of SPLVERIFIER is the AUTOFEATURE automata language to express specifications for the use with off-the-shelf software model-checkers. The automata language allows to express specifications alongside a feature's source code. The specifications are woven with code to form programs that can be used as input to existing software model checkers. Software model-checking is a formal technique to automatically verify that a program adheres to a given specification.

The toolchain can be applied to check feature-oriented product lines using two alternative approaches: All possible variants that can be composed from the product line's features can be composed and checked individually. The other approach introduced by this thesis is called variability-encoding. Here, an instrumented program that incorporates the behavior of all possible variants can be generated from the SPL. This makes it possible to check an entire product line without generating and checking all possible feature combinations.

# CONTENTS

CHAPTER

ONE

# INTRODUCTION

An aspect that can be observed in todays markets is that often not only one product needs to be developed but a software product line (SPL) consisting of multiple distinct but closely related software products. These products are related in that they might share parts of their code but they also differ in specific ways — be it in functionality, optimizations, or supported hardware architecture. However, they are located in the same problem space. Feature-oriented software development (FOSD) is a paradigm to develop software product lines aimed at the reuse of code across variants [AK2009]. To this end, the problem space is decomposed into features. A feature represents a design decision and presents a potential configuration option. Individual variants of the SPL can be composed automatically from the set of available features. This circumstance and the fact that the number of possible variants can increase exponentially with the number of features makes it hard to verify software product lines. In particular, feature interactions are a specific class of correctness problems inherent to software product lines: A feature interaction is a situation in which the combination of multiple features —even though each individual feature works as specified— leads to emergent and possibly critical behavior. It is a field of ongoing research to find techniques to detect and mitigate feature interactions in software product lines.

## 1.1 Problem Statement

Feature interactions present a major problem in feature-oriented software development [CKMR2003]. Software Model Checking offers the possibility to prove that a program adheres to a given specification [Clar1997] —a broader statement than compared to the result of software testing. We therefore investigate how software model checking can be integrated in the feature-oriented software development process. FEATUREHOUSE was chosen as the method to produce variants from the product line. Existing model checkers are reused for verification. The C programming language is employed for SPL development. As a starting point, a mechanism is needed to express specifications in a modular way that integrates with features. Additionally, alternative methods may be investigated that allow the detection of feature interactions by the means of software model checking.

Ideas are implemented prototypically in JAVA and Python to be able to test their applicability in feature aware verification.

## 1.2 Related Work

This thesis builds on previous research of feature-oriented software development and software model checking. Specifically, the concepts presented here are interpretations and adaptations of the following works:

- Verification of product lines has been studied previously. Post et al. study the Linux Kernel as a configurable product line [PSK2009]. Li et al. use model checking to find feature interactions in product lines where feature code is represented as state machines [LKF2002]. FEATUREALLOY can be used to express and verify feature-oriented designs [ASLK2010].
- The automata language to express the specifications that is presented here is modeled after the BLAST *Query Language* that is used by the BLAST software model checker [BCHJM2004]. The SLAM model checker also uses an automata language to express specifications [BRa2002].
- The concept of variability-encoding is inspired by the technique of *configuration lifting* [PoSi2008]. It generates a meta-program from a product line configurable via preprocessor directives that can then be used as input to software model checkers. Variability-encoding is an application of that concept to feature-oriented product lines.

## 1.3 Outline

Chapter 2 introduces Feature-Oriented Software Development and Model Checking in more detail. While these techniques can be applied to other languages also, this document focuses on their application in conjunction with the C programming language.

Next, chapter 3 builds on said chapter and introduces a first approach for integrating FOSD and software model checking that can be used to check individual variants. After an analysis of the additional requirements imposed by the feature-oriented development process the proposed solution is introduced and open problems are discussed.

Chapter 4 then shows how a modified composer can be used to enable checking a property for the whole product line instead of just being able to check the generated products individually. To that end, an instrumented program is generated from the SPL where the feature selection process is postponed to the early runtime of that program in such a way that the model checker needs to consider all possible variants of the SPL during its analysis.

CHAPTER

TWO

# BACKGROUND

This chapter introduces the underlying concepts and tools used in this thesis.

In section *2.1*, feature-oriented software development will be introduced as a paradigm for the development of software product lines. Specifically, the FEATUREHOUSE framework will be presented, which allows to automatically synthesize different products from a set of feature modules.

After that, section 2.2 software model checking is introduced as a formal method to prove, that a model adheres to a specification. Here, CPACHECKER will be described —a verification framework for programs written in the C programming language. Additionally, a short description of the *C Bounded Model Checker* CBMC is given.

The examples presented here are centered around an e-mail messaging product line that represents a simplified implementation of a subset of the e-mail system described by Hall that captures AT&T's domain knowledge on e-mail systems [Hall2005]. It has been chosen for its real world applicability and because feature interactions in that system have already been documented. The system will be used for a case study later on.

In the following chapters, the concepts and tools introduced here are applied in combination with the goal of detecting feature interactions using software model checking.

## 2.1 Feature-Oriented Software Development

A prominent example for a software product line is the Linux Kernel. Its users can choose from a variety of configuration options before building a specific kernel. The mechanism to allow this configuration was implemented specifically for the kernel. The implementation is based on the *C Preprocessor*. This means that all the variability of the kernel is encoded in clauses for conditional compilation that are scattered throughout the source code. It creates difficulties in maintaining the codebase, tracing a specific feature‘s code, and also makes it hard to test or check the consistency of the product line considering the possibly huge number of different feature combinations that a specific product can incorporate.

One solution to these problems is the use of *Feature-Oriented Software Development*(FOSD) [AK2009]. Here, the source code of a software system is decomposed into feature modules. A feature module represents a configurable unit of functionality of an application domain. It implements and encapsulates a design decision. Features serve as the basic building blocks for different products in the same problem space. The set of products generatable from the set of features constitutes the product line.

Decomposition of the system is achieved by either virtual or physical separation of features. Virtual separation is employed in the *Colored Integrated Feature Development Environment (CIDE)* [Kaes2007]. It maps features on top of an already composed system. With FEATUREHOUSE [AKL2009], the features are physically separated from one another in different directories; specific products are composed from features when needed. The *AHEAD tool suite* [Bato2006] which is aimed primarily at feature-oriented programming with JAVA also physically separates features.

### 2.1.1 Goals

To be able to better understand the goals of a feature-oriented approach to software development it is necessary to analyze the shortcomings of current approaches to software product lines: We will therefore revisit the example of the Linux kernel. As mentioned before the configuration mechanism of the Linux kernel is based on the C preprocessor. Different configurations are managed in the source code using `#ifdef` directives. Often, multiple directives are needed for the same concern in different places throughout the code thereby *scattering* the concern’s code. In turn, this may lead to similar code being distributed across different modules. A change to the implementation may require finding and editing all affected code. Another problem is *code tangling*. As the different configurations are managed with `#ifdef` directives, different concerns are implemented within the same source file. This makes the code more difficult to understand. Also the problem gets worse if the level of configurability is increased or the directives are nested. Code tangling and scattering can lead to code that is difficult to read and therefore hard to maintain and extend. For example, it may be problematic to find the source location corresponding to a feature or all code a feature consists of, e.g. in case of an error. This is also known as the *feature traceability problem*. Feature-Orientation directly addresses this problem to gain cohesion within features.

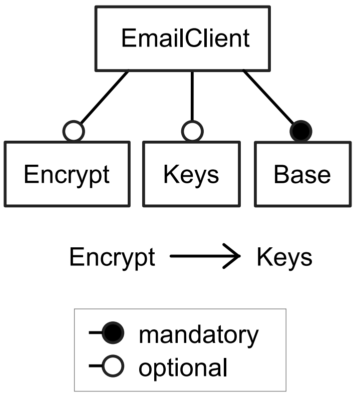
FOSD proposes the use of the feature concept throughout all phases of the software development process as it provides structure through feature separation, variation through product configurability, and the possibility to reuse features across different products. However, feature orientation also has limitations: a program cannot always be modularized across all possible concerns. So, while it offers traceability the concerns that should be modularized as features need to be selected carefully: it might become impossible to implement a specific concern as feature at a later time because the concern crosscuts existing features. This problem is also known as the *tyranny of the dominant decomposition* [TOHS1999].

### 2.1.2 Feature-Model

As stated before, the careful choice of features that make up the product line is an important aspect of FOSD. It must be possible to compose all necessary products from these features. Also, there might be restrictions on what constitutes a valid product: in what combinations and in which order can the features be combined? For example, all variants of the e-mail system product-line require the `Base` feature; the `Encrypt` feature depends on the `Keys` feature, which provides management of encryption keys. Other product lines might have more complex restrictions and dependencies.

In FOSD, the list of features, their order of composition, and their interdependencies constitute the *feature-model*. The restrictions are commonly represented by using diagrams or as a satisfiability problem.

The GUIDSL program and its model language is one implementation, that can be used to describe a feature-model. It can also be used to check feature compatibility, aid in product selection, and to check the product line for overall consistency. It is part of the AHEAD tool suite [Bato2006].



```
EmailClient : [Encrypt] [Keys] Base :: P1
              ;
%%
Encrypt implies Keys;
```

Figure 2.1: Feature-model diagram and corresponding GUIDSL model definition.

A GUIDSL model file consists of a grammar and a number of additional propositional formulas that define additional constraints. These two parts are separated by `%%`. Each grammar token stands for a feature. The order of composition is defined by the order the tokens appear in the grammar. Features are composed from right to left (analogous to function application). If the grammar is not sufficient, complex dependencies can be specified additionally using the propositional constraints. Figure 2.1 shows the aforementioned feature-model. On the left hand the feature-model is shown as a diagram, on the right it is expressed as a GUIDSL model. `P1` is the name of the grammar rule. All grammar rules are required to have a name.

The feature-model unifies all knowledge of the variability of the product line in one place. Thereby, FOSD also provides traceability for the concern of product line configurability.

### 2.1.3 Feature Interactions

Feature Interactions are undesired phenomenons, that can be observed in feature-oriented software systems [CKMR2003]. Specifically, a feature interaction occurs if a certain set of features produces unwanted behavior when selected in combination while the unwanted behavior is absent in products containing only a subset of these features. To give an example, one feature of the e-mail system mentioned before is the `Forward` feature which as the name suggests forwards incoming mail to other addresses. To that end, it rewrites the sender field of forwarded messages. In isolation this feature is working fine. However, if used in conjunction with the `Verify` feature the rewriting of the sender might interfere with the key selection process for signature verification. The `Forward` and the `Verify` feature are interacting and as a result variants that contain both features exhibit undesired behavior.

The concept of feature interactions originated in the telecommunications industry. In *Fundamental Nonmodularity in Electronic Mail*, Hall studies feature interactions on an e-mail product line [Hall2005]. In this system consisting of 10 features, he was able to identify 27 feature interactions. As indicated before, a rudimentary implementation of this system is used for a case study later on.

In summary, feature interactions are a major problem in FOSD as they are often not obvious [CKMR2003]. They are also difficult to detect because they only occur if a certain set of features is selected, i.e. they may be absent in most variants of the software product line.

### 2.1.4 FEATUREHOUSE

FEATUREHOUSE is a tool chain and framework for the automatic composition of software artifacts written in different languages [AKL2009]. As a general abstraction, FEATUREHOUSE uses the language-independent model of *Feature Structure Trees* (FSTs). A FST represents the hierarchical structure inherent in a software artifact and is used as an intermediate data structure that can be manipulated and especially can be merged with other FSTs.

The composition tool that is part of FEATUREHOUSE is called FSTCOMPOSER. It acts as a source-to-source translator and generates a product from a given list of features. Their corresponding source artifacts are located in *containment hierarchies*. These are directories each containing the individual files which make up a feature. FSTCOMPOSER is split up in three units: A parser, the actual composer, and a pretty printer. The parser and the pretty printer are language specific and are used to translate between the source language and FSTs. The composer itself is language agnostic because it operates only on the FSTs generated by the parser. FSTs are merged by the composer; the resulting product is then transformed back to the source language using the pretty printer. Support for a new language can be added to FEATUREHOUSE by giving an abstract description of its structure and composition rules in the form of an annotated grammar. A parser and a pretty printer can be generated from that grammar. The C programming language is already supported.

In the following sections FSTs and the composition algorithm used by FEATUREHOUSE will be described in more detail. The examples are oriented around the e-mail system. Because we only use FEATUREHOUSE to compose C artifacts, the following description focuses on the aspects relevant for it.

### 2.1.5 Feature Structure Trees

The primary abstraction to support language independent feature composition are *Feature Structure Trees* (FSTs). Each FST represents the hierarchical structure of a feature module‘s contents. The trees consist of feature structure nodes. All nodes have an associated type and a name. FST nodes can be either *Nonterminal Nodes* or *Terminal Nodes*. Terminal nodes are always leaf nodes, nonterminal nodes may have child nodes. All FSTs have a nonterminal root node. All introductions made in the feature are added below that node. The parser constructs the FST and assigns each node a name and a type. Terminal nodes also have a `content` attribute that is used to capture the corresponding value from the input. For example, the `content` of an include node would contain the include directive as it occurs in the source file.

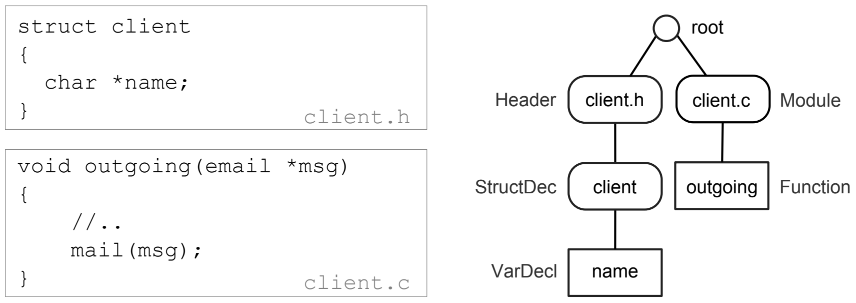
FSTs are constructed from feature modules using a language specific parser that is part of the composer. The following table lists some of the C language elements and the corresponding FST nodes the parser translates to:

| **Language Element** | **Node Type** | |
|---|---|---|
| Source file | Module | Nonterminal |
| Header file | Header | Nonterminal |
| Struct declaration | StructDec | Nonterminal |
| Include Directive | Include | Terminal |
| Variable/Field declaration | VarDecl | Terminal |
| Function declaration | Function | Terminal |

In contrary to an abstract syntax tree generated by a typical C Parser, a C function is a leaf node in the FST. The function body is available via the node‘s `content` attribute. This level of detail might not be enough for a compiler, but it is sufficient for the composer. As an example, Figure 2.2 shows part of the source code and the corresponding FST of the e-mail system’s `Base` feature. Nonterminal nodes are depicted as ovals; terminals are depicted as rectangles.

### 2.1.6 Feature Composition

To compose a product from a selection of feature modules, these have to be translated to FSTs first as described in the previous section. After that, these FSTs are



```
struct client
{
  char *name;
}                                client.h
```

```
void outgoing(email *msg)
{
    //..
    mail(msg);
}                                client.c
```

Figure 2.2: Simplified FST of the e-mail system‘s `Base` feature.

combined into one single resulting FST, which is then written out using the pretty printer.

The composition method used by FEATUREHOUSE is called superimposition and is described by Apel et al. [AL2008]. In the following, it will be denoted by •; the FST of the left operand is superimposed on the right e.g. `Encrypt` • `Base` superimposes the `Encrypt` FST onto the `Base` FST. In FEATUREHOUSE, the process of feature composition works as follows: The composer receives a compositional formula as input that describes the selected features and the order of composition. The composition order is relevant and a total ordering of all features is normally used. FSTs are composed two at a time and are processed from right to left. So, to generate the product described by `Encrypt` • `Keys` • `Base`, first `Keys` • `Base` is produced and later `Encrypt` is superimposed on the result.

The superimposition of two FSTs is defined as the composition of their root nodes. Beginning there, child nodes are composed recursively. Two nodes can only be composed, if they have identical names, identical types, and a identical relative location in their tree. We will refer to such nodes as corresponding nodes. How two corresponding nodes are composed, depends on the nodes being nonterminal nodes or terminal nodes. The different strategies will be described in the next sections.

### Composition Of Nonterminal Nodes

Two Nonterminals —a left and a right one— are composed the following way:

- Create a nonterminal node `result` with name and type of the input nodes.
- For all children of the right node, search for a corresponding child of the left node with the same name and type.

  - If such a corresponding node exists compose the right child with its corresponding node and add the resulting nodes to `result` as children.
  - If no corresponding node exists, add the right child as a child to `result`.

- Add all childs of the left node that have no corresponding node on the right to `result` as children.

Essentially, this places all children of both the left and the right nodes with no corresponding node below `result` in the generated tree. For corresponding nodes the result of their composition is added. Also, the children of `result` retain their order.

### Composition Of Terminal Nodes

Terminal nodes are composed by applying a *Composition Rule*. A composition rule takes the left and the right terminal node as input and returns one or more nodes as output. These are added to the resulting FST in place of the input nodes. FEATUREHOUSE already offers a variety of different composition rules. For new languages, composition rules can be added easily.

The composition rule that is applied depends on the type of the terminal nodes that are to be composed. The C language support of FEATUREHOUSE uses the `Replacement` composition rule on all terminal node types except on function nodes. Function nodes are composed using the `FunctionRefinement` composition rule. Both rules will be introduced now.

### The `Replacement` Composition Rule

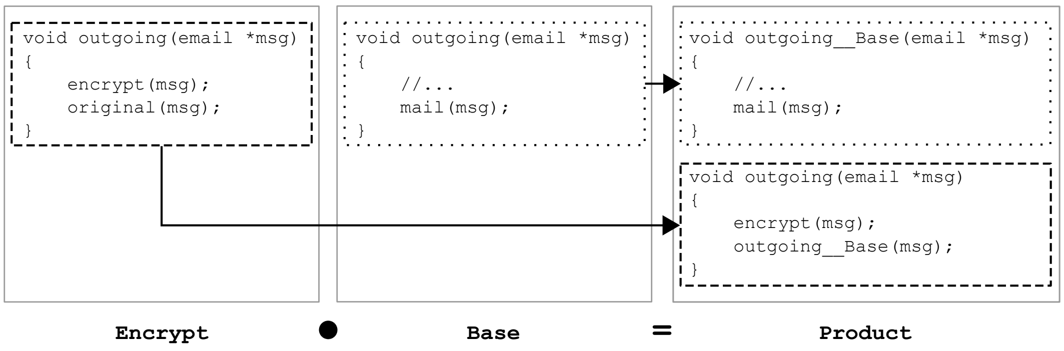
If this composition rule is applied the left node is added to the resulting FST. The node on the right is completely ignored: it is replaced by a revised version.

The replacement rule serves multiple purposes: e.g. when applied to include nodes, the replacement rule substitutes two identical includes by one; for a global variable, it allows to assign a refined value.

### The `FunctionRefinement` Composition Rule

This composition rule is used for function declarations. It was specifically added for the C language. Other composition rules like the replacement rule are reusable over different languages. The rule allows stepwise function refinement with access to the original implementation —a concept also found in object orientation inheritance.

The rule works as follows: If the contents of the left function node does not contain a call to `original`, this composition rule behaves like `Replacement`. The left function node replaces the function node on the right thereby fully overriding the former implementation. If a call to `original` is made, a new name for the right function node is generated and assigned. The call to `original` in the left node‘s contents is rewritten to the new name. Then both the left and the right node form the result and are appended to the resulting FST. Figure 2.3 shows an example of how this rule affects the source code of a function in the product. The `Base` feature’s `outgoing` function is refined by the `Encrypt` feature. The resulting product contains both implementations and the call to `original` has been rewritten.



```
void outgoing(email *msg)
{
    encrypt(msg);
    original(msg);
}
```

```
void outgoing(email *msg)
{
    //...
    mail(msg);
}
```

```
void outgoing__Base(email *msg)
{
    //...
    mail(msg);
}
```

```
void outgoing(email *msg)
{
    encrypt(msg);
    outgoing__Base(msg);
}
```

Figure 2.3: Refinement of the `outgoing` function of the feature `Base` by `Encrypt`(simplified).

As a final example for FST superimposition, Figure 2.4 shows the composition of `Encrypt` • `Base` on FSTs. As before, the FSTs have been simplified for demonstration purposes. The composition starts with the root nodes. This results in the composition of the two corresponding `Module` nodes. Being nonterminal nodes, the children need to be considered. For all children on the left, the corresponding nodes are looked up on the right. The `outgoing` nodes are composed using the function refinement rule thereby producing two distinct function nodes in the resulting tree. After that, the `encrypt` function node is added to the result.

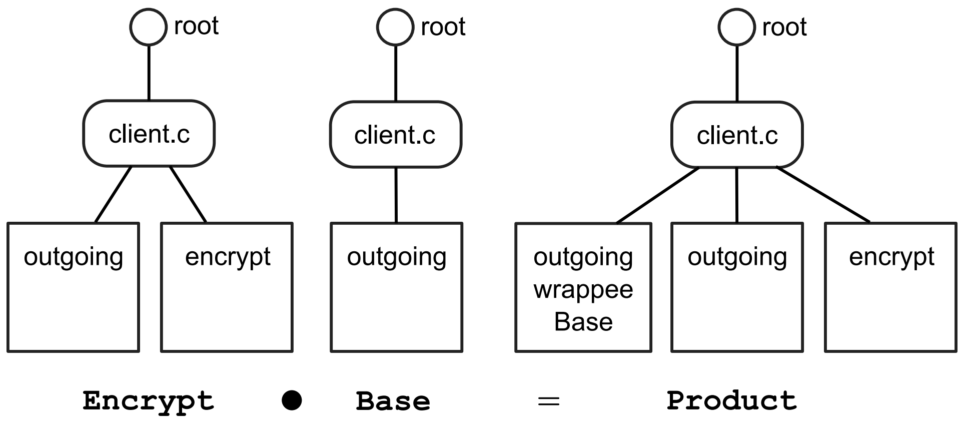


Figure 2.4: Simplified FSTs of the `Encrypt` feature and the `Base` feature are superimposed.

## 2.2 Model Checking

Given the inherent complexity of large software systems, quality assurance and verification is necessary. This is especially true for safety-critical systems. Currently, this is mostly done by peer-review and testing the software. However, testing leaves a lot to be desired: It only verifies the correctness of the software for a particular case.

Model checking is an automated technique for verifying finite-state reactive systems [Clar1997][BK2008]. The principle is currently applied in both the verification of hardware and software. In contrast to testing, model checking is a formal method and examines all possible states of a system. It considers systems that have finite state or may be reduced to finite state by means of abstraction.

The goal of model checking is to determine whether a system satisfies a certain safety property. To that end, the model checker receives a system description and a safety property as input. After that, most model checkers work automatically and need no expert knowledge to operate. The model checker then either confirms that the property is satisfied or reports that the property has been violated. In case of a violation, the model checker provides a counterexample in the form of an execution path that violates the property. It can be used to analyze the failure an can help reveal design errors.

The system descriptions accepted by model checkers vary in format: Some use special system modeling languages e.g. SPIN [Hol1997] a model checker for concurrent systems uses Promela —a language specifically designed for it. Other model checkers directly accept source code of general purpose programming languages. For example, BLAST [BHJM2007] and CPACHECKER [BK2009] both operate on C source code.

Depending on the model checker used, safety properties are specified by annotating source code, using a specialized specification language, or are expressed as a formula in temporal logic. Safety properties are requirements such as the absence of deadlocks, uninitialized variables, memory management errors, or other inconsistent states that might cause a program to crash or behave incorrectly.

To perform the analysis, the model checker uses a state graph of a given program. States represent states of the program's execution. A transition represents an execution step of the program. A state contains its variable values and location counters and can have predicates associated with it. The model checker tries to reach marked error states in the state graph by performing a reachability analysis that takes the predicates into account. If an error state is reached a bug is reported. If no error state is ever reachable, the verification is successful. Generally, explicit and symbolic model checking can be distinguished: In *explicit model checking*, all states are enumerated on the fly and are processed one at a time. Explicit model checking is very resource intensive as states may need to be stored for later use.

*Symbolic model checking* operates on sets of states at a time. Also, the transition relation of the state graph is not explicitly represented. Instead, the state space is constructed using boolean functions mostly in the form of *binary decision diagrams* (BDDs) —a very efficient way to represent and manipulate boolean functions. This way, symbolic model checking can handle even very large state spaces.

Model checking suffers from the *state-space explosion problem*. The state space of a realistic system can be very large, possibly infinite and may therefore easily exceed the amount of available memory. Hence, it is often infeasible to construct and explore the entire state graph. It is a field of ongoing research to find and improve techniques to reduce, abstract, and efficiently search the state space. Special care needs to be taken in the development of model checkers that the implementation remains *sound*: a model checker must not generate false negatives, i.e. the model checker must not report the absence of bugs in a program that actually violates a specification. Also, false positives should be avoided but can be tolerated to some extend.

FOSD adds additional complexity to model checking and verification in general, because of the added variability introduced through feature configurability. Instead of one program that needs to be verified the developer is confronted with a possibly huge number of variants.

### 2.2.1 CPAchecker

As mentioned before, one model checker used in this thesis is CPAchecker [BK2009]. It can be used to verify single threaded programs written in the C programming language. CPAchecker is written in JAVA and is built around the concept of *Configurable Software Verification*.

Instead of implementing a specific model checking algorithm it provides an extendable framework and infrastructure to integrate verification components easily. Components are called *configurable program analysis* (CPA) and each consist of an abstract domain and a set of operations that must conform to the CPA formalism [BHT2007]. CPAchecker can be used to perform a reachability analysis using a CPA. Also, multiple CPAs can be used in combination. Therefore, the individual CPAs are placed inside a CompositeCPA. Its operations can be automatically constructed from the individual CPAs' operations.

CPAchecker already provides a number of CPAs: Inspired by BLAST [BHJM2007] it implements symbolic model checking with predicate abstraction and explicit-value analysis. It also includes a CPA for pointer analysis.

Safety properties can be directly encoded in a CPA. Additionally, CPAchecker includes a CPA that allows safety properties to be specified as observer automata. These automata can be specified in configurable files and are loaded by the CPA on initialization. An error location automaton is already available. When CPAchecker is configured to use the observer automaton CPA with this configuration, all labels named `error` that occur in the source code are considered inconsistent states and CPAchecker reports a bug as soon as it reaches such a label. Observer automata are discussed in more detail in the next section.

Before CPAchecker can perform an analysis, the source code must be preprocessed and merged into the *C intermediate language* (CIL) [NPRW2002]. CIL applies certain transformations to simplify the source code. For example, it eliminates `for` loops by replacing them with `while` loops. By using this approach, CPAchecker's parser does not need to take all edge cases of the C language into account and can remain less complex.

For analysis, CPAchecker first constructs a syntax tree of the program in question. This tree is then transformed into a set of *Control Flow Automata (CFA)* —the main data structure the configurable program analysis works upon and constructs the state space from. A CFA node represents a control-flow location, a CFA edge represents an execution step. Due to the reduction to CIL only a small number of different edge types need to be considered: An assume operation, an assignment block, a function call and a function return.

After the set of CFAs has been constructed, CPAchecker‘s main algorithm performs the reachability analysis. If an error location is reached, the counter example is logged in the form of the error path and the assignments made along it. The counterexample can the be used to retrace the error and in turn can offer insight on how to correct the error.

### 2.2.2 CBMC

CBMC is a bounded model checker for C programs [CKL2004]. With bounded model checking the state graph is unwound up to a given bound to eliminate cycles. If no error state is reachable within that bound, the verification is successful. In practice, this means that for a successfully verified program bugs still can occur if loops or recursion are executed more often than the specified bound. The program is only guaranteed to be safe up to that bound. Using the bound reduces the number of states the model checker needs to analyze and therefore may result in a much faster analysis. CBMC can also verify programs that use heap allocations as it precisely reproduces the semantics of the C programming language for analysis. In contrast, CPAchecker does not currently support all constructs of the C language.

For verification CBMC builds the unwound state graph of the analyzed program and transforms it into a single equation. This equation is then passed to a decision procedure to check if the equation is satisfiable. If so, the verification has failed and the counter example is generated from the satisfying configuration.

### 2.2.3 Observer Automata

As stated before, safety properties can be expressed as formulas of temporal logic. However, automata are sometimes used instead because they are often easier to understand and can also be more tightly integrated with the target language e.g. may use functions that are present in the analyzed program.

Observer automata monitor the state of a system. They remain passive, i.e. they must not alter the state of the observed system. However, they can adapt their own state as a result of changes observed in the state of the observed system. Observer automata are not required to have a finite number of states. Typically, infinite states are achieved by adding memory access to the automata language. To express a safety property as an automaton, some of the observer's states are labeled as error states. If such a state is reached during analysis, the safety property was violated. Typically, the specific automata language provides a keyword to label error states as such.

For model checking, there are currently two approaches how observer automata are implemented: It is possible to implement the concept within the model checker or as a preprocessor that translates and embeds the safety properties in the source code. This is done by transforming the automaton's states and transitions into source code of the target language and embedding this code within the program's original source code. The resulting instrumented program then contains the safety property, e.g. in the form of added control flow around specific functions and labels that indicate inconsistent states. This approach is employed by SLIC [BRa2002] which is a specification language for the SLAM model checker that is used to verify device drivers [BRb2002]. On the other hand, the observer automaton CPA of CPACHECKER allows the execution of observer automata inside the model checker. This is realized by embedding an automaton interpreter into the model checker. During analysis the automaton CPA's transfer relation is used to update the internal states of the observer automaton.

CHAPTER

THREE

# MODEL CHECKING OF VARIANTS

This chapter describes one possible way to integrate model checking into the FOSD process. Here, model checking is performed on all variants of the product line in sequence. Essentially, the traditional process of FOSD is reused. After a specific variant has been composed it is verified using the model checker. This way, errors can be detected in relevant variants of the SPL. These can then be corrected by modifying one or more features of the SPL.

This chapter is structured the following way: First, Section 3.1 discusses the requirements and problems that arise when combining FOSD with model checking and discusses possible solutions. Section 3.2 introduces the proposed observer automata language we use to describe the specifications needed for the model checker. After that, section 3.3 goes into detail on how the different test scenarios are setup. Then, section 3.4 describes the tool that was additionally developed to aid in interpretation of the counterexample given by the model checker in case of a bug.

## 3.1 Requirements

To make use of a model checker, a way is needed to express the specifications. As already described in the background chapter model checkers use differing methods here. Often, the specification is directly given within the source code, e.g. in the form of error labels or assertions. This makes it hard to check against more than one specification as the code has to be prepared manually beforehand. In FOSD, this would mean an even bigger effort as this would need to be done for each variant generated from an SPL. Therefore, we externalize the specifications each to its own file by reusing the aforementioned concept of observer automata.

To be able to express the specifications to detect the feature interactions in the e-mail system we need a way to add error locations depending on the internal state of the system. With an observer automata language that supports only reading variables we would be forced to reimplement much of the functionality within the specification that is already exposed as a function in the e-mail system. Having to reproduce possibly large amounts of code can lead to errors in the specification. Also, over time the implementations given by a specific variant and a specification could drift apart. Therefore, we choose to support direct calls to functions offered by the analyzed program from within the specification. This is however not without its drawbacks: The parts of the application that are reused in the specification are implicitly trusted to work correctly. Also, special care needs to be taken that specifications remain side-effect free as violating this property would invalidate the analysis as side-effects could possibly affect the internal state of the analyzed program thereby altering its behavior.

Another problem is that specifications need to refer directly to code, e.g. variables or functions. In the context of FOSD variables and functions may not always be present in all variants. Therefore, only a subset of all specifications may be applicable to a particular variant. As a consequence specifications are placed within a particular feature that contains code they may access. Figure 3.1 shows an overview of the composition process. Feature code is composed as described in section *2.1*. Specifications located in features that are not selected are filtered out.

The next problem is more specific to the e-mail-system: It can be considered a feature-oriented library for e-mail related applications. It provides functionality to send and receive messages but there is no actual application in place to use that functionality —there is no main function. Therefore, to apply model checking, we needed a program that makes use of the exposed functionality. To be able to check the system in multiple configurations —referred to as *scenarios* in the following— a small domain specific language to describe these is provided. This reduces the amount of code that needs to be written thereby making the process less error prone as scenarios can be described more concise. Also, it allows to generate code that can be checked more efficiently by the model checker than a scenario modeled using the standard feature-oriented approach. Obviously, this scenario language needs to support FOSD to be applicable in this context.

With the scenario language and the observer automata it is then possible to generate a program that incorporates the error locations of the specification and that

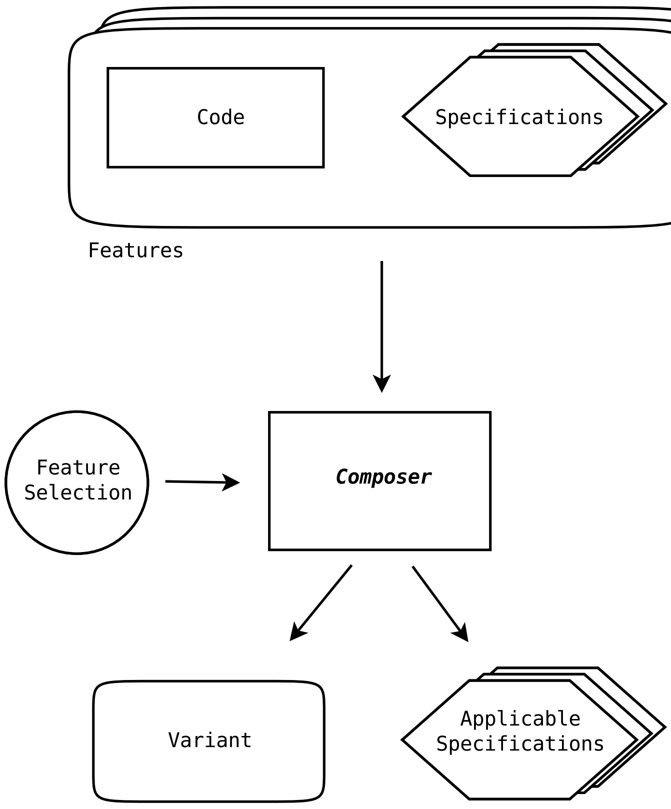


Figure 3.1: Composition of a variant.

operates within the parameters of the given scenario. This program —referred to as *checkable program* from now on— can then be given to the model checker for analysis. In case of safety, the variant is guaranteed to adhere to the specification. In case of a detected bug however, the counterexample of the model checker needs to be analyzed to deduct the features the problem is related to. Unfortunately, the variant has already been composed and various transformations applied to it at this point. Therefore, a tool is needed to assist the programmer in association of code to features. Here, we will make use of a tool that allows to inspect the error-path and visually decomposes it into features.

## 3.2 The Observer Automata Language

This section introduces the proposed specification language. The implementation of the translator is called AutoFeature. We will now look into how specifications are woven onto composed variants followed by a description of the automata language. This specification language is an adaption of the automata language used by the BLAST model checker [BCHJM2004]. Automata reside in separate files along with the code in the containment hierarchy.

### 3.2.1 Weaving Process

The automata language can be seen as a limited aspect-oriented language. The concepts are so similar that the *Aspect-C-Compiler* (ACC) —a conventional aspect-oriented source-to-source translator— is used for the actual weaving process [GJ2008]. ACC is an adaptation of AspectJ [KHHKPG01] that targets the C programming language. In a preceding step, the specification is translated into an aspect usable by ACC. This translator has been implemented using the ANTLR-Parser-Generator [P2007]. The C parts of the grammar are based on an existing C grammar for ANTLR, that was adapted to incorporate the functionality of the automata language.

As outlined before, the automata language is implemented as a preprocessor that translates the specification and weaves it onto the source code. Figure 3.2 shows an overview of the weaving process. To receive a program that we can supply to the model checker we translate the specification into an aspect. Then, we use ACC to weave the composed variant and the aspect and can then use the model checker to verify the resulting checkable program.

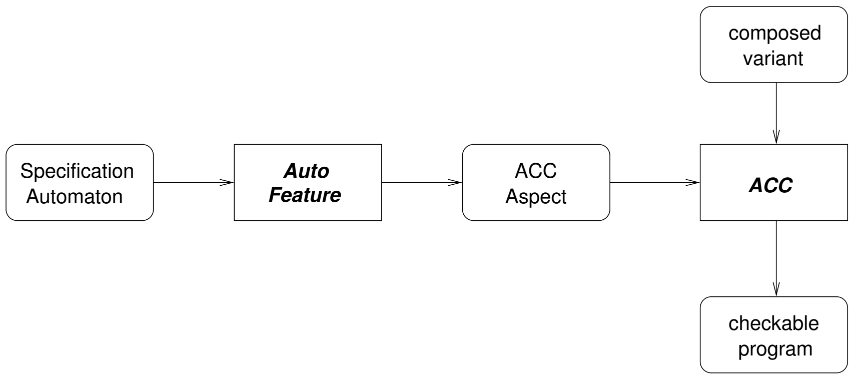


Figure 3.2: Process of weaving a specification and a variant.

### 3.2.2 Safety Automata

Automata can provide introductions to the code of the variant. By adding global variables and adding fields to structures automata can have states. Automata react to changes in the observed program by specifying *Event-Condition-Action* (ECA) rules. Events occur at the beginning and end of the execution of a function. If an automaton specifies an ECA rule for an occurring event and the condition holds, the action block is executed. Actions can contain C statements and may signal that an inconsistent state is reached by means of the `fail` statement. An automaton is translated into an aspect, where every automaton event is realized as a before-call or after-call pointcut respectively.

```
dummy {
    <typedefs>
}

<includes>

automaton <name> {
    introduction {
        <introductions>
    }
    <eca_rules>

}
```

Figure 3.3: Structure of a specification file

Figure 3.3 shows the basic structure of a specification file. If types of the observed programs are referenced, the appropriate header files that contain the type declarations or function prototypes need to be specified. The syntax mimics the syntax of the include directive of the C preprocessor. However, the translator does not parse included files. As a result, this means that if type names are referenced e.g. in action blocks these need to be declared. This can be done in a in a `dummy` block before the includes. This restriction is imposed upon the automata language by the C programming language. Because of its complex grammar C-parsers need access to type names to be able to parse the code. The `dummy` block can contain `typedef` declarations in the format used in C. If no type declarations are needed the whole `dummy` block can be omitted. The contents of the `dummy` block is not written to the generated aspect.

The body of an automaton may begin with an `introduction` block and may be followed by ECA rules. These constructs are described in the following sections.

### 3.2.3 The Introduction Block

Introductions necessary to express the temporal safety property can be added inside the `introduction` block of an automaton.

Valid introductions are:

- declarations (including function declarations) in C syntax.
- shadow declarations.

Shadows can be used to add fields to existing struct or union types.

The syntax is as follows:

```
shadow <identifier> {
    <declarations>
};
```

`identifier` must be a structure or union type name e.g. `struct email`, or a name assigned to a struct or union type using a `typedef` declaration.

`declarations` must be valid struct or union field declarations. These will be added to the type referenced by `identifier`.

All types referenced either as `identifier` or within `declarations` need to be visible to the automaton. This means that the appropriate header files have to be included at the top of the file using the `#include` directive as known from C.

### 3.2.4 ECA Rules

With ECA rules, an automaton can react to events within the observed application. Especially, if a specific event within the execution context of the application results in an inconsistent state an ECA rule‘s action block can be used to signal the failure to the developer.

An ECA rule for an automaton can be defined using the following syntax:

```
<event_pattern> [if(<expression>)] {
    <action_statements>
}
```

`event_pattern` specifies the event that the rule is bound to. This can match the beginning or the end of the execution of a function call. Specifying a condition is optional. `action_statements` are only executed if `expression` evaluates to true.

`action_statements` must be valid C statements. Additionally `fail` is available to signal an inconsistent state.

#### Event Patterns

AUTOFEATURE uses only a small subset of ACC‘s functionality. This was sufficient for the case study of the e-mail system. It offers the following two event patterns:

- the `before` pattern matches at the beginning of the execution of a function.
- the `after` pattern matches before the end of the execution of a function.

As with shadows, referenced functions and types need to be visible to the automaton.

#### `before` Patterns

`before` patterns can be specified as follows:

```
before <function> (<parameter_binding>)
```

`function` consists of the return type and the name of the function that the action should be attached to, e.g. `void mail`. The `parameter_binding` provides the means to access arguments passed to the function. For each parameter of the function prototype this consists of a binding name or _ followed by a colon and the parameter type separated by commas. All parameters have to be included. As an example, for the given function prototype `int add(int, int)` a possible `before` pattern would be:

```
before int add(_:int,a:int)
```

When function `add` is called during the execution of the program the value of the second argument to `add` will be available as `a` within the condition and the action block. Because the binding of the first parameter is set to _, the first argument is not made available within the action. With `before` patterns, the action is executed at the very beginning of the execution of the function.

#### `after` Patterns

`after` patterns can be specified exactly the same as `before` patterns. However, the action is executed right after the execution of the original function. This way, the return value of the function can also be made accessible within an action block.

The return value is made available using the following syntax:

```
after <result_binding> = <function> (<parameter_binding>)
```

`result_binding` may be any valid C identifier. The result will be available by that name within the condition and the action block. The type of the result is the return type of the function. `function` and `parameter_binding` are defined the same way they are specified when using a `before` pattern.

### 3.2.5 Action Blocks

Action blocks can contain valid C declarations and statements. Additionally, the `fail` statement is available inside action blocks. It is used to signal that the temporal safety condition has been violated. `fail` statements within actions are transformed into calls to `__automaton_fail`. This way the automata language is decoupled from the specific mechanism of signaling a failure. Most model checkers support labels to mark error states and also support checking assertions. The prototype of `__automaton_fail` is provided in `wsllib.h`. Two implementations are available in these files:

- wsllib.c — This implementation can be used to compile and run the variant. It simply prints an error message.
- wsllib_check.c — This implementation is used for analysis of the case study. It generates a label called error and jumps to it.

These files are automatically added during the weaving process. Actions must not have side effects on the internal state of the variant. Only variables and shadow fields introduced in the introduction block should be written to. As mentioned before, if functions of the analyzed program are called as part of the specification, these functions are implicitly assumed to function correctly.

### 3.2.6 Example

The following example shows a safety automaton that can be used to detect inconsistent encryption of messages. If a message is sent in encrypted form once, the message is assumed to be private. If this message is sent in unencrypted form after that, the safety property is violated. The specification implements this by introducing a global variable that tracks if the message was sent encrypted once. Later calls to mail then check the variable and a violation of the specification is signaled by means of the fail statement if it is sent in unencrypted form.

```
#include "Email.h"
#include "Client.h"
automaton EncryptConsistently {
    introduction {
        // mail_is_sensitive values:
        // -1 - uninitialized
        //  0 - no
        //  1 - yes
        int mail_is_sensitive = -1;
    }
    before void mail (client:int, msg:int) {
        if (mail_is_sensitive == -1) {
            mail_is_sensitive = isEncrypted(msg);
        } else if (mail_is_sensitive != isEncrypted(msg)) {
            fail;
        }
    }
}
```

### 3.2.7 Explicit States

Explicit states are an extension of the automata language. If explicit states are used, an automaton additionally has a current state. Initially, this is the first state defined in the automaton. In actions the state can be changed using the state change operator `-->` followed by a state defined in the automaton. States are defined like labels in C. All ECA rules that follow a state definition are only executed if the automaton is in that particular state. If explicit states are used all events must be inside a state. By default, the automaton starts in the first defined state. Explicit states are represented as a global variable in the resulting aspect.

Here, we see a safety specification that is equivalent to the specification given previously but that uses explicit states:

```
#include "Email.h"
#include "Client.h"
automaton EncryptConsistently2 {
    INIT:
        before void mail (client:int, msg:int) {
            if (isEncrypted(msg)) {
                --> CONFIDENTIAL;
            } else {
                --> PUBLIC;
            }
        }
    PUBLIC:
        before void mail (client:int, msg:int)
        if (isEncrypted(msg)) {
            fail;
        }
    CONFIDENTIAL:
        before void mail (client:int, msg:int)
        if (!isEncrypted(msg)) {
            fail;
        }
}
```

Initially, this automaton is in the `INIT` state because this state is declared first. If `mail` is called, the automaton transitions to `CONFIDENTIAL` or `PUBLIC` depending on encryption used or not. If `mail` is called again and the email does not use encryption consistently, the safety property is violated.

## 3.3 The Scenario Modeling Language

As mentioned before, the e-mail system used in the case study is a library. Therefore, to model different setups a way is needed to effectively express these scenarios using varying actions and different numbers of users. We also need to control the order in which actions can be executed in a specific scenario. Certainly, it is possible to express all this programmatically in C. Different scenarios could also be modeled as features. However, the process of specifying a possibly great number of scenarios makes this cumbersome. The use of a domain specific language has the following benefits: Scenarios can be expressed in a more condensed way that aids readability. Also, this layer of indirection allows to translate the scenario to code that is specifically optimized for the model checker. For example, the current implementation can in some cases reduce the number of functions that would otherwise be generated by the composer. Also, some model checkers like CBMC can better cope with `for` loops than `while` loops. For CPACHECKER this makes no difference as `for` loops are translated to `while` loops during the transformation to CIL.

The scenario modeling language is a prototype implemented in Python. The actual specification of a scenario is directly given in Python code. A library allows to express different *action execution patterns*. These patterns allow to express if and when which actions can be executed in a scenario. Examples are patterns such as sequential execution of actions, optional actions, permutations of actions, and executing actions multiple times in a row. A detailed description of the available action execution patterns is given in subsection *3.3.2*. Actions are provided as functions within the C code and can be referenced from within the scenario. For the test setup, we chose to only use parameterless functions with a result type of `void`.

Fig 3.4 shows a schematic of the scenario generation process. The scenario code is generated from the scenario description under the influence of the feature selection. Then, the resulting module is introduced to the code of the library variant thereby forming a variant that actually uses available library functions. The action mapping describes which actions are introduced by which features. When generating a scenario, all actions that are not part of the feature selection are omitted. Otherwise, dangling function references would be present in the variant. An alternative would be to compose the scenarios in a feature-oriented way. For the case study however, the former approach proved to be more feasible. It is assumed that scenario actions do not get refined by other features. Different scenarios can either live in independent files or can be grouped together for convenience.

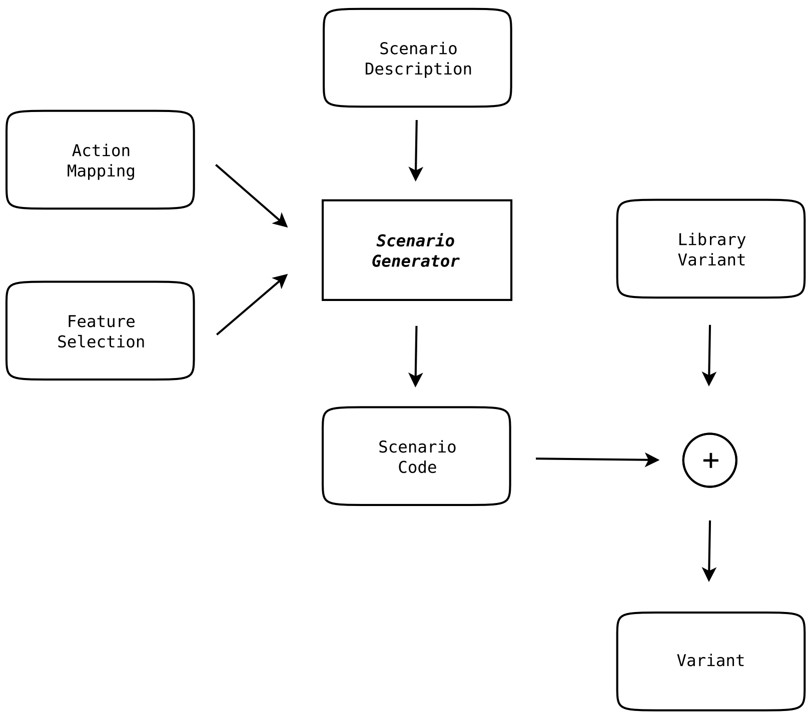


Figure 3.4: Scenario code generation.

### 3.3.1 Action Execution Patterns

This section describes the action execution patterns that are currently available. The implementation is currently lightweight and can be easily adapted to also support additional patterns should the need arise.

#### Actions

Actions are specified as quoted strings. Actions are assumed to take no arguments. To use actions with parameters or if the result value is needed the `C` execution pattern is supplied that can be used to embed arbitrary C code into scenarios. As an example, the action execution pattern `'myaction'` would be translated to `myaction()` in the resulting scenario code.

#### `Scenario`

This pattern is used to model sequential execution. If the pattern is executed, all elements are executed in sequence.

#### `Optional`

This pattern accepts only one nested pattern. As the name suggests, the nested pattern may or may not be executed. It is translated to an `if` block using a nondeterministic condition that encloses the action.

#### `Star`

This pattern also accepts only one nested pattern. Its function is that of the Kleene-Star known from regular expressions: The enclosed pattern may be run multiple times in a row or not at all.

#### `Plus`

Also borrowed from regular expressions, this pattern also allows only one nested pattern that may be executed multiple times in a row but is executed at least once.

#### `Permutation`

Nested patterns are executed in random order. Each pattern is executed exactly once.

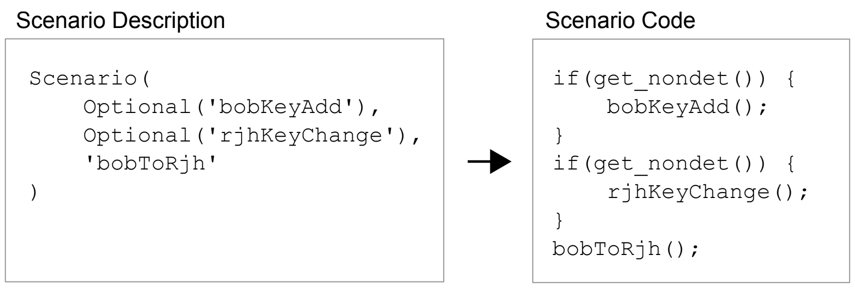
#### `SubsetPermutation`

Analogous to `Permutation`, nested patterns are executed in random order. However, some patterns may be skipped.

#### `C`

This pattern allows raw C code to be embedded. The code is given as sole argument and needs to be quoted.

## 3.3.2 Example



```
Scenario(
    Optional('bobKeyAdd'),
    Optional('rjhKeyChange'),
    'bobToRjh'
)
```

```
if(get_nondet()) {
    bobKeyAdd();
}
if(get_nondet()) {
    rjhKeyChange();
}
bobToRjh();
```

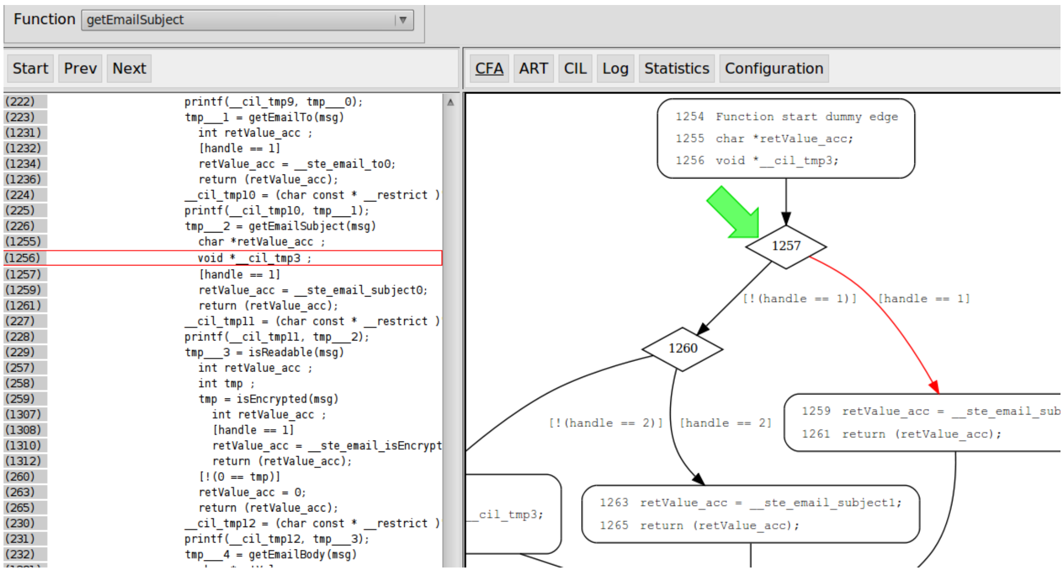
Figure 3.5: Translation of a scenario.

Figure 3.5 shows a scenario description and the code that is generated from it. `bobKeyAdd` and `rjhKeyChange` may be skipped. `bobToRjh` is always called last and may not be skipped. Assuming that `bobKeyAdd` and `rjhKeyChange` are provided by feature `Keys` these actions would have been omitted from the resulting code if the `Keys` feature was not selected. This way, the scenario modeling language can be used with FOSD while the scenarios can be expressed as if all features are always present. During scenario generation, the action mapping is used to decide if the generated code should contain a call to a particular action or if the action is omitted.

## 3.4 Counterexample Interpretation

After the analysis has been performed the result needs to be interpreted. In case the model checker has deemed the program to be safe it is guaranteed to adhere to the specification given that the model checker has been configured correctly and the specification contains no side effects. If the model checker does not terminate or runs out of system memory no assumptions must be made about the correctness of the program in respect to the specification used. In case of a bug, the programmer is now facing the task of interpreting the counterexample to locate the problem and to be able to construct a patch. In the context of FOSD, this is challenging as the variant has been analyzed in composed form. Therefore, the error-path and additional information created by the model checker is referring to the composed variant.

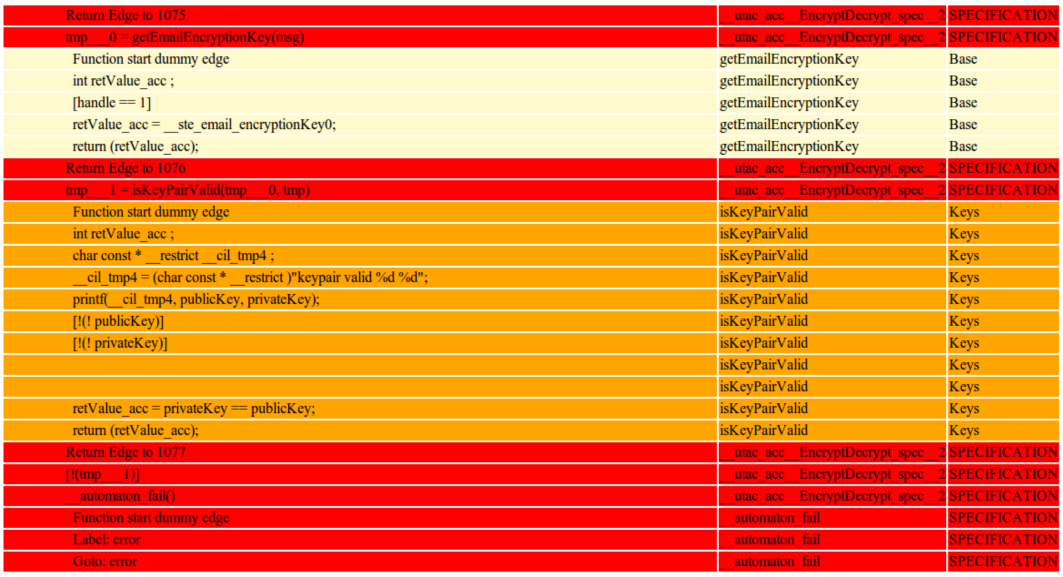
As previously stated however, FOSD should be employed throughout the whole development process. As a consequence —because we do not want to depart from that fundamental idea— we need a way to effectively analyze the counterexample in a decomposed form. Therefore, we reuse the concept of CIDE: Features in an already composed system are decomposed visually by using different background colors for code originating from different features. In this case, we applied the concept to the error-path returned by CPACHECKER. However, it should be possible to also use this approach to decompose the CFA and ART respectively.



```
(222)      printf(__cil_tmp9, tmp___0);
(223)      tmp___1 = getEmailTo(msg)
(1231)       int retValue_acc ;
(1232)       [handle == 1]
(1234)       retValue_acc = __ste_email_to0;
(1236)       return (retValue_acc);
(224)      __cil_tmp10 = (char const * __restrict )
(225)      printf(__cil_tmp10, tmp___1);
(226)      tmp___2 = getEmailSubject(msg)
(1255)       char *retValue_acc ;
(1256)       void *__cil_tmp3 ;
(1257)       [handle == 1]
(1259)       retValue_acc = __ste_email_subject0;
(1261)       return (retValue_acc);
(227)      __cil_tmp11 = (char const * __restrict )
(228)      printf(__cil_tmp11, tmp___2);
(229)      tmp___3 = isReadable(msg)
(257)        int retValue_acc ;
(258)        int tmp ;
(259)        tmp = isEncrypted(msg)
(1307)         int retValue_acc ;
(1308)         [handle == 1]
(1310)         retValue_acc = __ste_email_isEncrypt
(1312)         return (retValue_acc);
(260)        [!(0 == tmp)]
(263)        retValue_acc = 0;
(265)        return (retValue_acc);
(230)      __cil_tmp12 = (char const * __restrict )
(231)      printf(__cil_tmp12, tmp___3);
(232)      tmp___4 = getEmailBody(msg)
```

Figure 3.6: Counterexample viewer.

In the course of this thesis a counterexample viewer was developed for CPACHECKER. Figure 3.6 shows a screenshot of the viewer. On the left we see the error path. On the right, we can switch between views of the control flow automaton, abstract reachability tree, the CIL code, and the log and result messages generated by CPACHECKER. Using the previous and next buttons on the left we can step through the error path and see the current state on the control flow automaton, abstract reachability tree, and CIL code. A viewer is generated by a Python script that post-processes the output of CPACHECKER. The viewer itself is a JavaScript application, that can be executed in the browser.



| | | |
|---|---|---|
| Return Edge to 1075 | __utac_acc__EncryptDecrypt_spec__2 | SPECIFICATION |
| tmp___0 = getEmailEncryptionKey(msg) | __utac_acc__EncryptDecrypt_spec__2 | SPECIFICATION |
| Function start dummy edge | getEmailEncryptionKey | Base |
| int retValue_acc ; | getEmailEncryptionKey | Base |
| [handle == 1] | getEmailEncryptionKey | Base |
| retValue_acc = __ste_email_encryptionKey0; | getEmailEncryptionKey | Base |
| return (retValue_acc); | getEmailEncryptionKey | Base |
| Return Edge to 1076 | __utac_acc__EncryptDecrypt_spec__2 | SPECIFICATION |
| tmp___1 = isKeyPairValid(tmp___0, tmp) | __utac_acc__EncryptDecrypt_spec__2 | SPECIFICATION |
| Function start dummy edge | isKeyPairValid | Keys |
| int retValue_acc ; | isKeyPairValid | Keys |
| char const * __restrict __cil_tmp4 ; | isKeyPairValid | Keys |
| __cil_tmp4 = (char const * __restrict )"keypair valid %d %d"; | isKeyPairValid | Keys |
| printf(__cil_tmp4, publicKey, privateKey); | isKeyPairValid | Keys |
| [!(! publicKey)] | isKeyPairValid | Keys |
| [!(! privateKey)] | isKeyPairValid | Keys |
| | isKeyPairValid | Keys |
| | isKeyPairValid | Keys |
| retValue_acc = privateKey == publicKey; | isKeyPairValid | Keys |
| return (retValue_acc); | isKeyPairValid | Keys |
| Return Edge to 1077 | __utac_acc__EncryptDecrypt_spec__2 | SPECIFICATION |
| [!(tmp___1)] | __utac_acc__EncryptDecrypt_spec__2 | SPECIFICATION |
| __automaton_fail() | __utac_acc__EncryptDecrypt_spec__2 | SPECIFICATION |
| Function start dummy edge | __automaton_fail | SPECIFICATION |
| Label: error | __automaton_fail | SPECIFICATION |
| Goto: error | __automaton_fail | SPECIFICATION |

Figure 3.7: Excerpt of a decomposed view of an error-path.

Figure 3.7 shows a decomposed error-path. Different features are indicated using different colors as known from CIDE. The first column contains the elements of the error-path. The second column shows the active function that contains the code. The third column shows the feature that introduced the code. Code introduced by specifications is shown in red and can be distinguished from feature code as the third column is set to `SPECIFICATION`.

CHAPTER

FOUR

# MODEL CHECKING OF PRODUCT LINES USING VARIABILITY-ENCODING

Even though model checking of individual variants is beneficial, it leaves room for improvement. In settings, where the number of variants is small it could be implemented by automatically performing a set of configured checks on these variants whenever the product line changes in the spirit of continuous integration testing. However, there may be situations where this approach is unfeasible due to the computation effort of checking all variants against all specifications. Then, it might be more feasible to investigate model checking of the product line as a whole as opposed to checking the set of possible variants individually. For a given specification, the model checker would then check all possible variants —the product line— in a single run. Conceptually, this can be approached from two angles: A specialized model checker could be constructed. The model checker would be able to directly operate on the product line and would need access to the feature model and incorporate knowledge about the feature composition process. The alternative is to create an instrumented program that incorporates all possible variants and the feature selection in accordance to the feature-model and the composition rules. This is the approach we will investigate in the following.

We will refer to the process of creating the instrumented program as variability-encoding because the variability of the product line is encoded in the resulting program. The concept is also described briefly in [ASWB2011]. The assumption is that incorporating all variants in one single program and checking it in a single run may reduce the time to find an interaction or prove the safety of a product line as this eliminates the need to generate all variants and weave the specifications on each one. Also, the model checker needs to be run only once as opposed to multiple times when checking all variants. As individual variants may share large parts of their code these checks may also need to re-investigate code that has already been analyzed in related variants. On the other hand, variability-encoding creates a program that is bigger than all variants individually: all variants are incorporated into the variability-encoded product line including additional control flow that is necessary to select a particular variant at runtime. Therefore, it is not apparent if variability-encoding is beneficial to model checking of software product lines as the advantages of not having to generate all variants and possibly eliminating redundant checks weigh against the disadvantage of having to cope with a possibly much larger program to investigate. Hence, we will compare both approaches in the course of a case study in chapter 5.

Fig 4.1 shows an overview of the proposed process of variability-encoding. Similar to the first approach features consist of code and specifications. These parts are specified in exactly the same format. Therefore, variability-encoding can easily be applied to a product line instead of or in combination to checking individual variants. In contrast to composition, variability-encoding always uses all features. To construct the variability-encoded product line the variability-encoder needs to know the composition order. Also, because only valid variants must be considered, it needs access to the feature model. Analogous to the approach of checking all variants individually we use the variability-encoder to select the applicable specifications. As variability-encoding always considers all features no specifications are filtered out. However, hybrid approaches would be possible where only a subset of features is variability-encoded. The following sections explain the technique of variability-encoding. First, variability-encoding is defined followed by an abstract example. Then, we show that proving the safety of the variability-encoded program is equivalent to proving the safety of all possible variants of the product line.

## 4.1 Definition

Variability-encoding has been implemented by modifying FSTCOMPOSER. When composing variants feature selection happens at composition time. The basic idea of variability-encoding is to postpone the process of feature selection until early runtime. Then, —at application startup time— a variant is selected by setting a set of variables called *feature variables*. Each feature of the product line is represented by a feature variable. If set to `1`, the corresponding feature is selected; if set to `0`, the corresponding feature is not part of the selected variant. After initialization, the

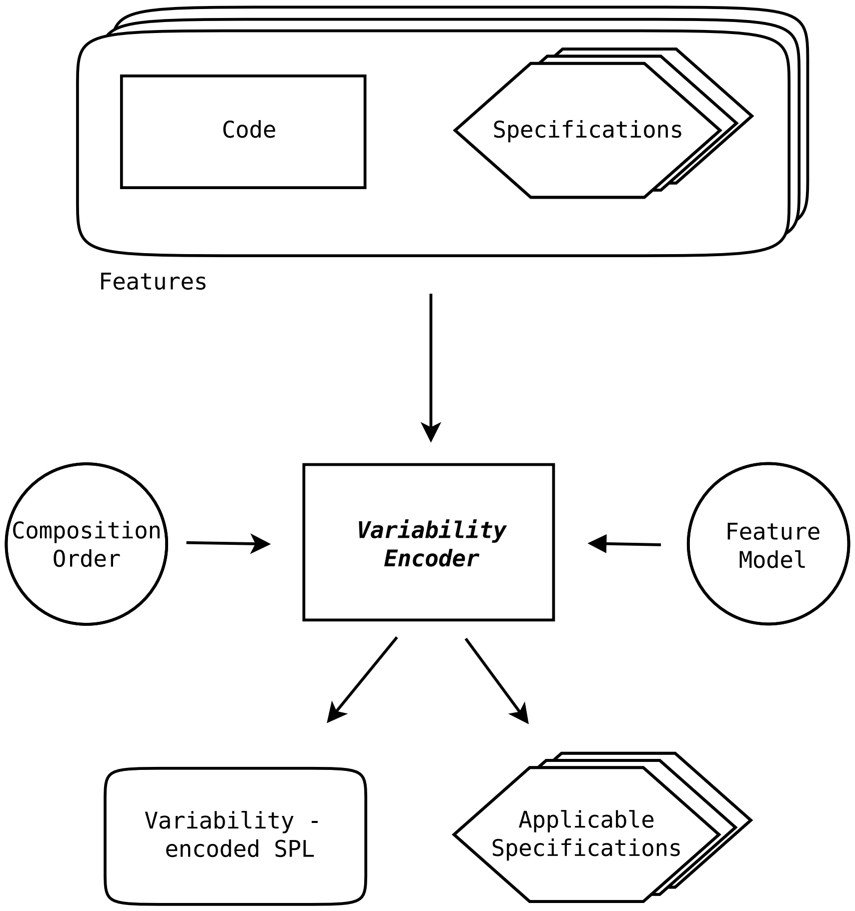


Figure 4.1: Variability-encoding of a product line.

values of the individual feature variables are not changed again —they are considered constants. The code of a specific feature is only executed if its feature variable has been set to `1`. This way the application behaves like a variant that has been composed using conventional composition.

We recall that function nodes are composed by applying the `FunctionRefinement` composition rule: the old node is given a new name $\alpha$, the new node is introduced and references to `original` within its body are pointed to $\alpha$.

For variability-encoding `FunctionRefinement` is exchanged for an alternative implementation to postpone the feature selection process to the early runtime of the program: if a function is to be replaced, it may still be needed by variants in which the refining feature $f$ is not present. Therefore, its function name $\alpha$ is rewritten to $\alpha_{before}$ and the new function node is introduced with name $\alpha_{role}$. As before, calls to `original` within the new function are rewritten to $\alpha_{before}$. A *feature switch* is generated with name $\alpha$. A feature switch is a function that dispatches to a variants implementation of the function depending on feature $f$ being selected or not, e.g. for a function `alpha` returning *int* that accepts no parameters the generated feature switch would be:

```
int alpha() {
    //choose implementation depending on
    //the value of feature variable of f
    if (__SELECTED_FEATURE_f) {
        return alpha_role_f();
    } else {
        return alpha_before_f();
    }
}
```

A specific variant of the product line can now be created by initializing the feature variables accordingly. To be able to check all variants in one run we set the feature variables to uninitialized values. The software model checker must now take all feature combinations into account. To exclude variants that do not satisfy the conditions imposed by the feature-model, we encode the feature-model into the feature selection process as outlined in the following pseudo code:

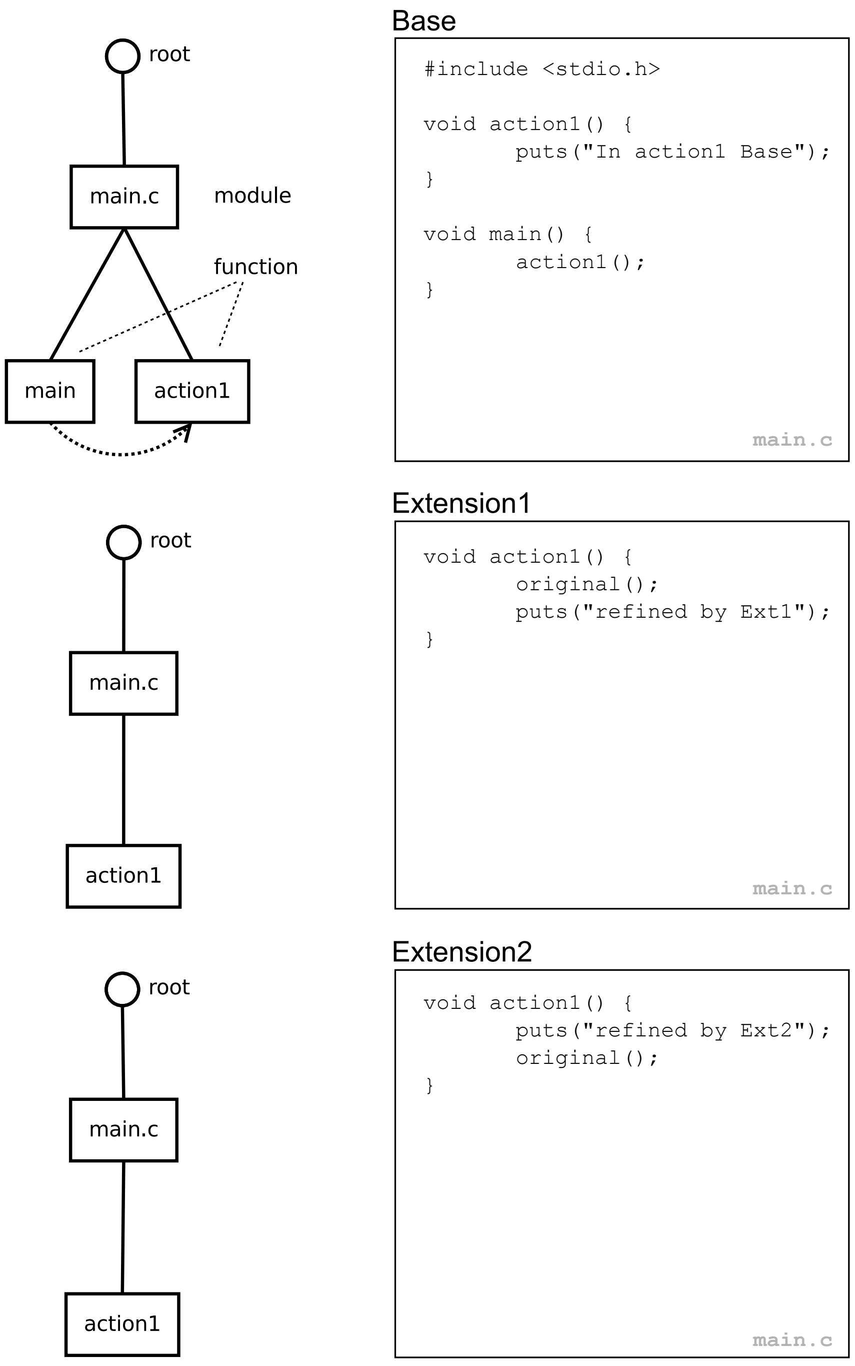
```
int main() {
    select_features();
    //check that selected features form a valid variant
    if (valid_product()) {
        //run the variant
        return original_main();
    }
}
```

Conceptually, `select_features` picks an arbitrary combination of features; if `valid_product` —a translation of the feature-model— is true the variant is executed. This way, the model checker needs to take all valid variants (and no more) into account when checking the safety of the product line. If a bug is found in the variability-encoded product line, variants that also exhibit the bug can be derived from the counterexample given by the model checker. If the variability-encoded product line is safe, then so are all valid variants.

## 4.2 Example

In the following, the normal composition process is compared to the construction of variability-encoding.

Let us assume a product line containing the three features `Base`, `Extension1`, and `Extension2` that are composed in that order. Each feature contains a single module named `main.c`. The contents of these files and the FSTs of the features are listed in figure 4.2. For brevity, include directives are not shown in the following FSTs. Also the order in which child nodes are depicted has been chosen for readability and does not represent the order in which nodes are placed by FST-COMPOSER. Dotted arrows between function nodes mean that a function **may** call a function it points to.



Base

```
#include <stdio.h>

void action1() {
       puts("In action1 Base");
}

void main() {
       action1();
}
```

Extension1

```
void action1() {
       original();
       puts("refined by Ext1");
}
```

Extension2

```
void action1() {
       puts("refined by Ext2");
       original();
}
```

Figure 4.2: Example FSTs and corresponding code

We will now compare the composition of a variant $v$ containing `Base` and `Extension1` to the variability-encoded product line. For variability-encoding all features are used. By selecting `Base` and `Extension1` using the introduced feature variables a variant is produced that behaves like the traditionally composed variant $v$.

In the following, variability-encoding is denoted by $\uplus$ . As before, superimposition is denoted by $\bullet$.

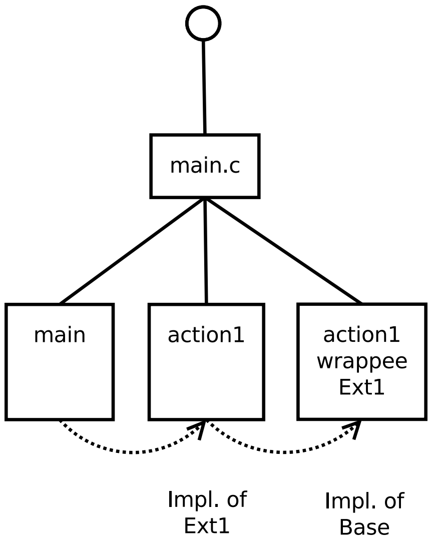


Figure 4.3: FST of `Extension1` $\bullet$ `Base`.

Figure 4.3 shows the resulting FST for `Extension1` $\bullet$ `Base`. Function `main` calls the implementation of `action1` defined in `Extension1`. The call to `original` is rewritten to call the implementation of `action1` provided by `Base`.

For variability-encoding two steps are necessary. Figure 4.4 shows the FST of the variability-encoded SPL after the first step `Extension1` $\uplus$ `Base`. Function `main` calls the feature switch for `Extension1`. If `Extension1` is not selected, the feature switch dispatches to the implementation of `Base` thereby skipping `Extension1`. If `Extension1` is selected, the feature switch dispatches to the implementation of `action1` provided by `Extension1`. Analogous to composition, this implementation calls the implementation of `Base` for calls to `original`.

Now, the FST of `Extension2` is superimposed on the FST that has been obtained

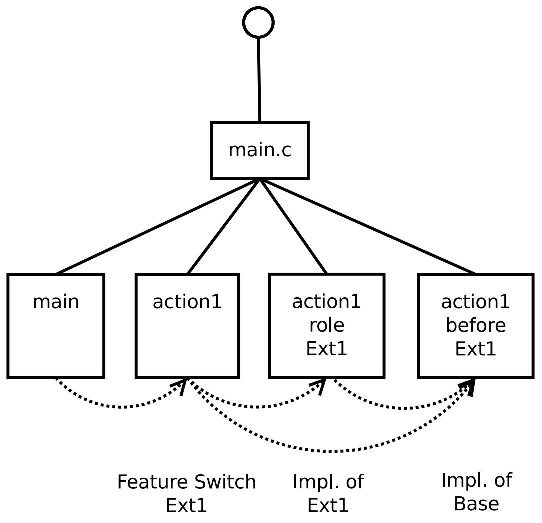


Figure 4.4: FST of `Extension1` ⊎ `Base`.

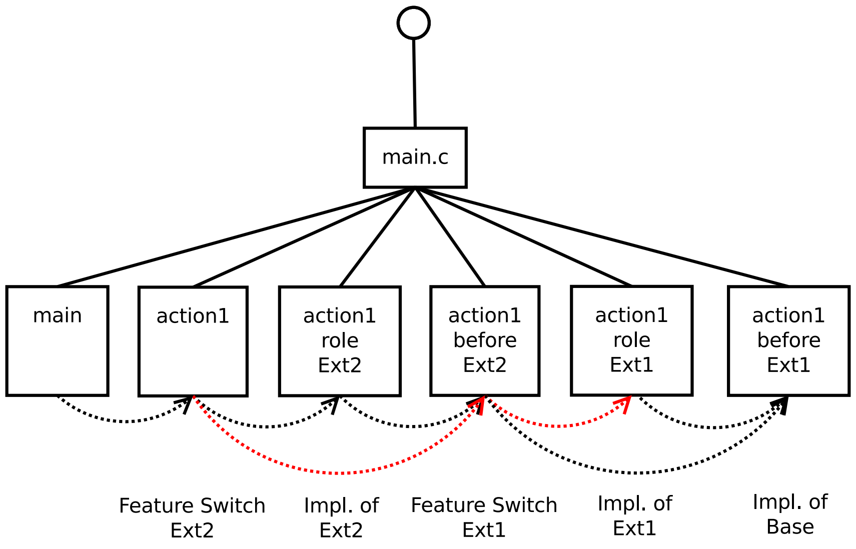


Figure 4.5: FST of `Extension2` ⊎ `Extension1` ⊎ `Base`.

in the previous step. Figure 4.5 illustrates the resulting FST. As before, the refinement of `action1` by `Extension2` results in the addition of a feature switch, a node for the implementation of `action1` provided by `Extension2`, and the renaming of the current implementation of `action1` that resulted of step one. If `Extension2` is selected using its feature variable, the implementation given in `Extension2` is called. If `Extension2` is not selected, the feature switch of `Extension1` takes over: if a function is refined by multiple features, feature switches form a cascade as a direct result of the application of the composition rule. Therefore, if only `Base` and `Extension1` are selected, the feature switches dispatch only to the implementations of `action1` provided by these features. This configuration is depicted with the red dotted arrows in figure 4.5. As the feature switches and feature selection process of the variability-encoded product line have no side-effects other than selecting the variant, the variability-encoded product line behaves like the composed variant.

## 4.3 Correctness Of Variability-Encoding

In the following, we will show that a variability-encoded SPL behaves like a composed variant if the feature variable values reflect the feature selection of the composed variant. We will therefore formalize the concept of variability-encoding using the following definitions:

Let $SPL$ be a product line consisting of a set $F$ of features of size $n$ with composition order $K = (f_1, \ldots, f_n)$ and the set $FM \subseteq \mathcal{P}(F)$ of valid feature combinations as the feature-model.

Feature selection of a variability-encoded product line is performed by the transformation $\sigma_X : V \to P$ with $V$ being a set containing the variability-encoded product line and intermediate results of the variability-encoding process, $P$ being the set of possible variants, and $X \in FM$ a feature selection.

By applying $\sigma_X$ to a variability-encoded product line, all feature variables of the features $f \in X$ are initialized with $1$ while all other feature variables are set to $0$.

We will use $P_1 \sim P_2$ to indicate that a program $P_1$ behaves like a program $P_2$.

### 4.3.1 Assumptions

The construction presented here relies on the following assumptions:

- The product line is type safe, i.e., every variant that is valid according to the feature-model is type safe. This means that the references of a feature $f$ are valid (non-dangling) in all variants that contain $f$, according to a feature-model. Also, fields provided by structures are only accessed using the fields name and not by memory address. As all fields of all structures are always available in a variability-encoded product line, access by memory address potentially accesses an unintended location.
- Variability-encoding only allows functions to be refined. Principally, the composer could also be used to refine global variables i.e. redefine their initial value.
- Included header files must not have side-effects other than declarations e.g. they may not redefine preprocessor macros.

### 4.3.2 Structures, Fields, And Global Variable Declarations

Variability-encoding uses standard composition rules for composing nonterminal nodes such as structures and terminal nodes such as includes, global variable declarations, and fields within structures. Therefore, the difference between the conventional and the variability-encoded composition for these cases is only that variability-encoding composes all features. This does not impose behavioral side-effects upon any variant part of the variability-encoded product line as the product line is type safe and header files are also side-effect free. Therefore, the variability-encoding of structures, fields, and global variable declarations preserves the behavior of every variant that is part of the product line. In addition, we now need to show that the variability-encoding of functions also preserves the behavior of all variants of the product line.

### 4.3.3 Variability-Encoding Of Functions

To show the correctness of variability-encoding for the composition of functions, we will use structural induction over the composition process of the variability-encoded product line:

$$v = \biguplus_{i=1}^{n} f_i = f_1 \uplus f_2 \ldots \uplus f_n$$

Therefore, let $v_k$ be the k-th intermediate step of the variability-encoding process:

$v_k = \biguplus_{i=1}^{k} f_i = f_1 \uplus f_2 \ldots \uplus f_k, 1 \leq k \leq n$

We will also use $C_X$ to denote a conventionally composed variant containing all features $f \in X \subseteq F$ (as with variability-encoding, features are composed in the composition order $K$).

**Base:** Let $k = 1$

Obviously, we see:

$v_1 = f_1$ and $C_{\{f_1\}} = f_1$ (1).

No feature switches were introduced. As $\sigma$ has no side effects, we can deduct that:

$\sigma_{\{f_1\}}(v_1) = \sigma_{\{f_1\}}(f_1) \sim f_1$ and with (1) we can state that $\sigma_{\{f_1\}}(v_1) \sim C_{\{f_1\}}$.

Therefore, variability-encoding of functions is correct for $k = 1$.

**Induction Hypothesis**: The intermediate steps of variability-encoding are constructed correctly —that is, for all $k$ and all $x \in FM \cap \mathcal{P}(\{f_1 \ldots f_k\})$ the following statement holds:

$\sigma_X(v_k) \sim C_X$

**Inductive Step**

$v_{k+1} = v_k \uplus f_{k+1}$

To construct $v_{k+1}$, we need to add the functions contained in $f_{k+1}$ to the result $v_{k+1}$. Here, we need to consider two cases: Functions are either introduced or they refine functions already present in $v_k$. As additional function declarations impose no side effect on the behavior of a program, we can make the same argument as made before for structures and global variable declarations. On the other hand, if a function is refined by $f_{k+1}$ both the old and new implementation are made available. Like with conventional composition calls to `original` are pointed to the old implementation. In contrast to conventional composition, the old implementation remains in the code base. As mentioned before, additionally available functions do not modify the behavior of variants in which this function would not be present. The introduced feature switch that dispatches to the old or to the new implementation has no other side effects on the behavior. Therefore, if $f_{k+1}$ is not selected, the old implementation of the function is called and according to the induction hypothesis the behavior of that function is correct. If $f_{k+1}$ is selected the feature switch dispatches to the new implementation. Then, the control flow of the program is exactly the same as with conventional composition until the next feature switch is encountered.

This feature switch is already covered by the induction hypothesis. Therefore, the construction of variability-encoding is also correct for functions.

## 4.4 Necessary Changes To The Scenario Modeling Language

To use the scenario modeling language presented in section 3.3 with variability-encoding, we need to modify the translator that produces the scenario code. To check individual variants we specified the scenario as if all features would have been available. Then, by supplying the action mapping and the feature selection to the translator we were able to generate scenario code for variants where specific actions were not available. The translator omitted these actions in the generated code and therefore did not create dangling references to functions. With variability-encoding all features are considered. However, depending on the values of the feature variables some actions must not be used to preserve the behavior of the individual variants. Therefore, instead of omitting the actions in the scenario we guard each action using the feature variable of the feature that introduced the action: if a variant is selected during the postponed feature selection that contains the feature that introduces the action, the action is executed. If the feature is not part of the selected variant the execution of the action is skipped. This way, we can reuse all scenarios that have been specified for model-checking individual variants with variability-encoding. The generated code is different however as actions are guarded by their feature variable instead of omitting them completely. Again, it has to be noted that actions may not be refined by other features. This would require additional guards in the scenario code and is not currently supported by the scenario translator.

CHAPTER

FIVE

# THE E-MAIL SYSTEM CASE STUDY

This chapter discusses the application of the concepts mentioned in the previous chapters to the aforementioned e-mail system product line.

## 5.1 System Description

The product line that is investigated here consists only of the e-mail client features of the e-mail system described by Hall. Therefore, we can only investigate nine of Hall's 27 feature interactions. We will also check for another interaction that was found in the course of the case study between the `Decrypt` and the `Forward` feature. It occurs when a host receives an email that he cannot decrypt and has configured automatic message forwarding. Because he cannot decrypt the message forwarding the message may not be reasonable. A detailed description of the interactions mentioned here has been done by Hall [Hall2005]. To be able to compare our approaches for cases where feature interactions are present and cases where the product line is safe we use different scenarios: for all investigated interactions, we use one scenario that leads to a feature interaction and an alternative scenario where no interaction occurs. The following table displays the feature interactions of the e-mail client product line:

| Interaction | Interacting features | Specification feature | $b$ | $n$ |
|---|---|---|---|---|
| 0 | Decrypt, Forward | Forward | 8 | 20 |
| 1 | AddressBook, Encrypt | Encrypt | 8 | 16 |
| 3 | Sign, Verify | Sign | 16 | 16 |
| 4 | Sign, Forward | Sign | 8 | 16 |
| 6 | Encrypt, Decrypt | Encrypt | 16 | 16 |
| 7 | Encrypt, Verify | Verify | 8 | 16 |
| 8 | Encrypt, Autoresponder | Encrypt | 8 | 16 |
| 9 | Encrypt, Forward | Encrypt | 8 | 16 |
| 11 | Decrypt, AutoResponder | Decrypt | 16 | 16 |
| 27 | Verify, Forward | Verify | 8 | 16 |

Interaction is the number used by Hall. The specification that detects a particular interaction is located in the feature given in the specification feature column. $b$ is the number of variants that contain the feature interaction. $n$ is the number of variants that needs to be checked, i.e. the number of variants where the specification feature is present.

To be able to compare the results of different model checkers the code of the e-mail system product line has been simplified to only contain constructs that are supported by the model checkers used. In detail, the code does not contain structures or arrays. Also, strings have been exchanged with integers.

## 5.2 Comparison Of Model-Checking Individual Variants And Variability-Encoding

To compare the two alternative work-flows of model checking all possible variants and model checking the variability-encoded product line, we will now discuss the approach taken for the comparison:

Comparisons are made on a per-interaction basis. Specifically, we measure the times needed to find an interaction or to prove the safety. As each specification is associated with a feature both approaches need to consider only feature combinations that contain this feature; all other combinations cannot contain error labels and can therefore be omitted. First, we measure the time to discover each feature interaction. For variability-encoding, we measure the time $T$ to find an interaction by checking the encoded product line against a particular specification. For checking the variants, we need to generate and check these in a predefined order. After the first erroneous feature combination has been identified, no further checks are performed because the identified interaction invalidates all existing safety proves (at least those in which the corresponding features are present). We note that for checking individual variants the absolute time to find an interaction is dependent on the order in which the checks are executed: It may be that, incidentally, we pick an order that contains an interaction early on, so we obtain the result by performing only a few checks. Or, it may be that only the very last feature combination we check contains the interaction, so the time to detect the interaction is the sum of the times for checking all applicable variants of the product line. Therefore, we measure the time $T_j$ to generate and check each possible feature combination $C_j \in C_1 \dots C_n$ for the same interaction. Then, we calculate the total time $P_k$ for each permutation $\pi(T_1 \dots T_n)$ of the checking order: If a feature combination $C_j$ is the first that contains an interaction, we sum the times $T_i$ for all $i \leq j$. With the safe scenarios, we obviously need to sum up all individual checks as no feature interactions are present in these cases.

## 5.3 Measurement Results

This section now presents the results of the measurements that were done using the model checkers CPACHECKER and CBMC. All tests have been executed on a machine with 32 GB available system memory. The file-system where output files were written was located on the local network.

### 5.3.1 CPACHECKER

CPACHECKER supports different approaches to model checking. In the following, we investigate two of the available configurations.

### Symbolic Predicate Abstraction

In this setup, the `symPredAbsCPA-bmc` configuration of CPACHECKER is used. Using it, CPACHECKER performs bounded model checking.

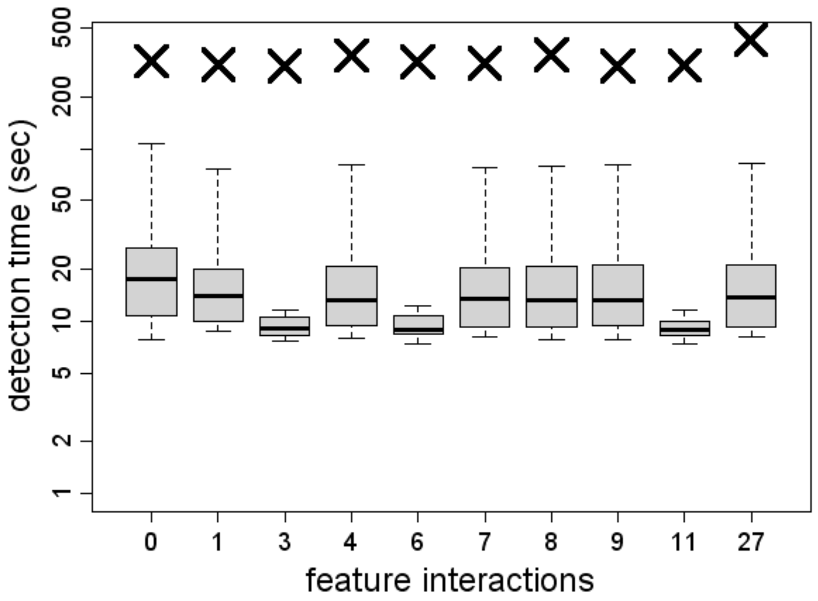


Figure 5.1: Check times for `symPredAbsCPA-bmc` with bug scenarios.

Figure 5.1 shows the results of the experiment. The time needed to find an interaction in the variability-encoded product line is depicted as a black cross. The possible times to find the interaction by checking the variants using the aforementioned strategy are represented as box-plots that show the minimum, maximum, median, and inner 50% of the values. As we can see, variability-encoding is always slower than checking the individual variants in these cases. Due to our strategy to abort after an interaction is identified and the $b/n$ ratio the median is close to the minimum value.

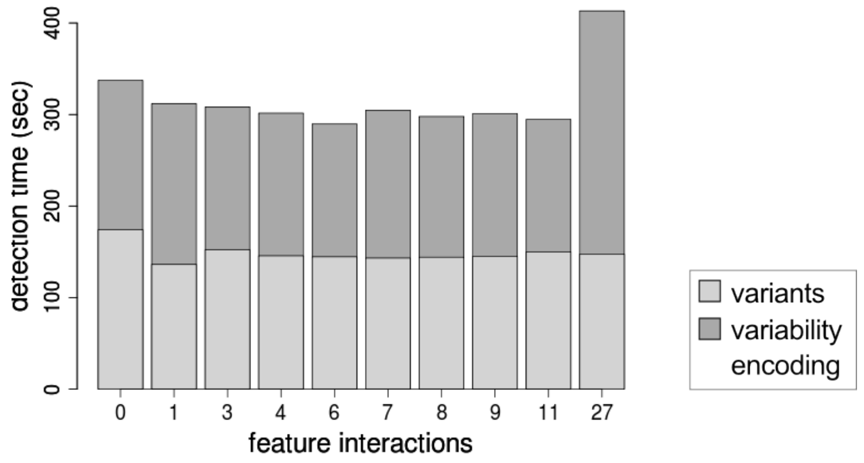


Figure 5.2: Check times for `symPredAbsCPA-bmc` with safe scenarios.

In figure *5.2*, we see the results when using `symPredAbsCPA-bmc` with scenarios that do not result in a feature interaction. As with the bug scenarios, variability-encoding is always slower to detect the interaction. However, it is only slower by a factor of two as opposed to factor 25 for the bug scenarios.

## Explicit Analysis

Here, we use the `explicitAnalysisInf` configuration of CPAchecker.

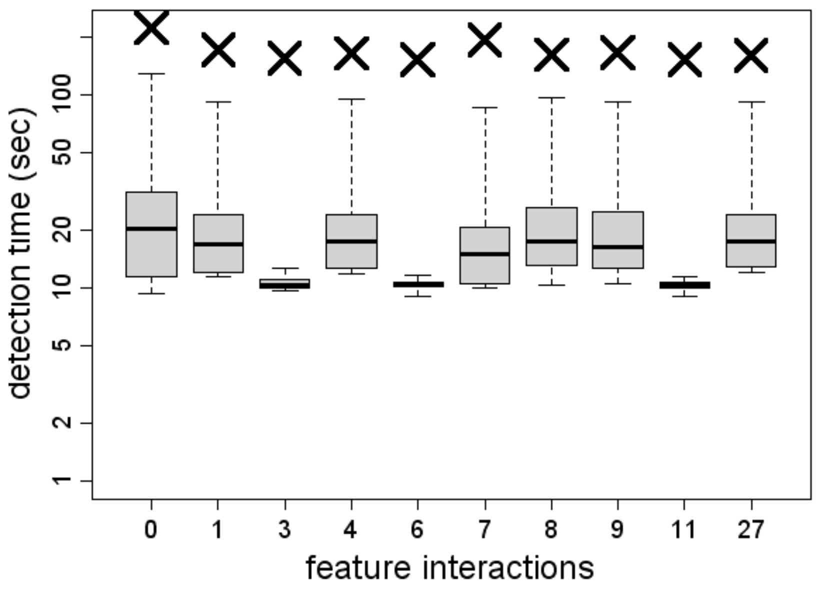


Figure 5.3: Check times for `explicitAnalysisInf` with bug scenarios.

As can be seen in figure *5.3* finding the interactions in the variability-encoded product line is also slower when using the `explicitAnalysisInf` configuration in the investigated cases. However, for these cases `explicitAnalysisInf` performs better than `symPredAbsCPA-bmc`.

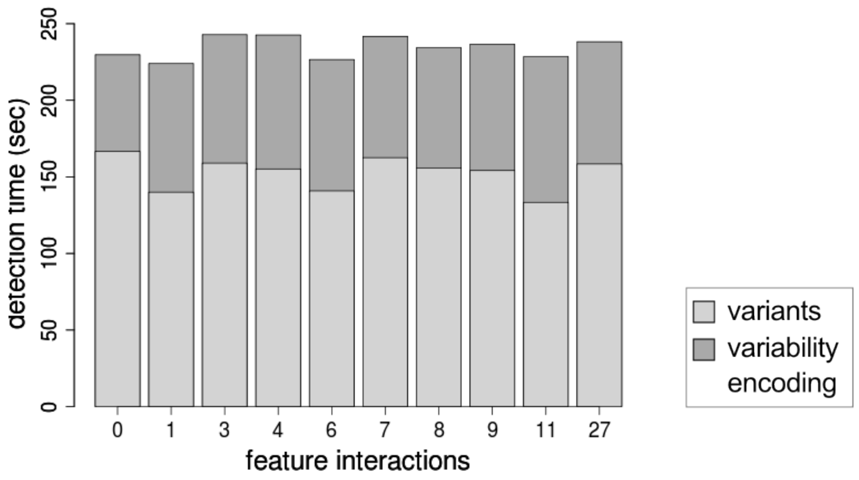


Figure 5.4: Check times for `explicitAnalysisInf` with safe scenarios.

Figure *5.4* now shows the times needed with `explicitAnalysisInf`. Still, the variability-encoding approach is slower than checking all applicable variants. As with bugs, the difference between both approaches is smaller than with `symPredAbsCPA-bmc`.

### 5.3.2 CBMC

In this section, we will now use the software model checker CBMC instead of CPAchecker. As mentioned before, the e-mail system does not contain loops or recursion. Therefore, setting a bound is not necessary as no constructs need to be unwound.

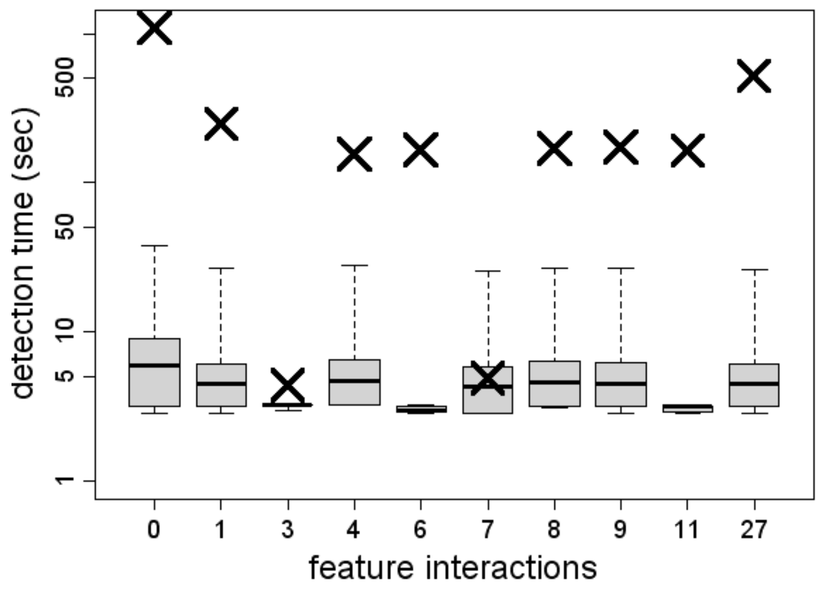


Figure 5.5: Check times for CBMC with scenarios leading to a bug.

Analogous to the CPACHECKER results, model checking of the variability-encoded SPL is mostly slower that checking individual variants. However, if we look at interactions 3 and 7 in figure 5.5, we notice that in these cases the results of variability-encoding are better than with CPACHECKER. For interaction 7, checking the variability-encoded product line is actually close to the median and within the inner 50% of the values.

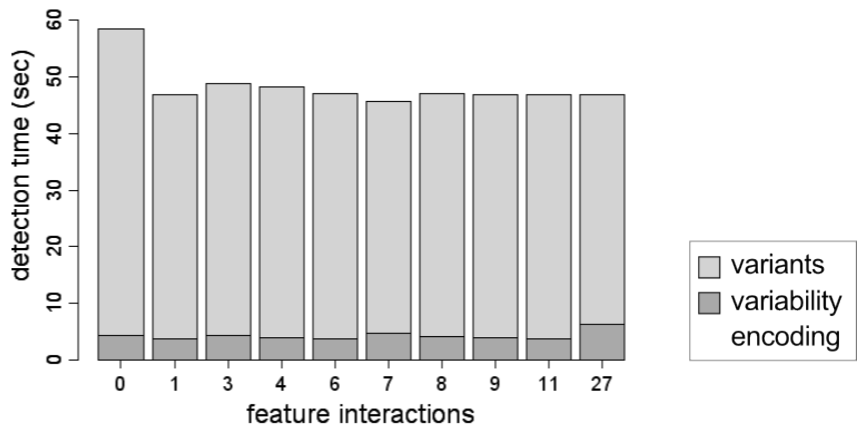


Figure 5.6: Check times for CBMC with scenarios that are safe.

For the safe cases depicted in figure 5.6, we now see the opposite results of the previous experiments. Here, checking the variability-encoded SPL is about factor 10 faster than checking the individual variants. Together with the results of interaction 3 and 7 these results suggest that variability-encoding may indeed be beneficial in certain cases.

## 5.4 Discussion

As can be seen, model checking the variability-encoded product line is slower than checking individual variants except for the safe scenarios when CBMC is used as model checker. As mentioned before, variability-encoding creates a program that is more complex than any individual variant because all features are used and control-flow is added in the form of feature switches and the postponed feature selection process. On the other hand, variability-encoding makes it unnecessary to generate and check all variants individually. Therefore, the strength of variability-encoding can come to play only if there are multiple checks necessary. If the product line does not contain an interaction all variants need to be proven safe. Here, it is most likely that model checking the variability-encoded product line is faster. With CBMC, this is actually the case for the investigated product line. If the product line contains an interaction it is however still possible that variability-encoding is beneficial. If the order in which checks are executed does not contain an interaction early on, a possibly huge number of model checks is necessary to detect the interaction. Especially, the number of necessary checks can be expected to be high if the interaction is only present in a small number of variants. For the cases where an interaction is present in the e-mail client product line the number of variants $b$ that contain an interaction is at least half as high as the number of variants $n$ that need to be considered. Therefore, there is at least a $50\%$ chance that we pick a checking order that requires only a single check to reveal the interaction. For other product lines, the ratio of $b/n$ might be lower which would increase the chances that model checking of the variability-encoded product line outperforms checking the individual variants. Unfortunately, as $b$ is not known beforehand, this ratio cannot be used as a criteria to decide when to use variability-encoding. It could also be argued that for values of $b/n$ that are close to 1 standard testing techniques are likely to reveal the presence of interactions and therefore make model checking as such unnecessary. Also, it has to be noted that the $b/n$ ratio is not the only factor that decides if variability-encoding is beneficial: The complexity of the feature code and the specific technique used by the employed model checker are also relevant. However, as these factors are dependent on the investigated product line, we cannot make a general statement here. Therefore, to be able to assess if variability-encoding is useful for model checking feature-oriented product lines more empirical data on the results with other product lines is needed.

CHAPTER

SIX

# CONCLUSION

Verification of feature-oriented product lines is an ongoing research effort and many unsolved problems remain. Particularly, software model checking offers the possibility to find feature interactions automatically in real code. In conjunction with feature-oriented SPLs the possibly huge number of variants that need to be accounted for makes the verification problem even harder than model checking single applications.

Both approaches presented in this thesis can be used to detect feature interactions: All possible variants can be checked in sequence or the product line could be variability-encoded and checked in one single run of a model checker. The proposed automata language and the scenario modeling language can be used with both approaches presented here. As seen in section 5.3 the results if variability-encoding is beneficial to verify feature-oriented product lines are inconclusive. To answer that question, more case studies will need to be performed. Also, it needs to be investigated if these results are representative for SPLs consisting of a greater number of features. Also, it might be necessary to modify the construction of variability-encoding to achieve better results. In addition, it might prove interesting to investigate hybrid approaches where only parts of an SPL are variability-encoded. This could be used to reduce the total number of checks necessary to check the product line to a feasible number while providing simpler programs to check than the variability-encoding of the whole product line. Also, if variability-encoding proves to be ineffective for model checking of feature-oriented SPLs, it might be sensible to investigate if heuristics can be found that can be used to determine an order to model check the variants that finds interactions quickly.

APPENDIX

A

# TOOL-CHAIN INSTALLATION AND USAGE

This section details, how the tool-chain is installed. It also describes how the results of the case-study can be reproduced. Necessary steps have been automated in Python. The tool-chain was developed and tested on Ubuntu Linux 10.4 and requires the following packages to be installed:

```
python build-essential python sun-java6-jdk python-pip cbmc
```

After that, additional Python packages need to be installed:

```
pip install colorama
pip install fabric
```

Then download and unpack the SPLVERIFIER tool-chain in a directory of your choice and change to that directory. The toolchain is available on the web on the support website for feature aware verification at http://fosd.de/FAV/. Alternatively, the code is available via the subversion repository at https://svn.infosun.fim.uni-passau.de/cl/project/splverifier.

You can verify the email system by issuing the following command:

```
fab check_emailsystem
```

This command performs the checks on the email system using the CBMC model checker. Without arguments `check_emailsystem` checks all generated variants and the variability encoded product line using one specification at a time.

To check only the variability-encoded product line run:

```
fab check_emailsystem:variants=0,ve=1
```

To check all variants but not the variability-encoded product line run:

```
fab check_emailsystem:variants=1,ve=0
```

The commands above use scenarios, in which bugs are present. To use the safe scenarios append `scenarios=safe` to the commands above, e.g. to check the safe scenarios using variability encoding run:

```
fab check_emailsystem:variants=0,ve=1,scenarios=safe
```

To use CPACHECKER instead of CBMC add `mc=CPAchecker` as a configuration option. The configuration file for CPACHECKER can be specified using the `mc_config` option.

# BIBLIOGRAPHY

[AK2009] Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. In *Journal of Object Technology (JOT)*, Volume 8, Number 5, pages 49-84, 2009.

[CKMR2003] Muffy Calder, Mario Kolberg, Evan Magill, and Stephan Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Volume 41, number 1, pages 115–141, 2003.

[Clar1997] Edmund M. Clarke. Model Checking. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *Lecture Notes in Computer Science*, page 54, Springer 1997.

[PSK2009] Hendrik Post and Carsten Sinz and Wolfgang Küchlin. Towards automatic software model checking of thousands of Linux modules — a case study with Avinux. In *Proceedings of Software Testing, Verification & Reliability (STVR)*, pages 155–172. John Wiley & Sons, 2009.

[LKF2002] Harry Li and Shriram Krishnamurthi and Kathi Fisler. Verifying Cross-Cutting Features as Open Systems. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering (FSE)*, pages 89–98, ACM, 2002.

[ASLK2010] Sven Apel and Wolfgang Scholz and Christian Lengauer and Christian Kästner. Detecting Dependences and Interactions in Feature-Oriented Design. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, pages 161–170, IEEE Computer Society, 2010.

[BCHJM2004] Dirk Beyer and Adam Chlipala and Thomas A. Henzinger and Ranjit Jhala and Rupak Majumdar. The BLAST Query Language for Software Verification. In *Proceedings of the Static Analysis Symposium(SAS)*, volume 3148 of *Lecture Notes in ComputerScience*, pages. 2–18. Springer, 2004.

[BRa2002] Thomas Ball and Sriram K. Rajamani. SLIC: A specification language for interface checking (of C). *Tech. Rep. MSR-TR-2001-21*, Microsoft Research, 2002.

[PoSi2008] Hendrik Post and Carsten Sinz. Configuration Lifting: Verification meets Software Configuration. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE),*pages 347-350, IEEE Computer Society 2008.

[Hall2005] Robert J. Hall. Fundamental Nonmodularity in Electronic Mail. In *Automated Software Engineering*, Volume 12, Issue 1, pages 41–79, Springer Netherlands, 2005.

[Kaes2007] Christian Kästner. CIDE: Decomposing legacy applications into features. In *Proceedings of the 11th International Software Product Line Conference (SPLC)*, Volume 2(Demonstration), pages 149–150, 2007.

[AKL2009] Sven Apel and Christian Kästner and Christian Lengauer. FEATUREHOUSE: Language-independent, automated software composition. In *Proceedings of the 31th International Conference on Software Engineering (ICSE).* IEEE Computer Society, May 2009.

[Bato2006] Don S. Batory. A tutorial on feature-oriented programming and theAHEAD Tool Suite. In *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in ComputerScience*, pages 3–35. Springer, 2006.

[TOHS1999] Peri Tarr and Harold Ossher and William Harrison and Stanley M. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st international conference on Software engineering (ICSE)*, pages 107–119, ACM 1999.

[AL2008] Sven Apel and Christian Lengauer. Superimposition: A language-independent approach to software composition. In *Proceedings of the ETAPS International Symposium on Software Composition (SC)*, volume 4954 of *Lecture Notes in Computer Science*, pages 20–35, Springer, March 2008.

[BK2008] Christel Baier and Joost-Pieter Katoen. Principles of Model Checking (Representation and Mind Series). The MIT Press, 2008.

[Hol1997] Gerard J. Holzmann. The model checker SPIN. In *IEEE Transactions on Software Engineering*, volume 23(5), pages 279–295, IEEE Computer Society, 1997.

[BHJM2007] Dirk Beyer and Thomas A. Henzinger and Ranjit Jhala and Rupak Majumdar. The software model checker BLAST: Applications to software engineering. In *International Journal on Software Tools for Technology Transfer (STTT)*, pages 505 — 525, Springer, September 2007.

[BK2009] Dirk Beyer and M. Erkan Keremoglu. CPACHECKER: A Tool for Configurable Software Verification. Technical report SFU-CS-2009-02, School of Computing Science (CMPT), Simon Fraser University (SFU), January 2009.

[BHT2007] Dirk Beyer and Thomas A. Henzinger and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proceedings of the 19th international conference on Computer aided verification (CAV07)*, pages 504–518, Springer, 2007.

[NPRW2002] George C. Necula and Scott McPeak and Shree P. Rahul and Wesley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the International Conference on Compiler Construction (CC)*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228, Springer, 2002.

[BRb2002] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 1–3. ACM, 2002.

[CKL2004] Edmund Clarke and Daniel Kroening and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[ASWB2011] Sven Apel and Hendrik Speidel and Philipp Wendler and Dirk Beyer. Feature-Aware Verification. *under submission*

[GJ2008] Michael Gong and Hans-Arno Jacobsen. AspeCt-oriented C Specification (v0.8). Working Technical Draft, Middleware Systems Research Group, January 2008.

[KHHKPG01] Gregor Kiczales and Erik Hilsdale and Jim Hugunin and Mik Kersten and Jeffrey Palm and William Griswold. An Overview of AspectJ. In *Pro-*

*ceedings of ECOOP 2001 — Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327—354. Springer, 2001.

[P2007] Terence Parr. The Definitive ANTLR Reference: Building Domain-Specific Languages (Pragmatic Programmers). *Pragmatic Bookshelf*, 2007.

APPENDIX

B

# ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich habe die Arbeit nicht in gleicher oder ähnlicher Form bei einer anderen Prüfungsbehörde vorgelegt.

Böblingen, den 22. Januar 2011 Hendrik Speidel