Universität Passau
Fakultät für Informatik und Mathematik

Master Thesis

A Case Study on Feature-Aware Verification

Stefan Boxleitner

October 11, 2011

Advised by
Dr.-Ing. Sven Apel

Boxleitner, Stefan:
A Case Study on Feature-Aware Verification
Master Thesis, Universität Passau, 2011.

Abstract

In this thesis, we report of a case study on feature-aware verification. Feature-aware verification is the formal verification of a feature-oriented software system on a feature-modular basis. Feature-modular verification means, that specifications are assigned to a particular feature and use feature-local knowledge only. That concerns the features' awareness of other features in a particular product of a software product line. Feature-local knowledge means, that a feature is only aware of features that it depends on. The verification process includes software model checking, in particular CPACHECKER is used. We evaluate the feasability of feature-aware verification by means of an e-mail system case study, which is based on Hall's specification of an email system from [Hal05]. Thereby we detect undesired feature interactions by feature-modular specifications.

Contents

List of Figures

Chapter 1

Introduction

The development of software systems is a demanding task. There are many development paradigms that try to minimize the effort of programming while maximizing maintainability.

Software product lines and the related paradigm of feature-oriented programming adopt a concept that is known in industrial engineering for a long time. In order to serve the widest possible group of customers, products are produced in many different forms. There are, for example, cars for families that provide a lot of storage capacity or cars for offroad usage that provide a four-wheel drive. Industrial engineers try to find commonalities in these products. Consider a database product line that supports large scale storage system, as well as the embedded systems of a car. Despite the wide variety of requirements, there are commonalities like data structures or a layered architecture. A great advantage of software product lines is the reusability of these commonalities. The development costs have to be paid only once. In feature-oriented software development, these commonalities are called features. A software product is a subset of all existing features in a software product line. A product of the abovementioned database product line that fits the requirements of a large scale storage system may contain features for transaction or backup support, whereas a database for embedded systems should rather be slim and fast and thus may not require backup support.

A common problem in feature-oriented software development lies in the influence that features exercise over each other. The behavioral modification of a feature by another feature is called feature interaction. Although feature interactions are not necessarily undesired, a feature interaction may alter a software system's behavior in an undesired way. The manual detection of undesired feature interactions is costly and difficult, as every valid feature combination may introduce new critical feature interactions. The automatic detection of critical feature interactions would be desireable.

In this thesis we introduce the approach of feature-aware verification. It provides a methology to verify software product lines by means of formal verification. In particular, we use so-called observer automata to formulate specifications on a feature-modular basis. That is, the specifications are created by means of feature-local knowledge only. Feature-locality is a concept that concerns the features' awareness of each other in a feature configuration. Feature-local knowledge restricts features to be aware of other features, they depend on.

We evaluated feature-aware verification by means of a case study that is based on Hall's specification of an e-mail system in [Hal05]. The case study comprises an e-mail system product line that contains a number of undesired feature interactions. We are able to successfully detect these feature interactions in a subset of Hall's original specification by means of feature-aware verification.

Structure

Chapter 2: Prerequisites gives an insight into the concepts behind the topic: feature-oriented software development, feature interactions and software model checking.

Chapter 3: The E-mail System describes our case study in detail. As introduction to the e-mail system topic, it summarizes Hall's work from [Hal05]. Then it describes the architecture and implementation of our system, structured by the particular features. We give an overview of the feature interactions, we tried to detect by our approch and introduce our scenario concept.

Chapter 4: Feature-Aware Verification introduces our approach on feature-aware verification. It explains the roles of feature-modular specifications and local knowledge. Then the concept of observer automata and the verification process in our e-mail system is described. Finally the chapter lists up our specifications and their assignement to particular features.

Chapter 5: Results and Conclusion summarizes the thesis and gives information to further research that is related to ours.

Chapter 2

Prerequisites

In this chapter, we give an outline of concepts and tools that were used in our e-mail system case study. Section 2.1 introduces feature-oriented software development and FeatureHouse. Section 2.2 provides a definition and examples for feature interactions. Section 2.3 gives a brief introduction to software model checking.

2.1 Feature-Oriented Software Development

Apel and Kästner give an overview of feature-oriented software development in [AK09]. Feature-oriented software development (FOSD) is a programming paradigm, which defines the construction, customization, and synthesis of large-scale software systems. The core concept thereby is the feature.

> “A feature is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option.” [AK09]

FOSD provides a view that arranges a software system by its features. This arrangement enables the generation of tailored software by selecting a subset of all features that fits a certain environment best. The valid subsets form a software product line (SPL).

Features are not just manifested in a software system’s observable bahavior. They describe commonalities and variabilities in the analysis, design, and implementation. FOSD is geared towards a clear mapping between a feature and its representation in every phase of the software development process:

Domain Analysis Domain analysis defines a set of features to use in the product line. It sets the scope of the domain. Developers decide wheter a feature should be included in a domain. It may be reasonable to exclude a feature because one can’t afford the means (time, money) of its implementation. The developers also collect

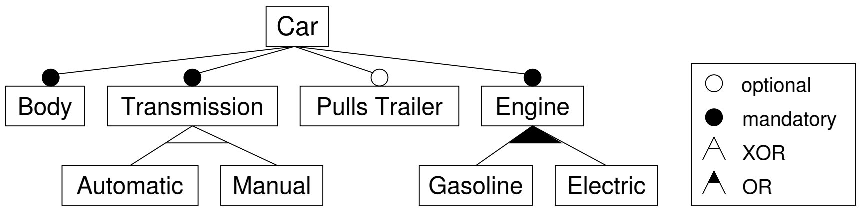

Figure 2.1: A feature model of a simple car. [AK09]

information about the releations between features. For example, a feature THEMES may depend on a feature GRAPHICAL USER INTERFACE. There are more details on the relations of features in Section 2.1.1

Domain Design and Specification Domain design and specification defines the essential structural and behavioral properties of a software product line by means of a formal or informal specification and/or modeling language. These properties are an important part of the e-mail system case study. They are discussed in detail in Chapter 4.

Domain Implementation Domain implementation typically creates a set of modular feature artifacts. Feature artifacts encapsulate the implementation per feature. It is necessary to make features explicit in code, in order to provide a clear mapping between a features and artifacts. There are languages like Jak or FeatureC++ that extend existing languages (Java and C++) with feature-based meachanisms.

Product Configuration Product configuration is the selection of a set of features. The user selects the features that fit his use case best. Then a generator traces the artifacts that are related to the feature selection and merges them into a software product.

2.1.1 Feature Models

We use the feature models that Batory describes in [Bat05]. Feature models express features and their relations among each other. A feature model reduces the set of all possible feature combinations to a subset of valid feature combinations.

It is common methology to organize features into a tree, called a feature diagram, where nodes are representing features. Nodes that have childs represent compound features. The relationships between parent and child features are expressed by the notation, given in the box to the right in Fig. 2.1. The figure also contains a model of a car product line. Every car needs a body, a transmission, and an engine. A car may have the ability to pull a trailer. The transmission can either be automatic or manual. The engine can be powered by gasoline, electric or by both, then being a hybrid electric vehicle.

```
package EmailSystem;

public class Client {
    private String name;

    public String getName() {
        return name;
    }

    public void incoming(Email email) {
        deliver(email);
    }

    //...
}
```

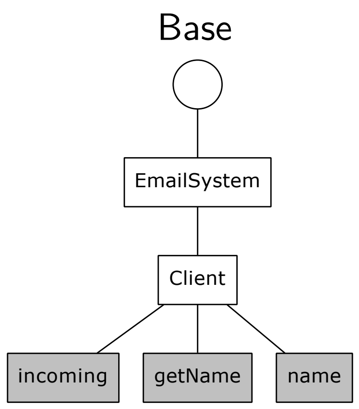

Figure 2.2: A Java feature artifact and its FST representation

Relations that don't fit in the tree structure are called cross-tree constraints. In order to express these constraints, feature models usually contain propositional logic apart from feature diagrams. For example, a car that can pull a trailer should have a lot of power, which is only achieved by a gasoline powered engine. Formal: the feature Pulls Trailer implies Gasoline.

2.1.2 FeatureHouse

The e-mail system case study is built by means of FeatureHouse. Apel, Kästner, and Lengauer introduced FeatureHouse in [AKL09]. FeatureHouse is based on the language-independent model of superimposition, presented in [AL08].

With the FSTComposer, FeatureHouse provides a tool for feature composition. Feature composition is the process of merging the feature artifacts of a valid set of features into a product.

Feature Structure Trees

FSTComposer transforms the language-specific feature artifacts into a language-independent representation, the feature structure tree (FST). Every feature is mapped to exactly one FST. FSTs can express any feature artifact with a hierarchical structure. The root of an FST represents the feature itself. The other nodes represent the artifacts' structural elements. Every node has a name and a type. For example, a Java artifact reflects this hierachical structure in its packages, classes, methods, etc. A XHTML artifact possibly consists of chapters, sections, paragraphs.

Figure 2.2 shows a Java feature artifact and its FST representation for a feature Base. The FST contains nodes for packages, classes, methods and fields but not for

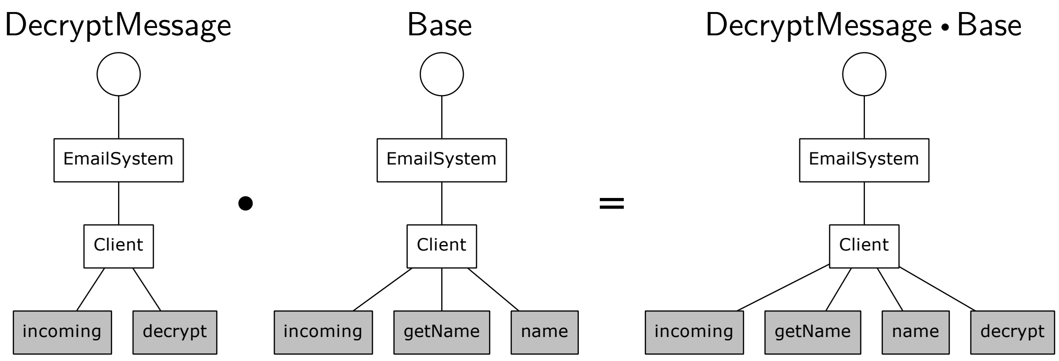

Figure 2.3: Superimposition of two FSTs

statements or expressions like an abstract syntax tree. Apel and Lengauer chose this particular grain of breakdown for Java to simplify the whole process while being still fine enough for a reasonable composition. However they point out that a coarser or finer grain would also be conceivable. The language elements that are too fine granular are included in the leaf nodes of the FST, which are called terminals (grey in the figure). The inner nodes are called nonterminals.

Superimposition

Superimposition is a binary operator that maps two FSTs R, B to another FST, denoted as $R \bullet B$. The two arguments of superimposition have a different meaning. R has stands for a refinement, an increment in functionality, B stands for an existing base. An arbitrary number of features can be composed by superimposition in a step-wise way. As FSTs are representations for feature artifacts, superimpostion is an implementation of feature composition. Apel et al. analyzed the algebraic properties of superimposition in [Ape+08]. Because of the associativity the evaluation order has no influence on the result. But the composition order is crucial, due to noncommutativity. The expression $A \bullet B \bullet C$ has in general a different meaning than $C \bullet B \bullet A$.

In FeatureHouse, the features to compose and their feature composition order are explicitly given in a text file. Every line contains the name of a feature. The composition order is determined by the line order. A file that contains

```
C
A
B
```

is interpreted as the expression B • A • C. That is, feature C is refined by feature A. The composition is then refined by feature B.

Superimposition of two FSTs starts with the composition of the root nodes. Nodes are composed according to the following set of rules:

DecryptMessage

```
public class Client {
    public void incoming(Email email) { decrypt(email); original(email); }
}
```

Base

```
public class Client {
    public void incoming(Email email) { deliver(email); }
}
```

DecryptMessage • Base

```
public class Client {
    public void incoming__wrappee__Base (Email email) { deliver(email); }
    public void incoming(Email email) {
        decrypt(email); incoming__wrappee__Base(email);
    }
}
```

Figure 2.4: Java method overriding

Nonterminals Two nonterminals r, b are composed into a node c by cloning r. The children of r, b are cloned and set as children of c. If r has a child c_r and b a child c_b, so that c_r has the same name and type than c_b, then c_r, c_b have to be composed first. The result c_c is then set as child of the resulting node c. In Fig. 2.3, for example, the nodes Client in feature DecryptMessage and Client in feature Base are composed in that way.

Terminals The composition of two terminals has to be treated specially, because of the included language-specific information. In Fig. 2.3 the nodes incoming in feature Base and DecryptMessage are composed in that way. Generally speaking, there has to be a custom composition rule for every type of nonterminal.

Method overriding is a composition rule for Java methods. Composition is done by overriding the method body of the basis by the method body of the refinement. The keyword original can be used to call the original method within the refined method body. Note the analogy to Java inheritance and the usage of the keyword super. This mechanism is implemented by wrappers. If there is a call to an overridden method, the method is kept in the FST under a new name, and the original call is renamed appropriately to meet this new name.

Figure 2.4 illustrates this process. It shows the source code of a class Client that includes a method incoming in the features Base and Decrypt. The resulting class contains a method incoming that is identical to incoming in feature Decrypt, except that the call to original is replaced by a call to a so-called wrappee. The wrappee contains the original method body of feature Base.

The terminal composition of similar structures in other languages, like functions in C can also be done be method overriding. Thus, the actual number of composition rules for terminal composition is de facto lower, than the number of nonterminal FST node types.

2.2 Feature Interactions

Keck and Kühn gave a survey on feature interactions in [KK98]. The feature interaction problem has its roots in the telecomminications systems of the 80s. The meaning of the term feature in telecomminications systems is slightly different from the concept that is depicted in Section 2.1. Bellcore, for example, defined feature as a “unit of one or more telecommunications or telecommunications management based capabilities a network provides to a user”. In telecomminications systems a feature rather resembles a service. In [Cal+03] the terms feature and service are used synonymical.

Zave provides an FAQ sheet about feature interactions on her website [Zav99]. A feature interaction occurs, when one or more features modifiy another feature, concerning its behavior in the overall system. That is, a feature behaves different in combination with one or more other features, than without them. The difficulty lies in the detection of potential feature interactions and in their classification as desired or undesired. The following examples come from [Zav99].

Examples In telephony, there are busy situations when someone is on the phone with someone else. There are features that handle busy situations, such as voice mail or forward. A priority system regulates the behavior of the overall system at an incoming call when the connection is busy. For instance, a call is forwarded, since forwarding has a higher priority than voice mail. Then the expected behavior of voice mail is modified by the feature forwarding, as it stays idle. Nevertheless, this behavior of the overall system is desired in that way, the occuring feature interaction is desired.

In another example, there are the participants Bob and Carol. Bob forwards all calls to Carol. Carol has a feature *Do not disturb* activated, which prevents others from disturbing her. A third party calls Bob, who forwards the call to Carol. Carol’s *Do not disturb* isn’t applied to forwarded calls, so she is in fact disturbed. Thus, the behavior of *Do not disturb* is modified by forward. In this case the overall system’s behavior is unwanted, this feature interaction is undesired.

The last example contains the users Alice, Bob and Carol. Alice is forwarding to Bob, Bob is forwarding to Carol. What is the desired behavior, when Alice is called? Should the call be forwarded to Carol? Considering Alice is absent, Bob is Alice’s secretary, currently using the phone of Carol, then the desired behavior would be to forward the call to Carol. If Bob is on vacation, Alice is temporarily using his office, and Carol is the secretary of Bob, the desired behavior would be to not forward the call. Only calls directly to Bob should be forwarded to his secretary then.

Feature interactions can't always be clearly classified as desired or undesired. There are authors that don't distinguish between desired and undesired feature interactions at all.

2.3 Software Model Checking

Software model checking is a subarea of model checking that considers software systems. Model checking is an approach in the field of formal verification. In contrast to informal methods like unit tests, formal verification delivers a proof, that a system behaves as expected. Baier and Katoen give an overview of model checking in their textbook [BK08].

Modeling Model checking is based on models that are mathematically precise descriptions of a system's behavior. These models are either generated directly from the system's source code in a programming language like C or from an abstract description in a modeling language. They are generally expressed by means of finite state automata. States represent the current values of variables, transitions represent statements.

A system's specification defines it's requirements, that is, *what* the system should do. For example, there should be no deadlock scenarios, or buffer overflows. These requirements can be formulated as properties. A safety property, for example, expresses that something should never happen. A liveness property specifies that something should happen finally. These properties can be formally described by temporal logic. Temporal logic is an extension to propositional logic. It contains operators that concern the system's behavior over time.

Model Checking Model checking is the process of verifying, that a certain property is valid in every state of a system's model. In mathematical terms, this process decides, if the system's description is a model of the temporal logic formula, representing the system's properties.

The number of states that are needed to aquire a precise model of the system may exceed the amount of available computer memory. The verification is valid for a model, not for the system itself. If the model of the system is inaccurate, the assertions of the model checking process may not apply to the system.

Chapter 3

The E-mail System

Our case study is built upon an e-mail system, introduced by Hall in [Hal05]. Hall describes a product line and feature interactions by means of AT&T domain knowledge. We adopted the basic structures of Hall's e-mail system, and built our own by means of the general purpose language C.

3.1 "Fundamental Nonmodularity in Electronic Mail"

As stated in Section 2.2, the understanding of features in the telecommunications industry is slighly different to our's. As he is from the telecommunications industry, Hall's perspective on features is coined by the service concept. Hall implements the e-mail system in his own specification language P-EBF, which is included in the ISAT tool suite.

E-mail Feature Components (EFCs) Hall's e-mail system consists of modular components that encapsulate features. They are called e-mail feature components (EFCs). EFCs have a common interface with a fixed set of input and output ports for message processing.

Hall's system consists of ten EFCs:

1. AddressBook allows the user to assign one or more e-mail addresses to an alias. An e-mail message with an alias as receiver is sent to every e-mail address, that is assigned to the alias.

2. SignMessage allows the user to digitally sign messages by means of public-key cryptography.

3. EncryptMessage encrypts sensible parts of messages, that is, subject and body. This is done by means of public-key cryptography. If there is a public-key provisioned for the receiver of an e-mail, the e-mail is encrypted by it.

4. DecryptMessage is the counterpart of EncryptMessage. It decrypts a message by the use of a private key. That private key must be the dual to the key that was used to encrypt the e-mail.

5. VerifySignature is used to verify a signature that has been applied by SignMessage. Verification is conditioned by the presence of the sender's public key.

6. AutoResponder can be provisioned with an automatic response to incoming messages. This behavior is useful, for example, when a user is on vacation. The automatic response then informs everyone, who sends an email, of that circumstance.

7. ForwardMessages allows forwarding of all incoming messages to a provisioned e-mail address.

8. RemailMessage is used to send messages anonymously. When a user sends a message to the remailer, the remailer replaces several fields in the message. The actual receiver is provisioned in the e-mail body, for example, in the first line. The sender field is replaced with a pseudonym. RemailMessage holds a mapping between original senders and pseudonyms. Thus, responses to anonymous messages can be delivered to the original sender.

9. FilterMessages allows to discard messages that match one of certain patterns. In particular, these patterns are suffixes of e-mail addresses that are provided by the user.

10. MailHost holds a list of user names. Received messages are either delivered to one of the users in the list or sent further on. When the receiver of an incoming message doesn't exist in the user list, the sender of the message is notified of that circumstance.

Compound EFCs EFCs are combined to complex structures by arranging them into a network and connecting their ports. These compound EFCs form the actual participants of the e-mail system: client, post office and remailer.

Figure 3.1 shows the compound EFC post office. One can see the flow of a message that arrives at the post office, in particular at the input port outgoing. The message proceeds through the EFCs FilterMessages and MailHost. According to, whether the message is dedicated to a user of the MailHost, the message is delivered, or sent further on. The message proceeds through the EFCs FilterMessages and MailHost. According to, whether the message is dedicated to a user of the MailHost, the message is delivered, or sent further on.

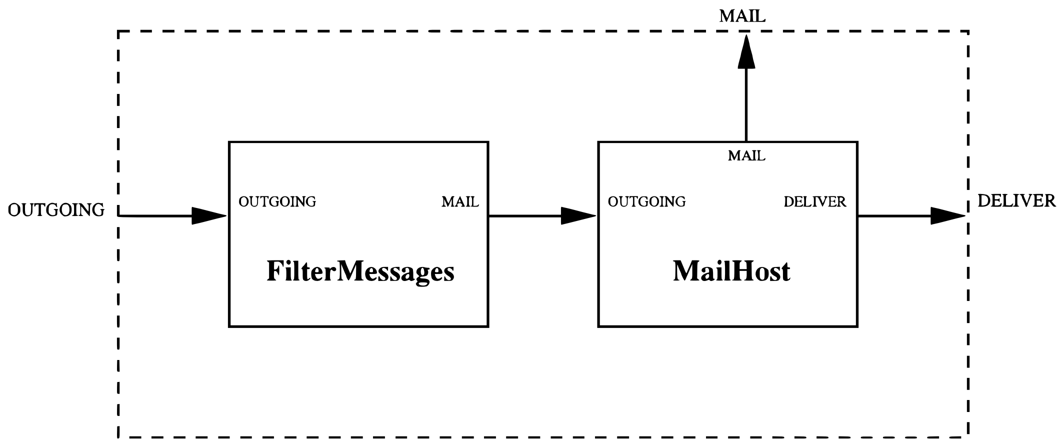

Figure 3.1: EFCs arranged to a compound EFC [Hal05]

3.2 Architecture

We built our e-mail system by means of FOSD, which is introduced in Section 2.1. Basically, we transformed EFCs into features. Although many ideas of Hall's e-mail system were inherited, there are also notable differences.

Distributed System A real e-mail system would have a distributed architecture. Each participant would potentially reside physically on a different host. There would be a separate process, with its own namespace for every participant. Since CPACHECKER does not support model checking of distributed systems, our system is technically limited to one process. A direct consequence is, that the participants in our system share one namespace.

Hall's component-based architecture allows him to configure each instance of a participant individually. In our approach all instances of a participant share the same set of features at composition time. For example, if a feature configuration contains AUTORESPONDER, every instance of client in the system is provisioned with that feature. However, most features occupy the possibility to disable its functionality at runtime. For example, if AUTORESPONDER is not provisioned with an autoresponse, the feature has no influence on the overall system. Thus, a participant de facto still can be configured different from the others at setup time of the system.

Simplifications Our system was designed for verifiability by model checking. Thus, we had to impose restrictions on the usage of certain language elements. We will amplify that in Chapter 5. In order to reduce the overall effort, we decided to constrain our system to one of the three participants: the client. Thus, the features that introduce or refine the other participants, i.e. REMAILMESSAGE, MAILHOST, and FILTERMESSAGES don't occur in our system.

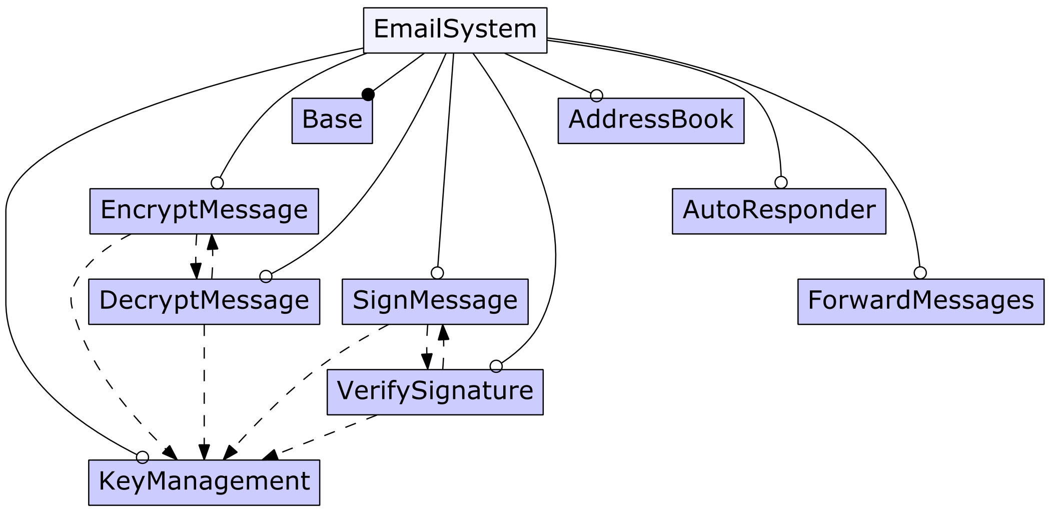

Figure 3.2: Feature Model of our E-mail System

Additional Features Appart from Hall's EFCs, our e-mail system introduces two additional features:

1. Base contains the obligatory basic client functionality. That includes operations, such as sending or receiving messages. It introduces data structures for messages and the client.
2. KeyManagement handles storage and generation of keys for public-key cryptography. We extracted this cross-cutting concern from the features EncryptMessage, DecryptMessage, SignMessage, and VerifySignature and avoid code scattering and code duplication. Modularization of cross-cutting concerns is a benefit of FOSD.

Feature Model Figure 3.2 shows the feature model of our e-mail system. Apart from Base, all features are optional. The features EncryptMessage, DecryptMessage, SignMessage, and VerifySignature depend on the functionality of KeyManagement. Since the features EncryptMessage and DecryptMessage can't be used individually in a practical way, they determine each other in our model. The same applies to SignMessage and VerifySignature. These cross-tree constraints are represented as dashed edges from a feature to its dependence.

3.3 Implementation

In contrast to Hall, who used the specification language P-EBF to implement his system, we use the C language for our e-mail system. The idea is to bring our e-mail system closer to a real system by means of a general purpose language.

Base The feature Base introduces the participant client. We chose a structure as representation for it, because it allows us to model a state for the client, which is refineable by other features.

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

The initial content of the struct client is simply the client's name.

Base also introduces the model of an e-mail message. We take the latest RFC for electronic mail [Res08] as basis. An e-mail message is divided in a header and a body. The body contains the actual message text. The header contains information about sender, receiver, subject, etc. of the message and is extendable. In an actual e-mail message, there is no guarantee, that the information in the header is genuine. For example, one could omit information about the sender or provide fake information. The information that is used for delivery of the message is located in an envelope, like in a letter. This envelope is part of the SMTP protocol that is described in an RFC [Kle08], too. SMTP data, including for example the sender, also is not necessarily genuine. We decided to omit the SMTP envelope and assume, that the information about sender and receiver in the e-mail message header is genuine.

Since other features introduce new fields to the message header, we model the e-mail message as refineable struct, too.

```
struct email
{
  int id;
  struct client *from; struct client *to;
  char *subject; char *body;
};
```

In contrast to an actual e-mail message, where the information is serialized into a string, our system models the e-mail message as structured data. This simplification reduces the complexity of the system from the model checking view. The fields from and to are pointers to actual clients. The field id is important to identify e-mail messages that are mechanically created and related to another message.

If a feature, for example, duplicates an e-mail message, edits some of the fields and sends it further on, then the id should stay the same. Thus, two messages with the same id are related to each other. Since we didn't find a language mechanism in C to enforce this behavior, we kept this behavior as convention for implementing other features. There are two functions for creating e-mail messages:

```
struct email *createEmail (struct client *from,
                           struct client *to, char *subject, char *body);
```

is used to create a new e-mail message with a new id,

```
struct email *cloneEmail (struct email *msg);
```

creates a clone of an e-mail message, preserving the id.

There are four functions that model the input and output ports of Hall's EFCs for the participant client.

```
void
incoming (struct client *client, struct email *msg)
{
  deliver (client, msg);
}

void
outgoing (struct client *client, struct email *msg)
{
  mail (client, msg);
}
```

All of them have an pointer to a client and a pointer to a message as parameters. The parameter client represents the client, on which the function is invoked. The parameter msg is the e-mail message that is to be processed. The input ports ports are incoming for arriving messages, and outgoing for messages that are to be sent. The output ports are deliver for passing the message to the user, and mail for passing a message to the network. In Base, the messages from the input ports are directly passed to the output ports. That is, there is no further processing of these messages.

The function for sending user-created messages is:

```
void
sendEmail (struct client *sender,
           struct client *receiver, char *subject, char *body)
{
  struct email *email = createEmail (sender, receiver, subject, body);
  outgoing (sender, email);
}
```

At first, an e-mail message is created, then there is a call to the function outgoing in the context of the sender. As stated above, the message is directly passed to the function mail:

```
void
mail (struct client *client, struct email *msg)
{
  incoming (msg->to, msg);
}
```

Then there is a call to the funtion incoming in the context of the receiver of the message. As claimed above, incoming directly passes the message on to deliver, which presents the message to the user. That is, the message is printed to the command line.

Figure 3.3 gives an exemplary overview of the control flow, when sending an e-mail message. A client Rjh sends an e-mail message to the client Bob. The messages

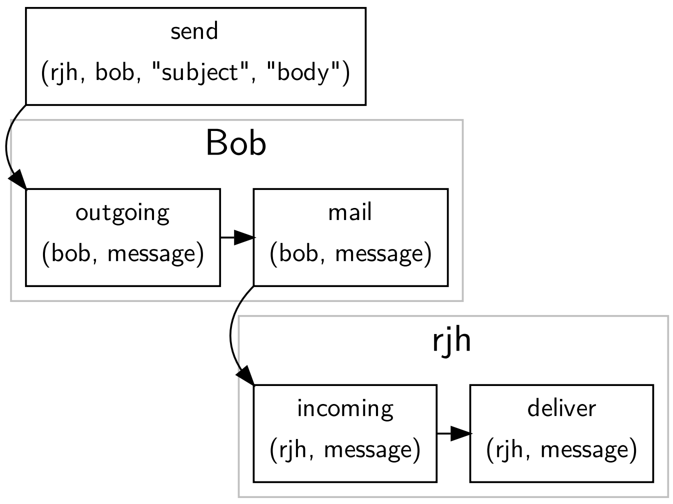

Figure 3.3: Control flow on sending an e-mail message

passes the context of Bob with its functions outgoing and mail, and arrives then at the context of Rjh, passing the functions incoming and deliver.

ForwardMessages FORWARDMESSAGES forwards all incoming messages to a client, which can be specified. Therefore, this feature refines the struct client by a field forwardReceiver, which is a pointer to a client.

```
struct client
{
  struct client *forwardReceiver;
};
```

If there is no forwardReceiver specified, that is, forwardReceiver is null, forwarding of messages is disabled and FORWARDMESSAGES doesn't affect the e-mail system's behavior.

If a forwardReceiver is provided, the function forward clones the incoming e-mail message, and replaces sender and receiver fields in the clone. As stated above, it is crucial for the verification of the system, that the message id of the clone is identical to the original message. Then the clone of the message is passed to outgoing.

```
void
forward (struct client *client, struct email *msg)
{
  if (!client->forwardReceiver)
    return;
  struct email *clone = cloneEmail (msg);
  clone->to = client->forwardReceiver;
  clone->from = client;
  outgoing (client, clone);
}
```

The call to forward takes place before the rest of original function incoming. This is implemented by means of the keyword original.

```
void
incoming (struct client *client, struct email *msg)
{
  forward (client, msg);
  original (client, msg);
}
```

The refined function incoming first calls forward and then processes the message further by its original functionality.

Consider the following example that involves the three clients Bob, Rjh, and Chuck: Rjh has forwarding to Chuck activated. Bob sends a message to Rjh. The message is forwarded to Chuck. The command line output of the program is:

```
1 deliver at chuck: id=0, from=rjh, to=chuck, subject=subject, body=body, isReadable=1
2 deliver at rjh: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=1
```

The processing of the forwarded message happens within the control flow of the original message's incoming mail processing. Thus, the forwarded message is delivered to Chuck (line 1) before the original message is finally delivered to Rjh.

AutoResponder The feature AutoResponder enables a client send an automatic response to the sender of every incoming message. Therefore the struct client is extended by a field autoResponse of type char * that stores the response. As in ForwardMessages, the functionality can be disabled by setting this field to NULL. The function autoRespond is called from a refined function incoming, like the function forward in ForwardMessages.

```
void
autoRespond (struct client *client, struct email *msg)
{
  if (!client->autoResponse || !isReadable(msg))
    return;
  struct email *response = cloneEmail (msg);
  response->from = client;
  response->to = msg->from;
  response->subject = concat ("auto␣response␣to␣", msg->subject);
  response->body = client->autoResponse;
  outgoing (client, response);
}
```

Since the response message contains the subject line of the original message, the two messages are related. Thus the automatic response message is created as clone from the original incoming message.

AddressBook The feature AddressBook provisions the client with the possibility to send e-mail messages to previously defined address book entries, also called aliases. Therefore, the struct client is refined by introducing a field addressBook. It points to a data structure that stores the aliases. We use an existing implementation of a singly linked list [1] for that purpose.

The address book entries map an identifier, represented as an integer, to a list of clients. For example, there may be an address book entry 5, which maps to Bob and Chuck.

```
struct addressBookEntry
{
  int alias;
  NODE *address;
};
```

AddressBook introduces a separate function to send e-mail messages to aliases.

```
void sendEmailAlias (struct client *sender, int alias,
                     char *subject, char *body);
```

The function looks up the given alias in the address book. Then it creates an e-mail message for the first corresponding client and passes the message to the function outgoing. After that, a clone of that message is sent outgoing for every additional client that is included in the alias. Thus, all outgoing messages from a call to sendEmailAlias have the same id.

KeyManagement The feature KeyManagement introduces a basic infrastructure for public-key cryptography. Public-key cryptography depends on key pairs that consist of a private key and a public key. The private key is kept secret at the owner, while the public key is distributed among all clients that want to participate in public-key cryptography with the owner of the corresponding private key. Thus, every client needs to store his own private key and the public keys of other clients.

KeyManagement therefore refines the struct client.

```
struct client
{
  NODE *keyring;
  int privateKey;
};
```

It introduces a field privateKey to store the private key of the client. Keys in our system are represented by numbers, in particular an integer value, instead of a string. Integers are easier to handle by model checkers, because of their finite value range.

The field keyring is a list of keyring entries. Keyring entries are modeled as a struct that contains a pointer to the owner of the key and the public key itself.

```
struct keyringEntry
{
  struct client *owner;
  int publicKey;
};
```

[1] `http://en.literateprograms.org/Singly_linked_list_(C)`

The generation of a key pair is done by the function generateKeyPair.

```
void
generateKeyPair (struct client *client, int seed)
{
  client->privateKey = seed;
}
```

Our system absracts from the complex mechanisms that are used to generate a real key pair from a random seed. We simply set the private key to the given seed itself. The corresponding public key is identical to the private key. Note, that this is only simplyfication that has nothing to do with symmetric cryptography.

When a client changes his private key and doesn't notify a corrensponding client about this change, the corresponding client has an invalid public key. KeyManagement therefore introduces a function that checks whether a public key and private key form a valid key pair.

```
int isKeyPairValid (int publicKey, int privateKey);
```

EncryptMessage EncryptMessage enables the client to encrypt outgoing messages by means of public-key cryptography. In real electronic mail, this functionality can be achieved, for example, by Pretty Good Privacy (PGP) [Cal+07] or Secure and Multipurpose Internet Mail Extension (S/MIME) [RT10]. The process of enrypting a message is done by means of the receiver's public key. Encrypted messages can't be processed by other features, before they are decrypted.

In our e-mail system, the function outgoing is refined analogous to the refinement of incoming in ForwardMessages:

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

Thus, every message that is sent outgoing is processed by the function encrypt.

```
void
encrypt (struct client *client, struct email *msg)
{
  NODE *foundKeyringEntry =
    list_find (client->keyring, findKeyringEntry, msg->to);
  if (foundKeyringEntry)
    {
      msg->encryptionKey =
        ((struct keyringEntry *) foundKeyringEntry->data)->publicKey;
      msg->isEncrypted = 1;
    }
}
```

The function encrypt looks for a keyring entry that contains the public key of the message's receiver. If there is such an entry, the message is encrypted by means of the found key. If no entry is found, the message is not encrypted.

The process of encrypting a message is simplified to setting a flag. Therefore, the struct email is refined by introducing a field isEncrypted.

In S/MIME, the decryption of an encrypted message would fail, if a wrong key was used for decryption. In order to model this behavior, the key that was used for encryption is also encapsulated in the struct email.

DecryptMessage The feature DECRYPTMESSAGE tries to decrypt incoming messages that are encrypted. The process of decrypting a message is done by means of the receiver's private key. The function decrypt is integrated in the message processing by a refinement of incoming.

```
void
decrypt (struct client *client, struct email *msg)
{
  if (msg->isEncrypted == 1 && msg->encryptionKey == client->privateKey)
    {
      msg->encryptionKey = 0;
      msg->isEncrypted = 0;
    }
}
```

As described above, the message can only be decrypted, if the decryption key is valid, that is, identical to the encryption key. Decryption is done by unsetting the isEncrypted flag.

SignMessage SIGNMESSAGE adds a signature to e-mail messages. Digital signatures are typically created from an encrypted hash value of a message. Unlike the encryption of messages, the encryption of the hash value is done by means of the sender's private key. PGP and S/MIME, as mentioned above, provide, besides from the encryption of messages, also support for creating and verifying digital signatures of messages.

Similar to ENCRYPTMESSAGE, SIGNMESSAGE reduces the signature in our system to a flag, which is introduced in the struct email.

```
struct email
{
  int isSigned;
  int signKey;
};
```

The function sign is called for all outgoing messages.

```
void
sign (struct client *client, struct email *msg)
{
  if (!client->privateKey)
```

```
    return;
  msg->signKey = client->privateKey;
  msg->isSigned = 1;
}
```

As long as the client provides a private key, outgoing messages get a signature. The verification of a signature can fail, similar to the decryption of a message. For example, when the receiver of the message has a wrong public key. To model this behavior, the key that was is used to sign the e-mail message is included in the message.

VerifySignature The feature VERIFYSIGNATURE enables the client to verify the digital signatures of SIGNMESSAGE. The verification of digital signatures requires the sender's public key.

All incoming messages are processed by the function verify.

```
void
verify (struct client *client, struct email *msg)
{
  if (!msg->isSigned)
    return;
  NODE *foundKeyringEntry = list_find (client->keyring, findKeyringEntry,
                                        msg->from);
  if (foundKeyringEntry
      &&
      isKeyPairValid (((struct keyringEntry *)
                        foundKeyringEntry->data)->publicKey, msg->signKey))
    {
      msg->isSignatureVerified = 1;
    }
}
```

Only messages that have a signature, can be verified. The function verify looks for a keyring entry of the message's sender. If the sender's public key is available, there is a check for usability of this key for verifying the signature. The key that was used to sign the message and the found public key of the sender have to be compatible, that is, identical in our system.

3.3.1 Control Flow and Composition Order

Feature selection, as explained in Section 2.1, is not sufficient to create a valid product in Hall's component based architecture. The selected EFCs need to be arranged and connected by means of their predefined interfaces into a e-mail message-passing network in a further configuration step.

Figure 3.4 shows the layout and the message flow inside the compound EFC client. Incoming e-mail messages are processed by the EFCs DECRYPTMESSAGE, VERIFYSIGNATURE, AUTORESPONDER, FORWARDMESSAGES and are then delivered. Outgoing e-mail messages are processed by the EFCs ADDRESSBOOK, SIGNMESSAGE, ENCRYPTMESSAGE and then sent on.

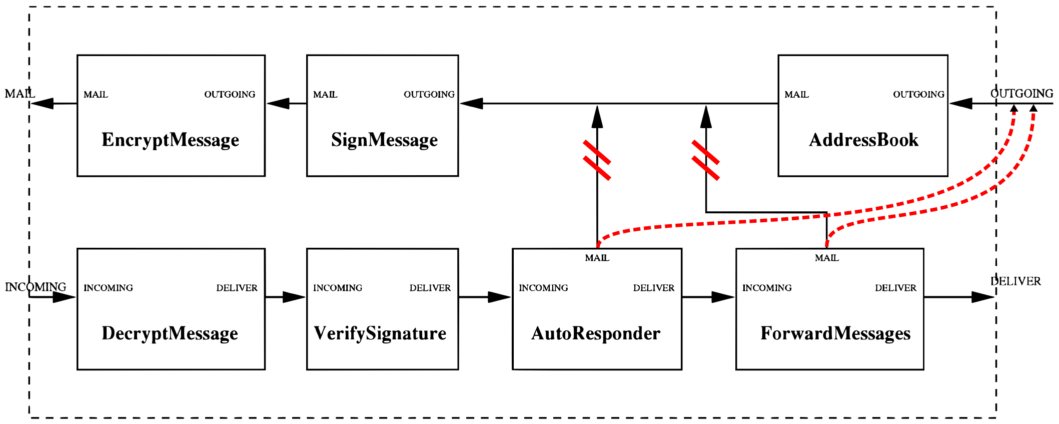

Figure 3.4: The compound EFC client [Hal05]

In order to arrange and connect the EFCs correctly, knowledge about the semantics of EFCs is necessary. DecryptMessage, for example, has to be placed first in the line of incoming message processing in the client, because encrypted messages have to be decrypted before they can be processed by other EFCs. There may be EFCs, where the placement is not necessarily distinct. AutoResponder and ForwardMessages, for example, could be switched without changing the systems behavior.

Our system models arrangement and connection of EFCs by means of feature implementation and composition order. The feature Base introduces the functions incoming and outgoing. In their unrefined form, incoming directly calls deliver and outgoing directly calls mail.

Every feature that occurs in Fig. 3.4 as EFC, integrates its additional functionality in our system by refining either incoming or outgoing. Thereby all features follow the same pattern: there is a call to an additionally introduced function before a call to original. The feature ForwardMessages, for example, introduces a function forward and refines the function incoming:

```
void
incoming (struct client *client, struct email *msg)
{
  forward (client, msg);
  original (client, msg);
}
```

We maintain the incoming and outgoing message processing order of the EFCs by imposing a constraint on the composition order of features: if an EFC A is placed after another EFC B in a processing chain, then the corresponding feature A has to be placed before the corresponding feature B in the composition order. Note, that this is only possible here, because all features follow the same pattern at refining either incoming or outgoing.

As notation for *feature* A *is placed before feature* B *in the composition order*,

```
Base
KeyManagement
EncryptMessage
SignMessage
AddressBook
ForwardMessages
AutoResponder
VerifySignature
DecryptMessage
```

```
Base
KeyManagement
ForwardMessages
AutoResponder
VerifySignature
DecryptMessage
EncryptMessage
SignMessage
AddressBook
```

```
Base
AddressBook
KeyManagement
ForwardMessages
AutoResponder
VerifySignature
DecryptMessage
EncryptMessage
SignMessage
```

Figure 3.5: Examples of valid composition orders for a full feature configuration

we write A < B. The incoming mail processing determines the following set of constraints for the composition order:

$$\begin{aligned} \textsc{ForwardMessages} &< \textsc{AutoResponder} \\ \textsc{AutoResponder} &< \textsc{VerifySignature} \\ \textsc{VerifySignature} &< \textsc{DecryptMessage} \end{aligned}$$

The outgoing mail processing determines another constraint:

$$\textsc{EncryptMessage} < \textsc{SignMessage}$$

The feature AddressBook is omitted here because its functionality is not part of the outgoing mail processing.

Since Base is a mandatory feature, all other features may be syntactically dependent on it, for example by refining the functions incoming of outgoing. Thus we impose another constraint: for every feature A that is different from Base applies:

$$\forall \text{A} : \textsc{Base} < \text{A}$$

These constraints form an order among the features in our system, which determines correctness of composition orders. Since the order is only partial, there may be multiple valid composition orders for a certain feature selection. Consider an e-mail system that contains all available features. There are many valid composition orders for this configuration that fulfill the abovementioned constraints. Figure 3.5 lists up three of them.

We demonstrate the effect of composition order and our refinement pattern on a smaller feature configuration, that contains only a subset of all features: There is a client that contains the features Base, AutoResponder and ForwardMessages. Following the abovementioned constraints, we have to compose our set of features in the following order:

```
Base
ForwardMessages
AutoResponder
```

```
void
incoming___wrappee___Base (struct client *client,
                           struct email *msg)
{
  deliver (client, msg);
}

void
incoming___wrappee___Forward (struct client *client,
                              struct email *msg)
{
  forward (client, msg);
  incoming___wrappee___Base (client, msg);
}

void
incoming (struct client *client, struct email *msg)
{
  autoRespond (client, msg);
  incoming___wrappee___Forward (client, msg);
}
```

Figure 3.6: Excerpt of a client configuration

Figure 3.6 shows the code that is generated by FEATUREHOUSE based on this feature selection and composition order. We list only the part that concerns the call sequence. When we omit the wrappees, the call sequence upon calling incoming is autoRespond, forward, deliver, which matches the incoming message processing of Hall's system.

In addition the linear processing chains, Fig. 3.4 contains two cross connections. The output port mail of the EFC AUTORESPONDER and FORWARDMESSAGES is connected to the input port outgoing of SIGNMESSAGE. That is, the EFC ADDRESSBOOK is skipped for outgoing messages of these EFCs.

Making the manifestations of a function in a particular feature callable for other features is difficult in FEATUREHOUSE. The introduction of a hook function outgoingSignMessage that is called in the outgoing refinement of SIGNMESSAGE makes the outgoing manifestation of SIGNMESSAGE callable for other features. However, the features that use this hook, fall into syntactic dependence to SIGNMESSAGE.

We decided to ommit the hook, altough a call to outgoing in the implementation of the features AUTORESPONDER and FORWARDMESSAGES enters the outgoing mail processing chain at different position, than in Hall's original specification. Figure 3.4 highlights these differences in the e-mail message processing by means of red color. Fortunately, the functionality of the feature ADDRESSBOOK is not part of the outgoing mail processing in our system. Thus, this deviation in design has no effect on the system's behavior.

3.4 Feature Interactions in the E-mail System

3.4.1 Scenarios and Variability

In our case study, we try to reproduce the interactions that Hall lists up in [Hal05]. For every interaction, he gives an example scenario that causes the corresponding undesired behavior. The interaction EncryptMessage vs. AutoResponder, for example, is given as follows:

> "Bob sends an encrypted message to Rjh who decrypts it, and who has his autoresponder provisioned. (rjh has no encryption key for Bob provisioned.) The autoresponse message includes the (now decrypted) subject line of the original message, so information is leaked because the autoresponse is transmitted as plain text over the network." [Hal05]

In order to reproduce these interactions, we created a scenario basis in the context of Hall's descriptions of feature interactions. This scenario basis contains commonalities among all scenarios. It is determined by participants and occurring events.

Participants Hall mentiones the clients *rjh*, *Bob*, and an unnamed third party, which we named *Chuck*. Our scenario basis contains a structure for each of them.

```
struct client *bob;
struct client *rjh;
struct client *chuck;
```

Events Every scenario comprises a sequence of input events to our system. We model these events as parameter-less functions. The event *Bob sends a message to Rjh*, for example, is implemented as:

```
void
bobToRjh ()
{
  char *subject = "subject";
  char *body = "body";
  sendEmail (bob, rjh, subject, body);
}
```

Subject and body of the e-mail message are filled with a placeholder value. Then the message is sent by sendEmail.

Every event can be associated with a particular feature. The event bobToRjh, for example, corresponds to the feature Base, as its implementation is completely based on functions that Base introduces. Thus, we decided to let features refine the scenario basis by introducing or refining events. Figure 3.7 illustrates this matter exemplarily. Vertically, it shows the feature Base and the feature KeyManagement, which is representative for features that model an EFC. Along the horizontal axis, there are the software artifacts client.h and test.h, representants of architecture

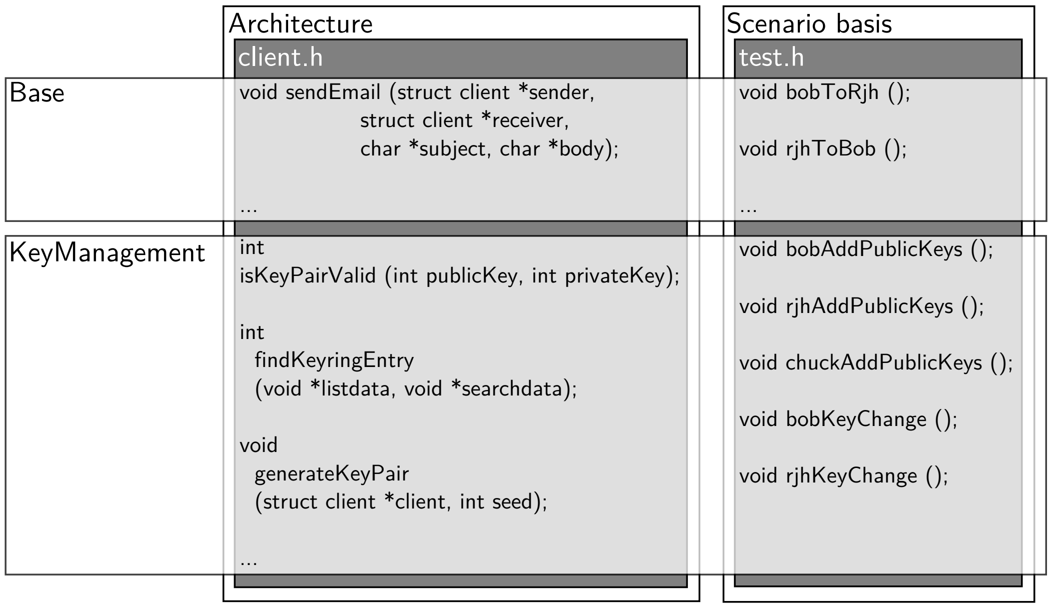
Architecture
client.h
Scenario basis
test.h
Base
void sendEmail (struct client *sender, struct client *receiver, char *subject, char *body);
...
void bobToRjh ();
void rjhToBob ();
...
KeyManagement
int isKeyPairValid (int publicKey, int privateKey);
int findKeyringEntry (void *listdata, void *searchdata);
void generateKeyPair (struct client *client, int seed);
...
void bobAddPublicKeys ();
void rjhAddPublicKeys ();
void chuckAddPublicKeys ();
void bobKeyChange ();
void rjhKeyChange ();

Figure 3.7: Features determine architecture and scenario basis

and scenario basis. The intersections between features and software artifacts denote the features' impact on architecture and scenario basis. The feature KeyManagement, for example, introduces a number of functions, such as generateKeyPair to the architecture, and - based on these functions - user-input-simulating functions, such as bobKeyChange to the scenario basis.

As senarios comprise sequences of input events, our system represents scenarios as a list of calls to the abovementioned user-input-simulating functions of the scenario basis. The example scenario from above translates to the following program:

```
bobKeyAdd ()
rjhSetAutoRespond ()
bobToRjh ()
```

3.4.2 Feature Interactions

The following section gives an overview of undesired feature interactions from Hall's descriptions that we were able to reproduce in the context of our system. For every interaction we present an event sequence that triggers the undesired behavior in the context of our system. In particular, this is a short program that consists of calls to the abovementioned user-input-simulating functions.

Every interaction is illustrated by a figure that contains the participants and the involved features. In addition to that, the descriptions contain the system's command line output for the scenario. The output is generated by the feature Base, when the functions mail and deliver are called.

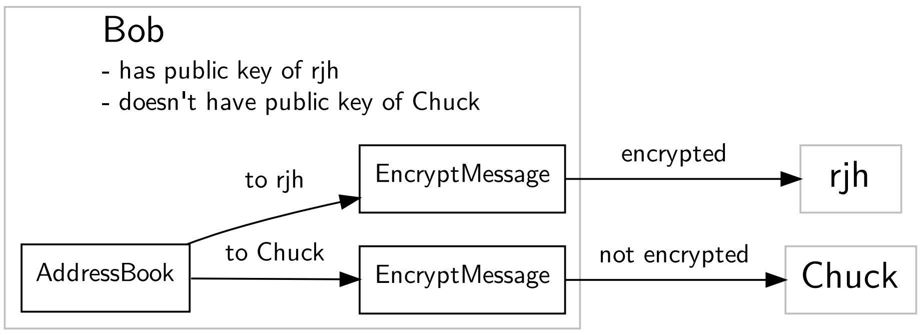

Figure 3.8: Interaction AddressBook vs. EncryptMessage

AddressBook vs. EncryptMessage

```
bobAddPublicKeys ();
```

Bob has an address book entry that includes Rjh and Chuck.

```
bobSetAddressBook ();
```

Bob knows Rjh's public key, but not Chuck's.

```
bobToAlias ();
```

Illustration: Fig. 3.8. Bob sends an e-mail message to the address book entry that includes Rjh and Chuck (bobToAlias). The feature AddressBook generates a separate message for every receiver, that is, Rjh and Chuck. Since both messages have the same content, they are related. As mentioned in Section 3.3, messages that are related, must have the same identifier. This identifier becomes more important in Chapter 4, when characteristics, as relationship of messages, need to be detected.

Both messages pass the feature Encrypt. The message to Rjh is encrypted, since Bob has Rjh's public key. The message to Chuck is not encrypted, since Bob doesn't have Chuck's key. Hence two messages that are related, are transmitted both encrypted and as plain text. Command line output:

```
mail at bob: id=0, from=bob, to=chuck, subject=subject, body=body, isReadable=1
       ,isEncrypted=0, encryptionKey=0
mail at bob: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=0
       ,isEncrypted=1, encryptionKey=2
```

One can see, there are two messages sent by client Bob with the same identifier. One of them is encrypted (isEncrypted=1), the other is not (isEncrypted=0). This is a violation of the security goal of the feature Encrypt.

SignMessage vs. VerifySignature

```
rjhAddPublicKeys ();
```

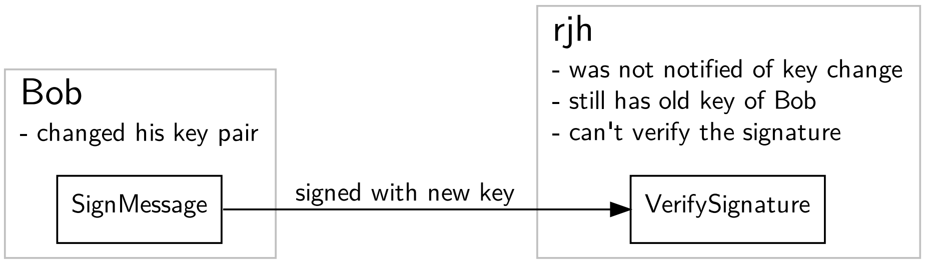
Bob
- changed his key pair
SignMessage
signed with new key
rjh
- was not notified of key change
- still has old key of Bob
- can't verify the signature
VerifySignature

Figure 3.9: Interaction SignMessage vs. VerifySignature

Rjh knows Bob's public key.

```
2 bobKeyChange ();
```

Bob changes his key pair. Thus, the old public key is invalid from now on. However, Rjh is not notified of the key change.

```
3 bobToRjh();
```

Illustration: Fig. 3.9. Bob sends a message to Rjh. Bob's SignMessage signes the message with the new key. Rjh's VerifySignature can't verify the signature of the message, because Rjh's keyring entry for Bob is outdated and invalid. Command line output:

```
mail at bob: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=1 ,isSigned=1,
      signingKey=4 ,isSignatureVerified=0
deliver at rjh: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=1 ,isSigned=1,
      signingKey=4 ,isSignatureVerified=0
```

One can see, that the message is signed by Bob, but is delivered to Rjh without a verification (isSignatureVerified=0).

The problem is, that Rjh can't decide whether the message was altered during transmission, or Bob changed his key. According to Hall, this behavior is undesired.

SingMessage vs. ForwardMessages

```
1 chuckAddPublicKeys ();
```

Chuck knows Bob's public key. Rjh doesn't know Bob's public key.

```
2 rjhEnableForwarding ();
```

Rjh sets Chuck as forward target. That is, all incoming messages are forwarded to Chuck.

```
3 bobToRjh ();
```

Illustration: Fig. 3.10. Bob sends a message to Rjh. The message is signed by Bob before trasmission to Rjh. Rjh can't verify the signature, since he doesn't

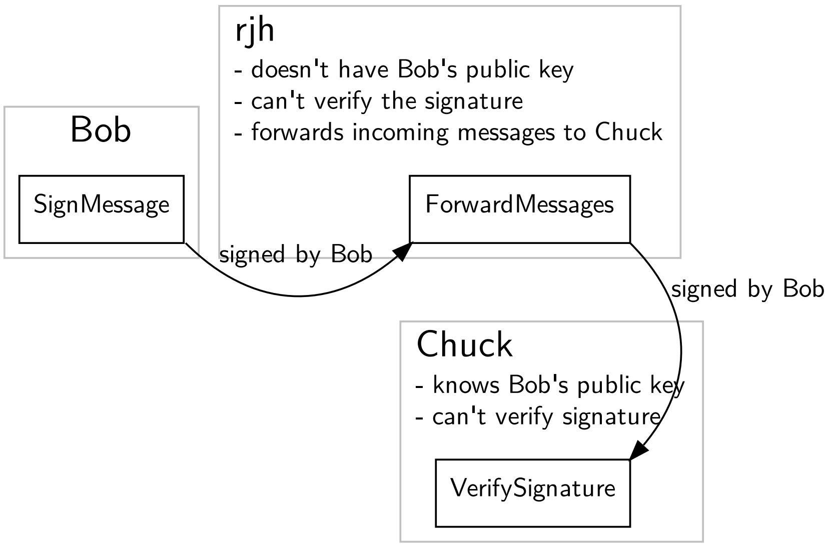

Figure 3.10: Interaction SignMessage vs. ForwardMessages

have Bob's public key provisioned. Then Rjh forwards the message to Chuck. The message's signature remains unaffected. Since Rjh, the sender of the forwarded message, doesn't match the original creator of the signature, Chuck's verification of the message fails. The undesired behavior at this point is, that Chuck can't verify the signature, although he knows the public key of the signature's original creator.

Command line output:

```
1 mail at bob: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=1 ,isSigned=1,
      signingKey=1 ,isSignatureVerified=0
2 mail at rjh: id=0, from=rjh, to=chuck, subject=subject, body=body, isReadable=1 ,isSigned=1,
      signingKey=2 ,isSignatureVerified=0
3 deliver at chuck: id=0, from=rjh, to=chuck, subject=subject, body=body, isReadable=1
      ,isSigned=1, signingKey=2 ,isSignatureVerified=0
4 deliver at rjh: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=1 ,isSigned=1,
      signingKey=1 ,isSignatureVerified=0
```

The forwarded message (lines 2-3) arrives at its destination before the original message (lines 1,4), since ForwardMessages executes the complete delivery of the forwarded message, before it is passed on. The output shows, that the signature is present all the time, but is never verified.

EncryptMessage vs. DecryptMessage

```
1 bobAddPublicKeys ();
```

Bob knows the public key of Rjh.

```
2 rjhKeyChange ();
```

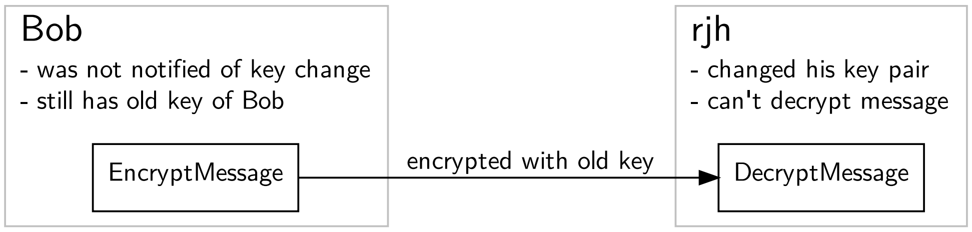

Figure 3.11: Interaction ENCRYPTMESSAGE vs. DECRYPTMESSAGE

Rjh changes his key pair. Similar to Section 3.4.2, Bob is not aware of that change and thinks, his keyring entries are up to date.

```
3  bobToRjh ();
```

Illustration: Fig. 3.11. Bob sends a message to Rjh. As Bob falsely thinks, he has a valid public key for Rjh, the message is encrypted before transmission to Rjh. Rjh can't decrypt the message, because he changed his key pair in the meantime. According to Hall this behavior of not being able to decrypt the message is undesired.

Command line output:

```
1  mail at bob: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=0
       ,isEncrypted=1, encryptionKey=2
2  deliver at rjh: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=0
       ,isEncrypted=1, encryptionKey=2
```

One can see, that the transmitted message is encrypted (line 1: isEncrypted=1). The message can't be decrypted before delivery and remains unreadable (line 2: isReadable=0).

EncryptMessage vs. VerifySignature

```
1  bobAddPublicKeys ();
2  rjhAddPublicKeys ();
3  rjhKeyChange ();
```

The procedure here is almost the same as in interaction ENCRYPTMESSAGE vs. DECRYPTMESSAGE. The difference is, that message signing is also involved here. In order to verify messages that are signed by Bob, Rjh knows Bob's public key (line 2).

```
4  bobToRjh ();
```

Illustration: Fig. 3.12. When Bob sends a message to Rjh, the decryption fails, as explained above. In this particular interaction, the verification of the message therefore also fails, because the message is still encrypted.

The undesired behavior in this case is Rjh's inability to verify the message, although he has a valid keyring entry for Bob.

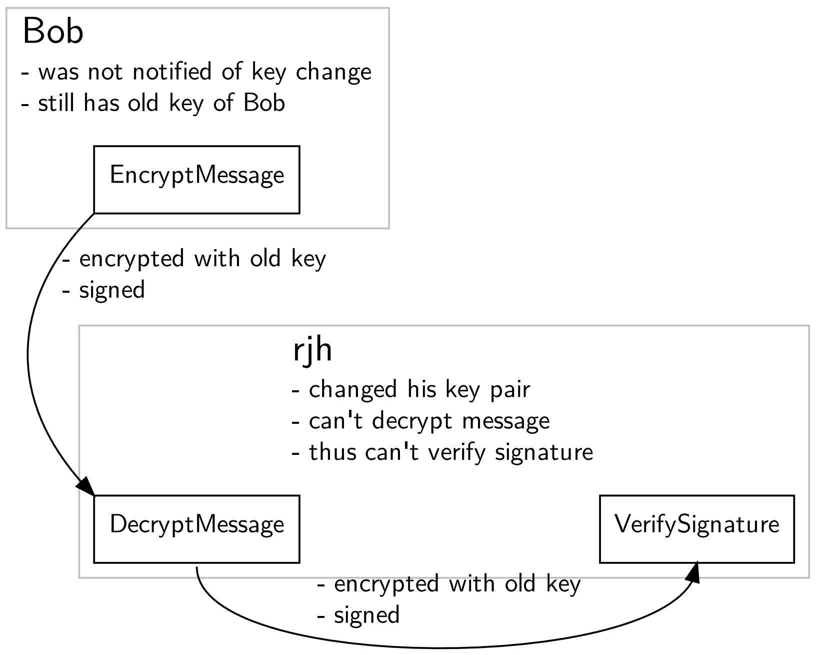

Figure 3.12: Interaction EncryptMessage vs. VerifySignature

Command line output:

```
mail at bob: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=0
      ,isEncrypted=1, encryptionKey=2 ,isSigned=1, signingKey=1 ,isSignatureVerified=0
deliver at rjh: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=0
      ,isEncrypted=1, encryptionKey=2 ,isSigned=1, signingKey=1 ,isSignatureVerified=0
```

The message is signed, when transmitted to Rjh, but the signature still is not verified when delivered to Rjh (isSignatureVerified=0).

EncryptMessage vs. AutoResponder

```
1 bobAddPublicKeys ();
```

Bob knows the public key of Rjh, Rjh doesn't know the public key of Bob.

```
2 rjhSetAutoRespond();
```

Rjh's auto response functionality is enabled.

```
3 bobToRjh ();
```

Illustration: Fig. 3.13. Bob's EncryptMessage encrypts the message before its transmission to Rjh. Rjh's DecryptMessage succesfully decrypts the message and passes it on. AutoResponder creates an automatic response to the message which contains the subject of the original message. Therefore, the automatic

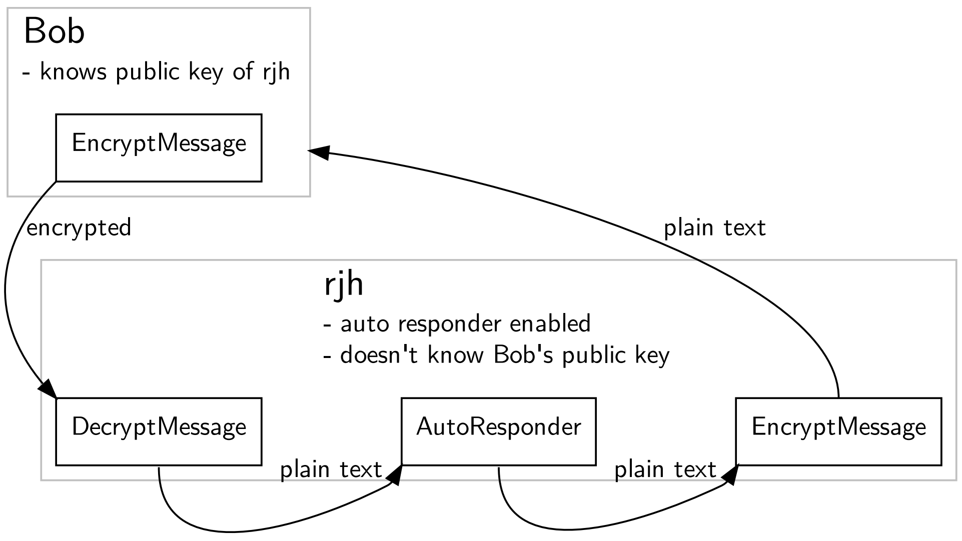

Figure 3.13: Interaction EncryptMessage vs. AutoResponder

response is related to the original message and has the same identifier. The automatic response is passed to Rjh's EncryptMessage. It is not encrypted before transmission, since there is no keyring entry for Bob.

Thus, the automatic response, which contains the subject of the original message, is transmitted as plain text. Again, the security goal of EncryptMessage is violated.

Command line output:

```
1 mail at bob: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=0
      ,isEncrypted=1, encryptionKey=2
2 mail at rjh: id=0, from=rjh, to=bob, subject=auto response to subject, body=I am not available.,
      isReadable=1 ,isEncrypted=0, encryptionKey=0
3 deliver at bob: id=0, from=rjh, to=bob, subject=auto response to subject, body=I am not
      available., isReadable=1 ,isEncrypted=0, encryptionKey=0
4 deliver at rjh: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=1
      ,isEncrypted=0, encryptionKey=0
```

Original message (line 1) and automatic response (line 2) have the same identifier. The autoresponse contains the original subject ("subject") in its own subject line and is not encrypted (isEncrypted=0).

EncryptMessage vs. ForwardMessages

```
1 bobAddPublicKeys ();
```

Bob knows Rjh's public key.

```
2 rjhEnableForwarding ();
```

Rjh forwards all incoming messages to Chuck.

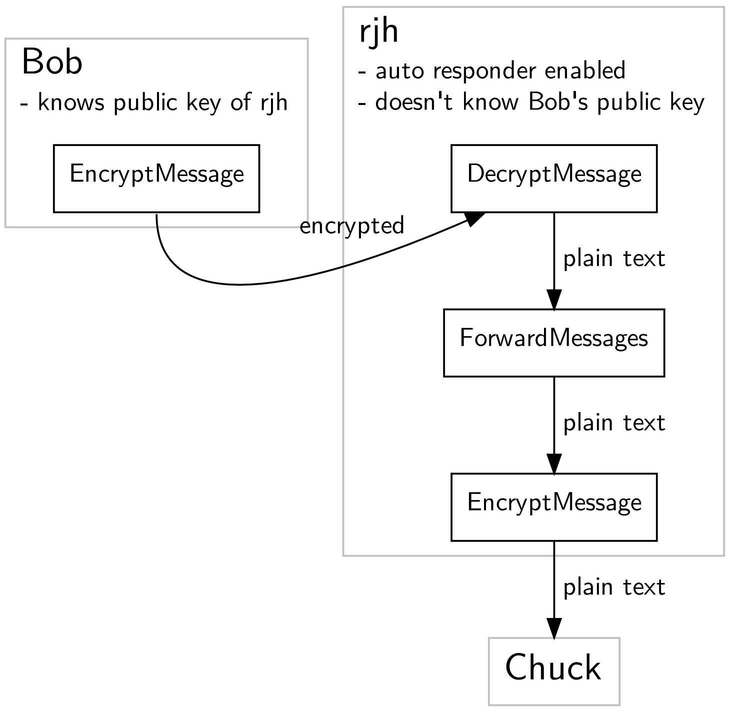

Figure 3.14: Interaction EncryptMessage vs. ForwardMessages

```
bobToRjh ();
```

Illustration: Fig. 3.14. Bob sends a message to Rjh. Bob's EncryptMessage encrypts the message before transmission to Rjh.

Rjh succesfully decrypts the message and passes the message on to ForwardMessages. ForwardMessages generates a second message. This generated message has the same subject and body, as the original message. Consequently the generated message is related to the original one. Both messages have the same identifier. The generated message is passed to Rjh's EncryptMessage, which doesn't encrypt the message due to the missing keyring entry for Chuck.

The forwarded message is transmitted as plain text to Chuck. Again, the security goal of EncryptMessage is violated, since the same content is transmitted both encrypted and as plain text over the network.

Command line output:

```
mail at bob: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=0
      ,isEncrypted=1, encryptionKey=2
mail at rjh: id=0, from=rjh, to=chuck, subject=subject, body=body, isReadable=1
      ,isEncrypted=0, encryptionKey=0
deliver at chuck: id=0, from=rjh, to=chuck, subject=subject, body=body, isReadable=1
      ,isEncrypted=0, encryptionKey=0
deliver at rjh: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=1
      ,isEncrypted=0, encryptionKey=0
```

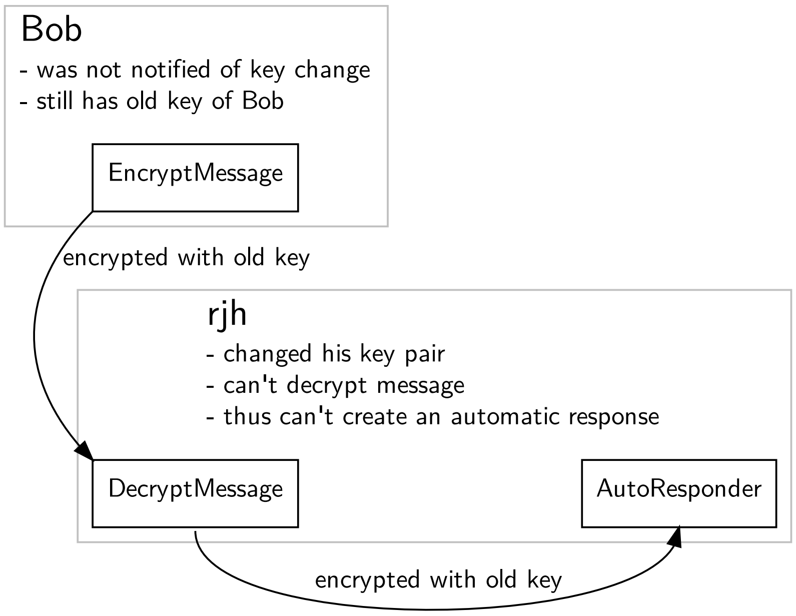

Figure 3.15: Interaction DECRYPTMESSAGE vs. AUTORESPONDER

The identifiers of original message and forwarded message are identical. The original message is encrypted (line 1: isEncrypted=1), the forwarded message is not (line 2: isEncrypted=0).

DecryptMessage vs. AutoResponder

```
bobAddPublicKeys ();
rjhKeyChange ();
rjhSetAutoRespond();
```

This interaction is strongly related to the interactions ENCRYPTMESSAGE vs. DECRYPTMESSAGE and ENCRYPTMESSAGE vs. VERIFYSIGNATURE. Again there is an unannounced key change of Rjh, so Bob's keyring entry for Rjh is invalidated without his knowledge. Additionally, Rjh has his automatic responder enabled.

```
bobToRjh ();
```

Illustration: Fig. 3.15. Bob sends a message to Rjh. When the message arrives at Rjh's DECRYPTMESSAGE, it can't be decrypted, because it was encrypted which with an invalid key. The AUTORESPONDER can't generate an automatic response, since the subject line of the still encrypted message is not readable. The failure of DECRYPTMESSAGE causes the AUTORESPONDER to fail, too.

Command line output:

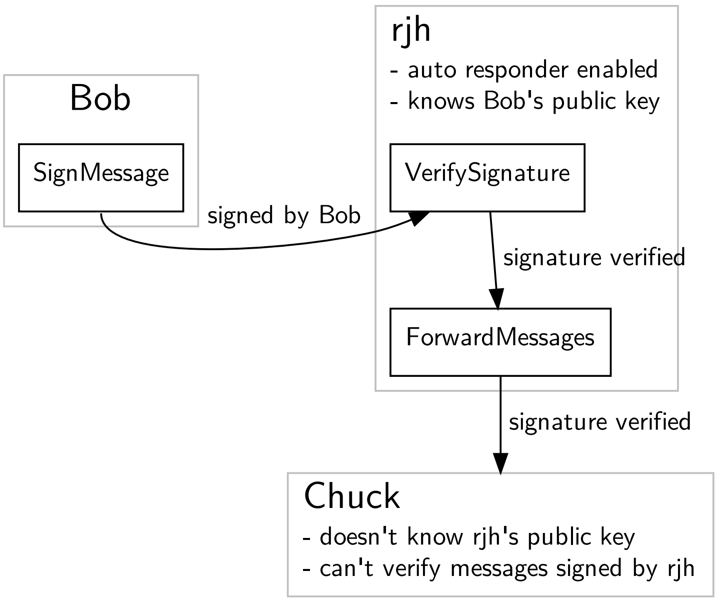

Figure 3.16: Interaction VerifySignature vs. ForwardMessages

```
mail at bob: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=0
        ,isEncrypted=1, encryptionKey=2
deliver at rjh: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=0
        ,isEncrypted=1, encryptionKey=2
```

One can see, that there is no automatic response generated or sent. Only the original message appears.

VerifySignature vs. ForwardMessages

```
rjhAddPublicKeys ();
```

Rjh knows Bob’s public key.

```
rjhEnableForwarding ();
```

Rjh forwards all incoming messages to Chuck.

```
bobToRjh ();
```

Illustration: Fig. 3.16. Bob sends a message to Rjh. Bob signs the message and transmits it to Rjh.

As Rjh knows the public key for Bob, he successfully verifies the signature of the message. The successful verification is indicated as flag to the message. In order to sustain the message identifier, ForwardMessages clones the forward message from the original message. This cloned message also contains the indication flag for successful verification of the message.

When the forward message is transmitted to Chuck, he falsely believes, that the signature of the forwarded message was sucesfully verified. But in fact the signature was not verified and the message could have been altered on transit from Rjh to Chuck.

Command line output:

```
1 mail at bob: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=1 ,isSigned=1,
        signingKey=1 ,isSignatureVerified=0
2 mail at rjh: id=0, from=rjh, to=chuck, subject=subject, body=body, isReadable=1 ,isSigned=1,
        signingKey=2 ,isSignatureVerified=1
3 deliver at chuck: id=0, from=rjh, to=chuck, subject=subject, body=body, isReadable=1
        ,isSigned=1, signingKey=2 ,isSignatureVerified=1
4 deliver at rjh: id=0, from=bob, to=rjh, subject=subject, body=body, isReadable=1 ,isSigned=1,
        signingKey=1 ,isSignatureVerified=1
```

The forwarded message (lines 2-3) contains the flag that indicates the successful verification of a message (isSignatureVerified=1) at time of transmission (line 2) and of delivery (line 3).

Chapter 4

Feature-Aware Verification

In this chapter, we introduce the approach of feature-aware verification. Given a software product line and a specification, we invstigate possible specification violations by means of software model checking. In particular, we focus on one class of specification violations: feature interactions, as described in Section 2.2. However our approach is not restricted to feature interactions.

4.1 Feature Modular Specification

As stated in Section 2.1, a software product line consists of features. In our approach we assign specifications explicitly to features. Thus, we gain a degree of modularization, where properties like feature traceability are preserved among the feature artifacts of the specification.

Local Knowledge Properties in the specification of a certain feature only have a restricted knowledge of the overall system. The knowledge is restricted to implementation details of the feature itself, and the presence of other features that the feature is dependent on, according to the feature model. If one regards the feature model in Fig. 2.1 again, local knowledge of feature PULLS TRAILER includes, for example, the assumption, that the feature BODY exists, but not what other features exist in a feature configuration.

Althought the specifications are restricted to local knowledge, their violation can indicate a feature interaction, because the verification is done after feature composition. Considering, for example, a text processing system. The feature BASE reads a text and outputs it. The feature TRANSLATE EN translates an arbitrary text into English, TRANSLATE DE translates a text into German. Both features depend on BASE and are developed in isolation. The specification of TRANSLATE EN dictates, that the outputted text has to be English. Note, that no assumptions about the actual feature configuration or other features have to be made. The specification of Translate En only refers to the invocaton of a function `output`, which is located in the feature Base.

Considering a feature configuration, that contains both Translate En and Translate De. Any text that is input is translated into English first, then translated into German and finally output.

The specification of feature Translate En is violated by Translate De, although the specification was created without knowledge of Translate De.

4.2 Observer Automata

The specifications in our approach are expressed by observer automata. Beyer et al. introduced them in [Bey+04]. Observer automata have their own set of states, which is independent of the observed program. They are executed in parallel to the observed program without altering it. The edges of observer automata consist of syntactic patterns. State transistions in the observed program may trigger a state transition in the observer automaton. Such a transition in the observer is made, when there is an outgoing edge from the current state that matches the current transistion in the observed program. Typically, an automaton contains states that indicate a violation of the specified property.

Figure 4.1 illustrates this behavior. The automaton model to the left represents the text processor from the example above. The observer automaton on the right contains the specification of Translate En. The state `v` indicates a specification violation. One can see, that the automaton model of the text processor contains auxiliary code that actually doesn't belong to the text processor, but to the specification of the feature Translate En. In particular, that is the check sequence bewteen state 5 and 7/8. This additional functionality has no influence on the behavior of the text processor but is necessary for the specification of feature Translate En. Our approach contains a methology that weaves this auxiliary code into the system to verify. This concept is explained briefly in Section 4.3 and in detail in Speidel's work [Spe11].

These automata have similarities to the pointcut and advice model in aspect-oriented programming, for example AspectJ [Kic+01]. Pointcuts are also based on syntactic pattern matching. Aspects have their own states and advices can be instrumented to make state transitions.

4.3 Verification Process

For our case study, we use a particular specification language to express the specifications in our system. Speidel describes this specification language for observer automata in [Spe11]. The specification language is intended to verify programs in the C programming language. However, this limitation to C is not by concept but by the choice of CPAChecker as model checking tool.

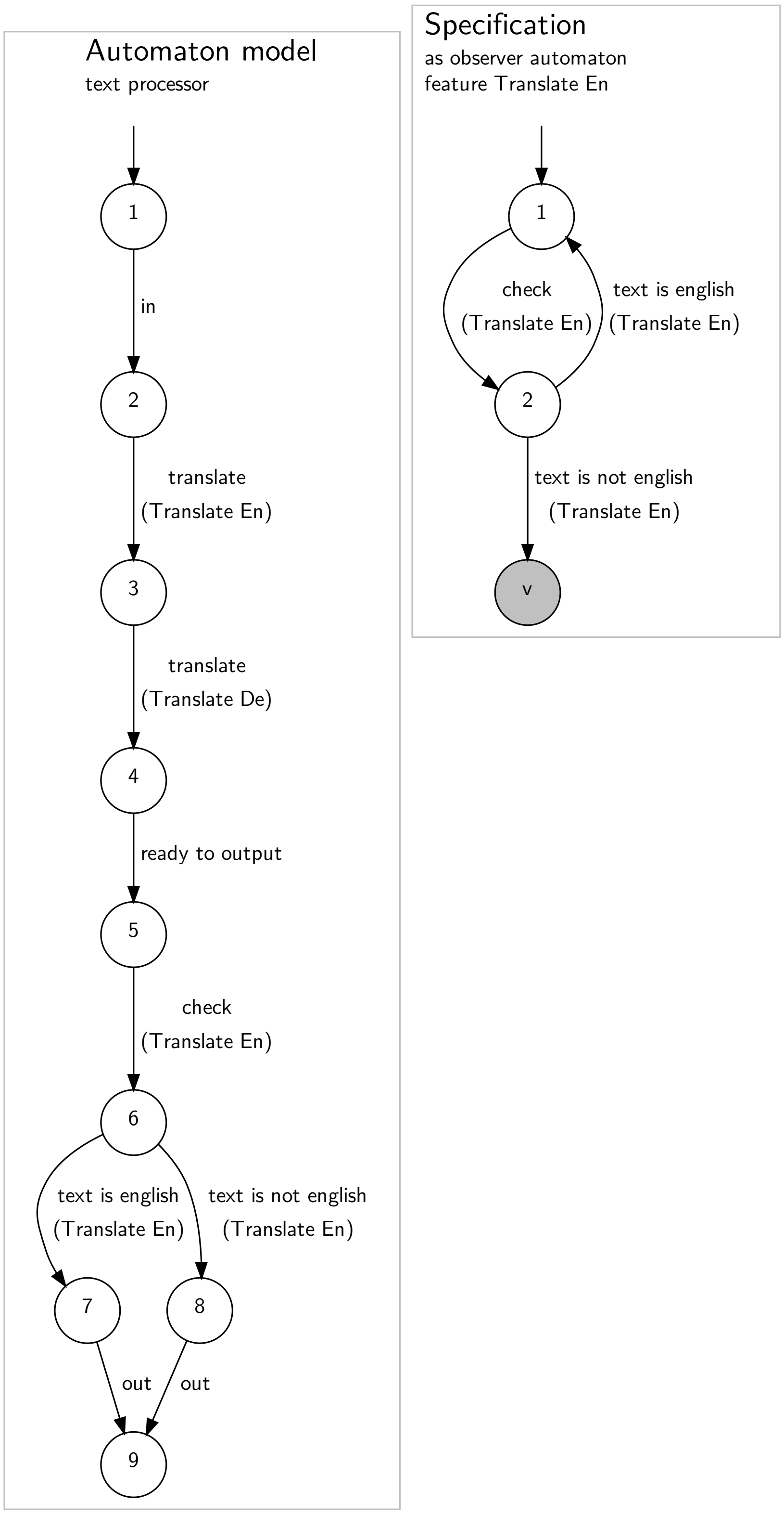

Figure 4.1: An observer automaton, monitoring a program

In Speidel's approach, the observer automata are woven into the program code. The model checker then verifies the reachability of states that indicate a specification violation. An interaction is deemed to be detected, when a specification is violated in the respective scenario that reproduces the interaction.

4.4 Specifications in the E-mail System

To proof the feasibility of the approach of feature-aware verification, we created specifications that allow us to detect the feature interactions, we listed up in Section 3.4.2.

Every specification is assigned to a particular feature. In order to maintain feature modularity, the specifications are restricted to local knowledge only.

These specifications were initially created as observer automata in a graphical representation. Speidel translated these automata into his specification language. For a better understanding, we present these specifications in their initial graphical form.

Cosidering the example automaton from Fig. 4.1, observer automata here are given in a modified form. In order to maintain clarity, they are simplified and already contain parts of Speidel's specification language.

In addition to pattern matches, the state transistions are conditioned by compound statements, which include propositional logic.

As the automata are woven into the observed program, it is possible to specify arbitrary declarations and statements for a state transition. Speidel calls this concept action block.

The propositional expressions and the action blocks are evaluated in a context that is generated by the respective pattern. A pattern match on a function, for example, generates an environment, where the function's parameters are accessible by the propositional expressions and the action block.

4.4.1 EncryptMessage and DecryptMessage

As EncryptMessage and DecryptMessage depend on each other in the feature model, they are treated together.

Prohibit information leaking

Observer automaton in Fig. 4.2. This specification prohibits information leaking, by forbidding the transmission of two related messages both encrypted and as plain text.

The motivation for the concept of related messages should become clearer now. Two messages are related, if one message is derived from the other automatically and carries sensitive information of the original message. Sensitive information can be the subject line, the body or parts of them.

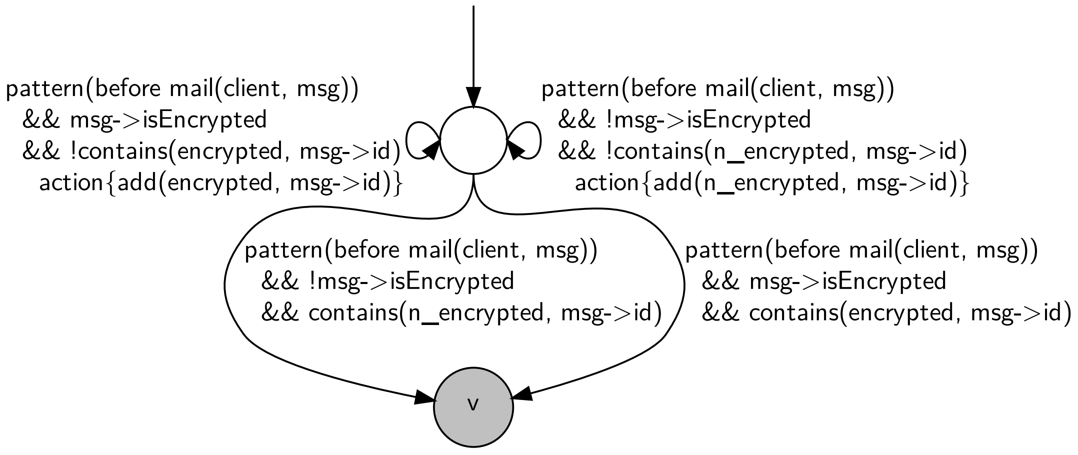
pattern(before mail(client, msg))
&& msg->isEncrypted
&& !contains(encrypted, msg->id)
action{add(encrypted, msg->id)}
pattern(before mail(client, msg))
&& !msg->isEncrypted
&& !contains(n_encrypted, msg->id)
action{add(n_encrypted, msg->id)}
pattern(before mail(client, msg))
&& !msg->isEncrypted
&& contains(n_encrypted, msg->id)
pattern(before mail(client, msg))
&& msg->isEncrypted
&& contains(encrypted, msg->id)
v

Figure 4.2: Prohibit information leaking in ENCRYPTMESSAGE

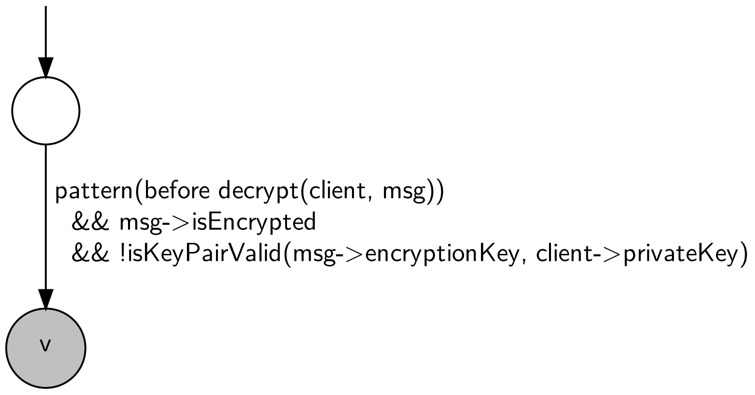
pattern(before decrypt(client, msg))
&& msg->isEncrypted
&& !isKeyPairValid(msg->encryptionKey, client->privateKey)
v

Figure 4.3: Prohibit inconsistent key state in DECRYPTMESSAGE

Concerning the automaton, all transmitted messages are monitored. When a message is sent encrypted, its identifier is stored in a data structure for encrypted message identifiers. When a message is sent as plain text, its identifier is stored in a data structure for unencrypted message identifiers.

When a message is sent unencrypted and its identifier is already stored in the data structure for encrypted message identifiers, then messages of the same relation class were transmitted first encrypted and then as plain text. When a message is sent encrypted and its identifier is already stored in the data structure for unencrypted message identifiers, then messages of the same relation class were transmitted first as plain text and then encrypted. In both cases a violation is detected.

Prohibit inconsistent key state

Observer automaton in Fig. 4.3. This specification prohibits inconsistent key states during the decryption of messages. The automaton therefore monitors the calls to

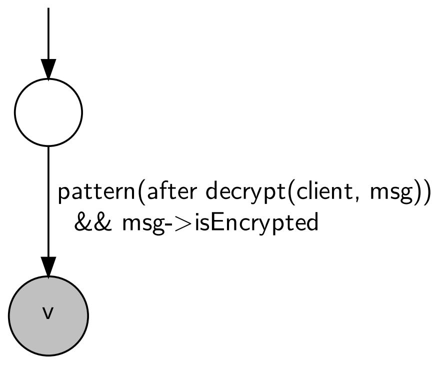

Figure 4.4: Prohibit decrypt failure

decrypt and verifies, wheter the key that was used to encrypt the message and the key that is used to decrypt the message, form a valid key pair. If that is not the case, there is a transistion to the violation state.

Prohibit decrypt failure

Observer automaton in Fig. 4.4. The local knowledge in feature DecryptMessage includes the fact that messages, whose decryption fails, can't be processed further. Thus, this specification forbids this state. When a message is still encrypted after the execution of decrypt, the specification is violated.

4.4.2 SignMessage and VerifySignature

SignMessage and VerifySignature also depend on each other in the feature model. They are treated together, too.

Prohibit inconsistent key state

Observer automaton in Fig. 4.5. This specification prohibits inconsistent key states during the verification of messages. The automaton therefore monitors the calls to verify. When verify is called for a signed message and the local keyring contains a key for the sender of the message, there is a check whether the key that was used to sign the message and the keyring entry form a valid key pair [1]. When they don't, there is an inconsistent key state.

Enforce signer and sender consistency

Observer automaton in Fig. 4.6. Sender and signer of a message have to be the same entity, otherwise VerifySignature can't determine the matching keyring

[1]The last two clauses in the state transistion are strongly simplified for reasons of readability. The actual expressions contain nested function calls with two and more parameters and a cast.

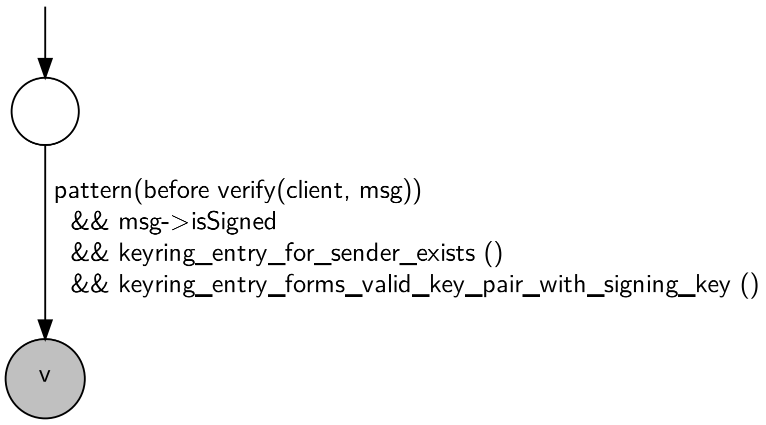
pattern(before verify(client, msg))
&& msg->isSigned
&& keyring_entry_for_sender_exists ()
&& keyring_entry_forms_valid_key_pair_with_signing_key ()
v

Figure 4.5: Prohibit inconsistent key state in VERIFYSIGNATURE

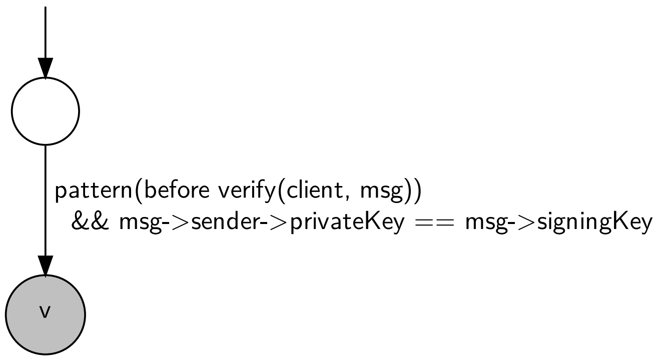
pattern(before verify(client, msg))
&& msg->sender->privateKey == msg->signingKey
v

Figure 4.6: Enforce signer/sender consistency in VERIFYSIGNATURE

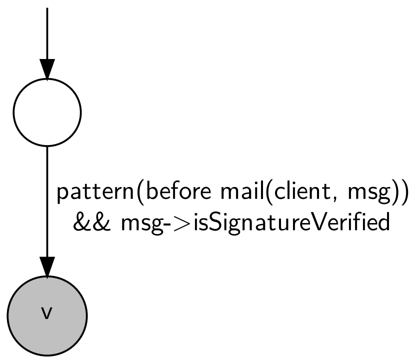
pattern(before mail(client, msg))
&& msg->isSignatureVerified
v

Figure 4.7: Prohibit sending of verified signatures in VerifySignature

entry to verify a signature. When the private key of the message's sender and the key that was used to create the signature are identical, this property is guaranteed.

Prohibit sending of verified signatures

Observer automaton in Fig. 4.7. A verified signature is only valid within a participant in the e-mail system. Thus, the transmission of messages that have a verified signature flag is not allowed.

Chapter 5

Results and Conclusion

We built an e-mail system that is based on Hall's work in [Hal05] as part of our case study. Hall implemented his system by means of a formal verification language that is part of his Interactive Specification Acquisition Tools (ISAT) tool suite, our system is built on the general purpose programming language C.

Technical Problems Technical hurdles prevented us from realizing the complete system. CPAChecker, the tool we primary used in our approach, is an academic project and still under development. It does not support all elements of the C language. In particular, dynamic memory allocation isn't fully integrated yet. This problem came to light, while implementing the features AddressBook and KeyManagement, which use list structures for data handling. Although we were able to work around this issue, we decided to ommit post office and remailer, because of this issue. As both participants would have included a user account management, an extensive use of dynamic data structures in the implementation would have been likely. Thus, our system includes only one of the originally three participants.

E-mail Feature Components (EFCs) and Features Every EFC is associated to a particular participant. As our system contains the participant client only, we ommited an implementation of the EFCs that concern post office and remailer. Expressed in numbers, our system contains seven of ten EFCs from Hall's system. Since two of the three ommited EFCs contain only basic functionality of the respective participant - RemailMessage for the remailer and MailHost for the post office - we implemented almost all features that actually increment the functionality of a participant. Table 5.1 gives details on the differences between our system and Hall's.

Feature Interactions The abovementioned restriction of our system to the client as the only participant strongly limits the number of interactions that are repro-

Table 5.1: Comparison between our system and Hall's system at a glance

	Our system	Hall's system
Implementation	General purpose language (C)	Formal specification language (ISAT tool suite)
Participants	Client	Client, post office, remailer
Features / EFCs	AddressBook, SignMessage, EncryptMessage, DecryptMessage, VerifySignature, AutoResponder, ForwardMessages, Base, KeyManagement	AddressBook, SignMessage, EncryptMessage, DecryptMessage, VerifySignature, AutoResponder, ForwardMessages, RemailMessage, FilterMessages, MailHost

Table 5.2: List of feature interactions in our system and Hall's system

	AddressBook	SignMessage	EncryptMessage	DecryptMessage	VerifySignature	AutoResponder	ForwardMessages	RemailMessage	FilterMessages	MailHost
AddressBook	0/0	0/0	1/1	0/0	0/0	0/0	0/0	0/1	0/0	0/0
SignMessage	0/0	0/0	0/0	0/0	1/1	0/0	1/1	0/1	0/0	0/0
EncryptMessage	0/0	0/0	0/0	1/1	1/1	1/1	/1 1	0/1	0/0	0/0
DecryptMessage	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0	0/0
VerifySignature	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0
AutoResponder	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/1	0/1
ForwardMessages	0/0	0/0	0/0	0/0	0/0	0/1	0/1	0/2	0/1	0/1
RemailMessage	0/0	0/0	0/0	0/0	0/2	0/2	0/0	0/0	0/1	0/1
FilterMessages	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
MailHost	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/1	0/0

ducible at all. Particularly the participants post office and remailer are involved in 16 of overall 27 interactions that Hall found. There are two further interactions that manifest themselves in infinite message loops. We did not investigate this matter further under the aspect of the abovementioned technical problems. But with the exception of these two, our system is able to reproduce all interactions that concern the implemented subset of participants and EFCs.

Table 5.2 gives an overview of found feature interactions. The figures represent the number of feature interactions found for a pair of features. The first one stands for our system, the second one for Hall's system. For example the figures 1/1 in the first line express, that Hall found one interaction, that involves the features AddressBook and EncryptMessage and our system is able to reproduce this interaction. The table is not symmetrical over the diagonal, because Hall's naming convention for feature interactions. In his component-based view of features, an e-mail message has to pass the EFC A before the EFC B in a scenario that triggers the feature interaction A vs. B.

Specifications In order to detect feature interactions using feature-aware verification, we created specifications for particular features that have feature-local knowledge only. These specifications allow us to detect all reproducible interactions. Table 5.3 gives an overview of all specifications and the interactions that can be detected with them. One can see, that serveral specifications cover more than one interaction, because they are not necessarily suited to detect one paricular interaction. The fact, that we were able to assign all specifications to security relevant features (SignMessage and VerifySignature or EncryptMessage and DecryptMessage) is noteworthy. Our specifications seem to protect mainly security properties.

Unexploited Variability The e-mail system context offers still unexploited variability in the generation of scenarios. Our scenario basis contains only a small number of participants. A real-world e-mail system is scalable and there may be many interactions that only occur in large scale scenarios with a large number of participants. This matter could be a starting point for further research in this area.

5.1 Related Work

The last section exemplary lists up projects that are to a certain extent related to the work of this thesis.

Software Product-Line Group The Software Product-Line Group at the University of Passau does further research on safety in software product lines at this thesis' time of origin. There is another promising implementation of our e-mail system in Java, for which the Java Pathfinder [1] is used a model checking solution.

[1] `http://babelfish.arc.nasa.gov/trac/jpf`

Table 5.3: Specifications and coverage of detected interactions

	Specifications					
	EncryptMessage / DecryptMessage			SignMessage / VerifySignature		
Interactions	Prohibit information leaking	Prohibit inconsistent key state	Prohibit decrypt failure	Prohibit inconsistent key state	Enforce signer / sender consistency	Prohibit sending of verified signatures
AddressBook vs. EncryptMessage	×					
EncryptMessage vs. AutoResponder	×					
EncryptMessage vs. ForwardMessages	×					
EncryptMessage vs. DecryptMessage		×				
EncryptMessage vs. VerifySignature			×			
DecryptMessage vs. AutoResponder			×			
SignMessage vs. VerifySignature				×		
SignMessage vs. ForwardMessages					×	
VerifySignature vs. ForwardMessages						×

Being developed by the National Aeronautics and Space Administration (NASA), the Java Pathfinder seems to be a mature tool that carries out few or no restrictions to the Java code to verify. This alternative system extends ours and fully implements Hall's specifications of an e-mail system from [Hal05].

Verification at Specificaton Level There is further research done on detection of feature interactions at specification level. That is, the examined systems are formulated by means of a formal specification language in contrast to our system, which is implemented in the general purpose language C. One might argue that a general purpose language rises the level of realism. However, our system still is an approximation that abstracts from many aspects of a real e-mail system.

The work of Li, Krishnamurthi, and Fisler [LKF02; FK01] and Jouve, Gall, and Coudert [JGC05] is examplary for that.

Bibliography

[AK09] Sven Apel and Christian Kästner. “An Overview of Feature-Oriented Software Development”. In: *Journal of Object Technology* 8.5 (2009), pp. 49–84.

[AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. “FEATUREHOUSE: Language-independent, automated software composition”. In: *ICSE*. IEEE, 2009, pp. 221–231. ISBN: 978-1-4244-3452-7.

[AL08] Sven Apel and Christian Lengauer. “Superimposition: A Language-Independent Approach to Software Composition”. In: *Software Composition*. Ed. by Cesare Pautasso and Éric Tanter. Vol. 4954. Lecture Notes in Computer Science. Springer, 2008, pp. 20–35. ISBN: 978-3-540-78788-4.

[Ape+08] Sven Apel et al. “An Algebra for Features and Feature Composition”. In: *AMAST*. Ed. by José Meseguer and Grigore Rosu. Vol. 5140. Lecture Notes in Computer Science. Springer, 2008, pp. 36–50. ISBN: 978-3-540-79979-5.

[Bat05] Don S. Batory. “Feature Models, Grammars, and Propositional Formulas”. In: *SPLC*. Ed. by J. Henk Obbink and Klaus Pohl. Vol. 3714. Lecture Notes in Computer Science. Springer, 2005, pp. 7–20. ISBN: 3-540-28936-4.

[Bey+04] Dirk Beyer et al. “The Blast Query Language for Software Verification.” In: *SAS*. Ed. by Roberto Giacobazzi. Vol. 3148. Lecture Notes in Computer Science. Springer, 2004, pp. 2–18. ISBN: 3-540-22791-1.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008, pp. I–XVII, 1–975. ISBN: 978-0-262-02649-9.

[Cal+03] Muffy Calder et al. “Feature interaction: a critical review and considered forecast”. In: *Computer Networks* 41.1 (2003), pp. 115–141.

[Cal+07] J. Callas et al. *OpenPGP Message Format*. RFC 4880 (Proposed Standard). Updated by RFC 5581. Internet Engineering Task Force, Nov. 2007. URL: `http://www.ietf.org/rfc/rfc4880.txt`.

[FK01] Kathi Fisler and Shriram Krishnamurthi. "Modular verification of collaboration-based software designs". In: *ESEC / SIGSOFT FSE*. 2001, pp. 152–163.

[Hal05] Robert J. Hall. "Fundamental Nonmodularity in Electronic Mail". In: *Autom. Softw. Eng.* 12.1 (2005), pp. 41–79.

[JGC05] Helene Jouve, Pascale Le Gall, and Sophie Coudert. "An Automatic Off-Line Feature Interaction Detection Method by Static Analysis of Specifications". In: *FIW*. Ed. by Stephan Reiff-Marganiec and Mark Ryan. IOS Press, 2005, pp. 131–146. ISBN: 1-58603-524-X.

[Kic+01] Gregor Kiczales et al. "An Overview of AspectJ". In: *ECOOP*. Ed. by Jørgen Lindskov Knudsen. Vol. 2072. Lecture Notes in Computer Science. Springer, 2001, pp. 327–353. ISBN: 3-540-42206-4.

[KK98] Dirk O. Keck and Paul J. Kühn. "The Feature and Service Interaction Problem in Telecommunications Systems. A Survey". In: *IEEE Trans. Software Eng.* 24.10 (1998), pp. 779–796.

[Kle08] J. Klensin. *Simple Mail Transfer Protocol*. RFC 5321 (Draft Standard). Internet Engineering Task Force, Oct. 2008. URL: `http://www.ietf.org/rfc/rfc5321.txt`.

[LKF02] Harry C. Li, Shriram Krishnamurthi, and Kathi Fisler. "Verifying cross-cutting features as open systems". In: *SIGSOFT FSE*. 2002, pp. 89–98.

[Res08] P. Resnick. *Internet Message Format*. RFC 5322 (Draft Standard). Internet Engineering Task Force, Oct. 2008. URL: `http://www.ietf.org/rfc/rfc5322.txt`.

[RT10] B. Ramsdell and S. Turner. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification*. RFC 5751 (Proposed Standard). Internet Engineering Task Force, Jan. 2010. URL: `http://www.ietf.org/rfc/rfc5751.txt`.

[Spe11] Hendrik Speidel. "A Specification Language for Observer Automata in Feature-Oriented Verification". MA thesis. Universität Passau, 2011.

[Zav99] Pamela Zave. *FAQ Sheet on Feature Interaction*. 1999. URL: `http://www2.research.att.com/~pamela/faq.html` (visited on 03/21/2011).

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Ort, Datum *Stefan Boxleitner*