Variability Modeling of Cryptographic Components (Clafer Experience Report)

Sarah Nadi
Software Technology Group
Technische Universität Darmstadt, Germany
nadi@cs.tu-darmstadt.de

Stefan Krüger
Secure Software Engineering Group
Technische Universität Darmstadt, Germany
stefan.krueger@cased.de

ABSTRACT

Software systems need to use cryptography to protect any sensitive data they collect. However, there are various classes of cryptographic components (e.g., ciphers, digests, etc.), each suitable for a specific purpose. Additionally, each class of such components comes with various algorithms and configurations. Finding the right combination of algorithms and correct settings to use is often difficult. We believe that using variability modeling to model these algorithms, their relationships, and restrictions can help non-experts navigate this complex domain. In this paper, we report on our experience modeling cryptographic components in Clafer, a modeling language that combines feature modeling and meta-modeling. We discuss design decisions we took as well as the challenges we ran into. Our work helps expand variability modeling into new domains and sheds lights on modeling requirements that appear in practice.

1. INTRODUCTION

Variability modeling is a way to understand the commonalities and differences of products in a particular domain. Besides being a useful form of documentation and communication about a system and its offered products, it provides a basis for automated reasoning and configuration of these products. Variability modeling has been successfully applied in many domains such as the automotive domain [1], databases [2], and systems software [3]. However, different domains and applications pose different challenges to variability modeling, both conceptually and in terms of language support. Reporting these challenges and learning from them is important for future applications of variability modeling and for improving modeling languages. In this paper, we apply variability modeling to a new domain, that of cryptography, using the Clafer modeling language [4].

There are a large number of cryptography algorithms available with different configurations, uses, and trade-offs. Application programmers, who are not necessarily security experts, struggle with choosing the correct algorithm and settings. Previous research has already shown that application developers often misuse cryptographic APIs, leading to security vulnerabilities [5,6]. In our previous work, we outlined a task-based solution that helps application developers easily and securely use cryptography in their applications [7]. Our long-term goal is to provide a tool that allows users to select the high-level tasks they want to perform (e.g., encrypt file or protect password) and then generate the corresponding correct and secure code for them. In this process, developers can also configure certain requirements that may be specific to the task (e.g., the type of data they will process or memory requirements). Only algorithm combinations that match these requirements will be used in the generated code. Thus, at the heart of such a system lies a variability model that allows us to reason about different configurations. In our previous work, we identified several requirements for a modeling language (hierarchal structure, non-boolean values, references, commonality & variability, and automated reasoning) [7] and accordingly started exploring Clafer.

In this paper, we focus only on the variability modeling aspect of this solution and dive deeper into the details involved and our experience in using Clafer. We discuss different design decisions and modeling alternatives we considered during the process as well as some limitations of Clafer we ran into. Additionally, we briefly discuss alternatives to Clafer that may address the cryptography modeling requirements we found. In summary, our contributions are as follows:

- Variability modeling of a new domain (cryptography).
- A realistic application of an existing variability modeling language, Clafer.
- A report on design decisions and challenges faced during variability modeling of cryptography.
- Potential solutions and workarounds to the identified issues, where applicable.
- Discussion of alternative languages for our domain.

We believe our experience is useful for researchers and practitioners who (1) want to use Clafer or (2) want to apply variability modeling in similar domains.

2. BACKGROUND

2.1 Cryptography

Cryptography provides a means to protect sensitive data [8]. In the process, it relies on several classes of algorithms or functions such as symmetric encryption ciphers, asymmetric encryption ciphers, key derivation functions, digests (i.e., hash functions), signatures, etc. Such algorithms are usually

© 2015 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

combined to achieve a certain *task*. For example, to perform the task of encrypting data based on a given password, one must first use a key derivation function to derive a secret key that is then used by a symmetric cipher to encrypt the data. Internally, the key derivation function relies on a digest.

Various algorithms are suitable for the above task. For example, there are several encryption algorithms such as `AES` or `DES`. Within `AES`, three key sizes are supported (128, 192, or 256 bits), whereas `DES` only supports 56 bit keys. While `AES` is considered secure, there have been some cryptanalytic attacks against `DES` so it is no longer recommended. In general, for non-broken algorithms, the larger the key, the harder it is to crack the encryption. The choice depends on the level of security needed, but there are often trade-offs to consider. For example, more secure algorithms may have slower performance [9]. In addition to these considerations, some algorithms have proven attacks (e.g., the `MD5` digest [10]), but are still supported for compatibility or because they may still be used in some scenarios. All such domain knowledge and variability must be encoded in a model that allows us to reason about algorithms suited for a particular task.

2.2 Clafer

Clafer (class, feature, reference) is a lightweight modeling language [4]. It combines concepts from feature modeling, meta-modeling, and class modeling. Clafer supports many modeling tasks such as feature modeling, configuration, domain modeling, and example-driven modeling. Clafer unifies several modeling concepts such as features, instances, and attributes into a single concept called *clafer*. Note the difference in capitalization: Clafer refers to the language while clafer refers to the modeling concept used. While clafers can represent any concept, they can be either *abstract* or *concrete*. Abstract clafers are marked by an `abstract` keyword which tells the instance generator not to try to resolve variability or generate instances for that clafer. In addition to concepts from traditional and attribute-based feature modeling [11], Clafer also supports inheritance and references. The following example defines an abstract concept `Person`, which has a sub-concept `Age` of type integer.

```
abstract Person                  Bob : Person     person -> Person
  age -> integer                   [age = 40]

abstract Teen : Person           Alice : Teen
  [age >= 13 && age <= 19]         [age = 15]
```

Note that both `Person` and `Age` are clafers. `Teen` extends `Person` by adding the constraint that age must lie between 13 and 19 years. `Bob` and `Alice` are two instances of `Person` and `Teen` respectively. In other words, they extend these clafers but provide concrete values for all properties. The last clafer `person` is a reference to `Person`.

Clafer supports two backend reasoners for instance generation. The first is based on Alloy [12] and the second is based on the Choco constraint solver [13]. If we run the instance generator on the above example, we would get two instances where `person` can either be `Bob` or `Alice`. Clafer also comes with an extensive tool suite that includes the compiler and a multi-objective optimizer [14].

3. MODELING CRYPTOGRAPHIC COMPONENTS USING CLAFER

In this section, we report on our experience modeling cryptographic components in Clafer. Figure 1 shows a model with three tasks that we refer to throughout this section. Lines 1-6 and Lines 8-13 define (desugared) enumeration types for the security and performance levels of an algorithm, respectively (more details in Section 3.3). Line 15 defines an abstract clafer `Algorithm` that has a name and a description (both of type string) as well as security and performance levels defined as references to the previous enumerations. Line 21 defines an abstract clafer `Digest` that extends `Algorithm`. `Digest` adds a new clafer `outputSize` to indicate the size of the produced hash. Similarly, Line 24 defines a `KeyDerivationAlgorithm` that also extends `Algorithm` and adds some extra properties to it. Notice that the child clafer `digest` is optional as indicated by the question mark. This is because some key derivation functions, such as `pbkdf2` on Line 79, can work with several underlying digests[1] while others, such as as `scrypt` on Line 86, already use a fixed digest internally (and thus no digest needs to be specified) [15]. However, algorithms that specify a digest to use should only use accepted secure digests [16] that are not broken as indicated by Line 28. We also choose to limit the output size on Line 29 to the popular output sizes used instead of defining an allowed range and leaving the instance generator to find suitable values. On Line 30, we fix the number of iterations to 1000 as this is the accepted default.

The remaining parts of the model similarly define `Cipher` and `Task`. Starting from Line 42, we define concrete instances of the abstract clafers we previously defined. Some of the concrete clafers such as `AES` still have unresolved variability, while others such as `DES` assign concrete values to all their child clafers. Note that this model is of course not comprehensive of the cryptography domain.

We now discuss various design decisions of the model and the challenges we ran into. All our discussions are based on using Clafer v0.4.1. For the backend, we mainly use the Choco-solver back-end due to its faster performance [1]. However, we also discuss possible optimizations in this section that would allow us to use Alloy more effectively.

3.1 Multiple Product Types

A feature model typically represents a family of products. A valid instance of the model is one possible product. For example, a car feature model contains all the possible features of a car (transmission, power windows, etc.) and a model instance would be one model of the car. In the end, there is only one type of product being represented, a car. For our purposes, this product is *a task*. However, the problem is that there is no way to have a common representation for all tasks. For example, let us look at the `SymmetricEncryption` and `SecurePassword` tasks on Lines 94 and 98 respectively. While they are both cryptographic tasks, they have no common algorithms. While it can be argued that a task consists of one or more `Algorithm` clafer, designing tasks to only depend on `Algorithm` is too generic and meaningless. This is because either any combination of algorithms will be considered as (meaningless) valid instances or we have to create a large amount of constraints to ensure only meaningful algorithm combinations that represent realistic tasks. This even gets more complicated when in addition to the list of algorithms belonging to a task, there exist additional constraints that govern the combination of these algorithms (e.g., the constraint on Line 106 in the `PasswordBasedEncryptionTask`).

[1] Note that technically, key derivation functions use an HMAC with an underlying digest, but we skip the HMAC here for simplicity.

```
1  //enum Security
2  abstract Security -> integer
3  Broken: Security = 1
4  Weak: Security = 2
5  Medium: Security = 3
6  Strong: Security = 4
7
8  //enum Performance
9  abstract Performance -> integer
10 VerySlow: Performance = 1
11 Slow: Performance = 2
12 Fast: Performance = 3
13 VeryFast: Performance = 4
14
15 abstract Algorithm
16     name -> string
17     description -> string
18     security -> Security
19     performance -> Performance
20
21 abstract Digest : Algorithm
22     outputSize -> integer
23
24 abstract KeyDerivationAlgorithm :
       Algorithm
25   iterations -> integer
26   outputSize -> integer
27   digest -> Digest?
28   [digest.security.ref != Broken]
29   [outputSize = 128 || outputSize =
       192 || outputSize = 256]//fix
       popular output sizes
30   [iterations = 1000]//accepted
       default # of iterations
31
32 abstract Cipher : Algorithm
33
34 abstract SymmetricCipher : Cipher
35     keySize -> integer
36
37 abstract SymmetricBlockCipher :
       SymmetricCipher
38
39 abstract Task
40   description -> string
```

```
41 //group similar algorithms together
42 Ciphers
43   AES : SymmetricBlockCipher
44     [description = "Advanced
         Encryption Standard (AES)
         cipher"]
45     [name = "AES"]
46     [keySize = 128 || keySize = 192
         || keySize = 256]
47     [keySize = 128 => performance =
         VeryFast && security =
         Medium]
48     [keySize > 128 => performance =
         Fast && security = Strong]
49
50   DES : SymmetricBlockCipher
51       [description = "DES encryption"]
52       [name = "DES"]
53       [security = Broken]
54       [performance = VeryFast]
55       [keySize = 56]
56
57 DigestAlgorithms
58   md5: Digest
59     [description = "MD5 digest"]
60     [name = "MD5 digest"]
61     [performance = VeryFast]
62     [security = Broken]
63     [outputSize = 128]
64
65   sha_1: Digest
66     [name = "SHA-1"]
67     [description = "SHA-1 digest"]
68     [performance = VeryFast]
69     [security = Weak]
70     [outputSize = 160]
71
72   sha_256: Digest
73     [description = "SHA-256"]
74     [name = "SHA-256 digest"]
75     [outputSize = 256 ]
76     [security = Strong]
77     [performance = Slow]
```

```
78 KeyDerivationAlgorithms
79   pbkdf2 : KeyDerivationAlgorithm
80     [name = "PBKDF2"]
81     [description = "Password-Based
         Key Derivation Function 2"]
82     [performance = Slow]
83     [digest]
84     [security.ref =
         digest.security.ref]
85
86   scrypt : KeyDerivationAlgorithm
87     [name = "scrypt"]
88     [description = "Scrypt
         password-based key
         derivation"]
89     [no digest] //Already uses
         HMAC_SHA256 internally
90     [performance = VerySlow]
91     [security = Strong]
92
93 Tasks
94   SymmetricEncryption : Task
95     [description = "Encrypt data
         using a secret key"]
96     cipher -> SymmetricBlockCipher
97
98   SecurePassword : Task
99     [description = "Represent
         password in a secure way
         for storage"]
100    kda -> KeyDerivationAlgorithm
101
102  PasswordBasedEncryption : Task
103    [description = "Encrypt data
         using a given password"]
104    kda -> KeyDerivationAlgorithm
105    cipher -> SymmetricBlockCipher
106    [cipher.keySize = kda.outputSize]
```

Figure 1: Clafer model of cryptography algorithms and tasks

Based on the observations above, we believe that while we use many concepts from feature modeling, our produced model is conceptually not exactly a feature model in the traditional sense. This is of course in addition to the fact that we make use of class modeling concepts provided by Clafer such as inheritance and referencing, which drives it further from traditional feature modeling. However, in essence, we see this as a variation of variability and domain modeling.

3.2 Extent of Partial Variability Use

On a high-level design, we have two choices to model the available cryptographic algorithms. The first is to define what combinations of features is allowed and then let the instance generator generate all possible instances. Let us take the example of an encryption cipher. In its simplest form, a symmetric encryption cipher has a name representing the algorithm used and a (positive) key length, as follows:

```
abstract SymmetricBlockCipher
  name -> string
  keySize -> integer
  [keySize > 0]
```

While this correctly represents all ciphers, if we try to generate instances, we would get all key sizes from 1 until the maximum integer value allowed by the solver. Additionally, no meaningful value will be assigned to name. To make this more meaningful, we can look at all available algorithms and check the key sizes they support. For some algorithms such as AES, a pre-determined set of key sizes (128, 192, and 256 bits) are supported while in other algorithms such as the asymmetric cipher RSA, key sizes may take on any integer value between 512 and 65,536 bits. If we assume that all symmetric ciphers are of the first type, we can add the following constraint to `SymmetricBlockCipher`.

```
[keySize = 64 || keySize = 128 || keySize = 192 || ...]
```

Adding this constraint greatly restricts the number of generated instances. However, it does not solve finding the algorithm name and does not solve the fact that different algorithms may have different supported key sizes. While we can solve this by adding a constraint that specifies a set of allowed names and then add specific constraints that match each algorithm name to its allowed key sizes, this adds too much details to the abstract definition and quickly makes the model hard to follow.

The second way, which we opted for, is creating a clafer for each available algorithm. We initially created a clafer for each variation of the algorithm. For example, we had separate `AES-128` and `AES-256` clafers that assigned concrete values to all related properties. However, to avoid redundancy, we decided to only create partial instances by not specifying certain properties and leaving the instance generator to decide based on a set of constraints. For example, on Line 43 in Figure 1, we define `AES` and specify that the

key size can be 128, 192, or 256. Based on the key size, we add constraints that set the other properties such as performance and security. In this case, the instance generator would generate three separate instances of `AES`.

3.3 Ordinal Attributes

In cryptography, there are several ordinal algorithm attributes such as security or performance levels. While both properties can be represented as integers that encode discrete values (e.g., 1 to 4 where 4 is the highest), integer values on their own are less comprehensible to the user. It is therefore better to use more meaningful names for these values, such as `Slow`, `Strong`, etc. Each algorithm would then have exactly one security or performance level. While simply having named constants with integer values would work, this may likely lead to silent type issues. For example, if the ranges for security and performance levels are different, but both are defined as integer constants, one might mistakenly assign one of the performance levels to a security attribute.

Coming from a feature modeling background, we initially used an xor-group to model security and performance levels. *Exclusive-or (xor) groups* are a convenient way of modeling mutually exclusive choices. Besides the fact that xor-groups do not support ordering, we also discovered that they are not suited for certain constraints that we need, which is why we use enumeration in our model in Figure 1. To illustrate this, let us use an xor-group instead of an enumeration as follows (`KDA` stands for key derivation algorithm).

```
abstract Algorithm
  xor Security
    Broken
    Weak
    Medium
    Strong

abstract Digest: Algorithm
  ...
```

```
abstract KDA: Algorithm
  ...

pbkdf2: KDA
  ...
  [Security =
       digest.Security]
```

The problem lies in the constraint that specifies that the security level of `pbkdf2` is the same as that of the used digest. Using an xor-group and the above syntax always results in the instance generator finding no instances. It took us some time to figure out that this happens because each concrete `Digest` and `KDA` clafer creates its own instance of `Security` and thus no two `Security` instances would ever be the same, making the constraint always false. Since Clafer combines concepts from class modeling and feature modeling, this behavior is actually by design to represent UML containment, which implies non-sharing of instances. However, it is a bit misleading since the compiler does not issue any errors when specifying such a constraint on the xor-group. For now, we thus use enumerated values as they allow us to compare security or performance values since there is only one instance per enumeration literal in the model.

While Clafer actually supports explicit enumerations (e.g., `enum Performance = Broken | Weak | ..`), this would not allow us to specify order-related constraints such finding algorithms that have a security level higher than weak `[Security > Weak]`, for example. Fortunately, an `enum` is actually syntactic sugar for an abstract clafer and concrete clafers inheriting from it [17], which is what we use to specify ordered enumerated values on Lines 1-13. Having all levels inherit from a single clafer also solves the type issue mentioned above.

3.4 Ignoring Irrelevant Parts of the Model

In our task-oriented approach, the model contains a set of tasks. At any given time, the user is only configuring one task. However, Clafer tries to evaluate and create instances of the *complete* model. This creates lots of redundancies, because instances of clafers that are irrelevant to the current task are still instantiated. For example, assume the model shown in Figure 1 only has one task, `SymmetricEncryption` on Line 94. In this case, if there are no additional constraints specified, we would expect to get four possible instances (i.e., solutions) of this task: three instances of `AES` with each of the allowed key sizes and one instance of `DES`. However, we actually get 54 instances instead.

The reason for these "extra" instances is that Clafer tries to remove all variability in the model by assigning concrete values to all non-abstract clafers. Thus, it would still try to resolve the variability of `pbkdf2` and `scrypt` on Lines 79 and 86 (9 possible combinations). Additionally, whenever the cipher in the `SymmetricEncryption` task is assigned to `DES`, it would still try to resolve all the variability in `AES`. Thus, it would find $1 * 3 * 9$ valid instances where the cipher is `DES` plus $3 * 9$ where the cipher is `AES` resulting in a total of 54 instances, out of which only four are actually relevant and unique with respect to the target task. This behavior results in unnecessary computation time wasted on finding solutions for irrelevant clafers. The number of instances generated also becomes huge as the model becomes larger, making it difficult for a human to compare and reason about them.

Suggested Solutions. We see two possible solutions. The first is supporting a *module system* where we can define parts of the model in different files and then each task would also be defined in a separate file. In each task file, only relevant modules would be included. Based on a poll shown on the Clafer website, a module system is actually the most demanded feature. While a module system would greatly improve the situation, it will still not completely solve the problem in some cases. For example, if the target task needs a key derivation function, the module containing all key derivation functions will be included and in that case, both `pbkdf2` and `scrypt` will still contain variability, resulting in the same issues above.

The second is to introduce a new keyword that marks the particular clafer of interest (e.g., `evaluate`). The Clafer instance generator can then perform some sort of model slicing that only identifies clafers that contribute to the marked clafer in some way. It would then only evaluate these selected clafers. This can even be done by changing irrelevant clafers to abstract ones. Ideally, specifying the target clafer can be set in a constraint (e.g., `[evaluate SymmetricEncryption]`) such that we can easily append the constraint to the existing model depending on the user's selection.

Current (Pragmatic) Solution. To overcome this challenge, we currently post-process the generated instances and filter redundant ones. For example, if the user is configuring the symmetric encryption task, we only look for the instances of that clafer. We then look for the assigned algorithms in the different instances and keep only the unique ones.

3.5 Default Values

Our main use of the model is to find suitable algorithms for a specific task, possibly given specific user requirements specified through a configurator. Many configurators use

a *reconfiguration* approach where all features have default values that the user can change during the configuration process (e.g., the Linux kernel's Kconfig configurator [3]). This approach is suitable for the security domain since we want to provide the user with (the average) secure defaults that should change only if she has specific requirements. However, default values that can later be changed are not currently supported by the reasoning tools used by Clafer.

At first, we attempted to define default values as constraints. For example, to specify that the default key size of symmetric ciphers is 128, we would add `[keySize = 128]` after Line 35 in Figure 1. This would allow us to display the default value in the configurator. Thus, if the user just clicked next, she would still have secure values. It also means that we would only need to specify key size values for ciphers that have a key size other than 128, making the task of creating the model easier. When we defined default values this way in the abstract clafer `SymmetricCipher` and then assigned a different value in its concrete clafers (e.g., Line 55 for the concrete `DES` clafer), the solvers saw this as a contradiction since they interpreted it as `keySize` should be *both* 128 and 56. Thus, instead of overriding the default value, it added the constraint to the list of existing constraints.

After our initial attempts, we have been informed by the Clafer language designers that in principal, Clafer supports default values by using a second type of assignments. While regular assignments like `keySize -> integer = 128` only use an equal sign, assignments of default values like `keySize -> integer := 128` contain an additional colon. However, none of the current backends support this notation, which is why we cannot use default value assignments at the moment.

Suggested Solutions. The best solution is to have at least one of the solvers support default values. We also believe that the current way of defining default values may be unintuitive and more prone to typing errors. Thus, we suggest introducing a `default` keyword to explicitly state the default value of a clafer, avoiding any confusion or hasty typos.

A sub-optimal solution is to selectively handle constraints in a concrete clafer, before calling the solvers. If a constraint in the child clafer exists in the parent abstract clafer, then *ignore or replace* the constraint in the abstract clafer. The problem here is that this rule will not be valid all the time. For example, a constraint in the abstract clafer could be that `[keySize >= 64]` while a constraint in the concrete clafer is `[keySize = 56]`. In this case, the concrete clafer constraint should not replace the abstract clafer and the correct behavior is that the model should not generate instances since there is a conflict (i.e., this concrete clafer actually violates a constraint). A heuristic could be to only replace constraints if they both involve only equality.

Current Handling. We do not currently include default values in the model itself due to the above issues. Instead, to be able to display the appropriate default values to the user in the configurator, we use an external properties file that includes different default values for different properties. These values are only used in the configurator and do not alter the model itself.

3.6 Back-ends and Optimization

Since we use large integer values for properties such as key size or number of iterations, we mainly used the Choco solver backend due to its faster performance [1]. However, using Alloy may be desirable since some of the new language features in Clafer 0.4.1 are currently only supported by the Alloy backend. For example, short-hands such as including allowed key sizes directly in the clafer definition `keySize -> 128, 192, 256` instead of defining the clafer and then adding the allowed values as a constraint. One way to overcome the slow performance caused by the large integer values is to introduce units and use smaller values. For example, instead of defining key sizes as bits, we can use bytes in the model and display bits to the user. Similarly, instead of specifying 1000 for iteration size, we can specify 1 and have the application logic multiply it by 1000.

3.7 Configuration

We created an Eclipse plugin that provides a configurator for the model. We used the Java APIs provided by the Choco solver project to access the JavaScript compiled version of the model programmatically [18].

3.7.1 Identifying Allowed Ranges

One challenge when creating the configurator input controls is to determine the allowed input ranges and only display those to the user. Such allowed values or ranges can be specified in the Clafer model. For example, `[iterations = 1000 || iterations = 1500 || iterations = 2000]` or `[outputSize >= 128]`. The problem is that again these ranges look the same as any other constraint in the model. Therefore, there is no easy way to distinguish between constraints that do not affect display (or sanity checks) in the configurator and those that do. We can of course apply heuristics, but this means that we have to check every constraint to see if the heuristic holds or not. We currently only determine the type of input control in the configurator based on the clafer type (integer, string, enumeration, etc.).

3.7.2 Querying

The main purpose of the model is for application developers to find valid solutions to the cryptography tasks they want to perform. Our idea is to pose high-level questions to the application developer as applicable to each task. For example, if the developer wants to perform password-based encryption, one question may be *"Is high performance important to you?"* If the developer answers yes, then we need to restrict the performance of the cipher used to `Fast` or `VeryFast`. In other words, we need to append the following constraint `[cipher.performance.Fast || kda.Performance.VeryFast]` to the `PasswordBasedEncryption` task in the model.

We currently use the provided Java APIs to create a new model with the appended constraints and then call the solver. While this works so far, it requires us to recompile the model and trigger the instance generator again, which can be costly. We avoid multiple invocations of the instance generator by appending all constraints at once and then calling the instance generator. However, this can of course mean that no valid instance can be found. Currently, we only notify the user when this happens but have still not used the UnSAT Core provided by each of the backends to indicate possible solutions to the user. Since the end user is not aware of the underlying model, the challenge would be to present the information provided by the UnSAT core in a way that is understandable to the user.

4. OTHER MODELING LANGUAGES

While this paper focuses on presenting our experience with modeling cryptography in Clafer, the issues we faced led us to also look at related modeling paradigms and languages. We look at three directions: ontology languages, software taxonomies, and other textual variability languages.

4.1 Ontology Languages

There has traditionally been a clear distinction between ontologies and models [19]. On the one hand, ontologies are a means to capture domain knowledge in a single machine-readable representation. They reflect a shared and purpose-independent understanding of a domain. On the other hand, models are seen as any other form of representation of a certain system or domain that is not an ontology. For example, class models in UML, variability models in Clafer, or traditional feature models all qualify as models. A model is a representation from a specific viewpoint and only for a limited purpose and is, thus, not shared, but only used by a small user group. Another important difference is the open- vs. closed-world assumption. Ontologies view everything not specified as unknown (open-world), while models define it as forbidden (closed-world) [19], which results in different approaches to reasoning. However, in recent years, commonalities between models and ontologies have increasingly been emphasized [20, 21], sometimes even as far as claiming that ontologies are a subset of models. Based on this reduced gap between ontologies and modeling, we look at whether ontology languages are suitable for our application scenario. Although there are several languages available [22], we focus on the Web Ontology Lanugage (OWL) as it is the most popular and best supported language.

OWL is a family of ontology languages for the semantic web. Hierarchical structures can be modeled in OWL by means of classes, individuals, superclass-subclass relationships, and properties. Similar to Clafer, OWL does not support default values (if a class once defines a property value, no subclass can override it). However, contrary to Clafer, one individual in OWL can belong to multiple classes. As OWL is an ontology language, it focuses on machine-readability and follows the open-world assumption. This makes its syntax so verbose that even small OWL models become difficult to read for humans. OWL has been widely adopted and, thus, a great range of tool support exists. However, contrary to Clafer, tools focus on classification and not instantiation [23].

Since ontologies represent a shared understanding of all experts of a certain domain, we also look at whether ontologies for the cryptography domain already exist. If so, they can either replace our Clafer model or we can at least reuse some of the encoded knowledge. Gyrard et al. [24] design IT-Security-related ontologies, including one defining cryptographic primitives. However, for our purposes, their ontologies suffer from several drawbacks. First, they do not include all information we need in our model. For example, supported key lengths for symmetric block ciphers or performance properties are missing. Second, none of the ontologies define high-level implementation tasks such as password-based encryption, but remain on the protocol level (e.g., OpenPGP or WPA2). This means that we would still have to extend these ontologies. However, since some of these available ontologies define a big variety of cryptographic algorithms, we could at least make use of this already encoded knowledge and complement missing information in our model, where necessary.

4.2 Software Taxonomies

Cryptography is a algorithm-heavy domain, so it is no surprise that the Clafer model we present in Figure 1 consists mostly of algorithms. Cleophas et al. [25] propose software taxonomies to model domains from an algorithmic perspective. In their paper, they define a taxonomy as models that represent a conceptual hierarchy of all domain-relevant algorithms. Along this hierarchy, algorithms are refined and become more concrete. For cryptographic algorithms, `Cipher` would be higher than `Symmetric Cipher`, which are then followed by the actual algorithms such as `AES` or `DES`. Such taxonomies are thus similar to our representation of cryptographic algorithms in Clafer. However, we also define high-level tasks in the model, which are directly used for product generation and which taxonomies are not capable of capturing. If we used a software taxonomy to model the cryptography domain, we would need to find a second modeling language to define the tasks. In summary, while we do see some similarities between them and our way of modeling cryptographic algorithms in Clafer, their focus on algorithms is too limited for our purposes as we also want to specify non-algorithmic elements.

4.3 Other Variability Modeling Languages

We now discuss related variability modeling languages that may be suited to the cryptography domain. Since Clafer is a textual variability modelling language whose capabilities exceed those of traditional feature models, we focus our discussion on such modeling languages. Based on a recent survey [26], we identify three relevant languages (apart from Clafer): TVL [27], IVML [28], and VSL [29].

TVL. The Textual Variability Language (TVL) [27] was introduced as a textual notation that covers all aspects of feature modeling, but provides extra functionality (e.g., attributes) that is necessary to model realistic cases. Our experience with modeling cryptography stresses such needs.

In contrast to Clafer, TVL supports modularization through a simple module system that relies on preprocessing. In other words, contents of other files can be included into a module by using `include` statements. This simple support already allows us to better structure our model and separate different algorithm types in different files. However, based on their survey, the authors conclude that users want a more sophisticated modular system that includes explicitly exporting features or attributes in a module. Moreover, similar to Clafer, TVL does not support default values or ignoring parts of the model.

IVML. The INDENICA Variability Modeling Language (IVML) is designed to support the variability modeling requirements relevant to supporting the customization of complex service platform ecosystems [26,28]. IVML provides interesting language features relevant to our application. For example, IVML supports ordered enumerations. IVML also supports default values and allows them to be overridden later on in the model. We note that IVML differentiates default values from regular assignments in constraints by using `=` for default values and `==` for assignments.

IVML has a module system that allows dividing a model

into several sub-models in different files. IVML also allows more control over how the model is evaluated through the `eval` keyword. The keyword tells the instance generator to evaluate this constraint first. This leads to the assignment of related variable values which in return reduces the search scope when evaluating the whole model. This is a different behavior from what we suggest in Section 3.4, but it may be the case that it can be adapted to ignore certain parts of the model and only evaluate a specific task and its related features, without continuing to fully evaluate the model. Additionally, IVML supports feature model versioning to support evolution. This might be relevant as security specifications evolve over time.

VSL. The textual Variability Specification Language [29] provides advanced variability modeling support including cardinality-based feature modeling [11]. Similar to IVML, VSL also supports default values. VSL provides modularization mechanisms through keywords such as `featuremodel`, `config`, `configLink`, and `entity` that represent a feature model, a configuration of a feature model (partial or complete), a relationship between two feature models, and a combination of any of the above, respectively. Additionally, entities themselves can be composed of other entities. This allows for a better separation of concerns and provides some form of modularity while designing the model. It also separates instances from the general (or abstract) definitions in the model. Similar to Clafer, VSL's syntax is relatively simple once the user grasps all special keywords for modularity and more complex constraints (links).

4.4 Discussion

It has long been suggested to integrate ontologies into software engineering, either as a replacement of domain models or at least as the basis for more purpose-focused domain modeling. However, it seems that due to the different focus of modeling and ontology engineering, there are still obstacles for wider adoption. We, therefore, believe that variations of variability modeling are better suited for our needs.

Our investigation of software taxonomies leads us to a similar result. While we do see some similarities between them and our way of modeling cryptographic algorithms in Clafer, their focus on algorithms is too limited for our purposes as we also want to specify non-algorithmic elements.

In terms of variability modeling, more detailed investigation of the above modeling languages is needed. Based on our brief comparison, IVML seems promising for our needs since it solves most of the challenges we faced. However, we (subjectively) see that Clafer's syntax is simpler to understand. It additionally has an extensive tool suite and APIs which makes it very easy to use. Some of the language features we need may also get implemented in future versions of Clafer. However, for an objective comparison, we have to create an IVML model to practically investigate whether we can successfully model all aspects of cryptography.

5. RELATED WORK

In this paper, we discuss Clafer mainly as a variability and domain modeling language. It has been used before for such purposes as modeling the Linux Kernel variability model [3], Oracle's Merchandise Financial Planning domain model [30], and a bike sharing system [31]. Due to its expressiveness, however, Clafer has also been applied to other application scenarios, including metamodels and Product-Line architecture models [1, 30].

Berger et al. [32] note that the lack of experience reports on variability modeling techniques may impede the progress of variability modeling research. Through several studies, they look at variability modeling experiences mainly in the systems and automotive domains [3,33]. While these studies discuss variability modeling in general, Alférez et al. [34] investigate a domain-specific variability modeling language VM for its suitability for the video domain. Furthermore, Hubaux et al. [35] report on the industrial usage of TVL after interviewing experts from the industry. Interestingly, their participants mention the lack of convenient features such as default values as an issue, something we also encountered when modeling with Clafer.

However, apart from Murashkin's work on automotive architecture models [1], none of the above studies look explicitly at how to use Clafer and the modeling trade-offs involved. Moreover, to the best of our knowledge, there is no work that looks at variability modeling of cryptography. That said, cryptographers do divide cryptographic algorithms into different classes such as asymmetric and symmetric encryption algorithms [36]. The Clafer model's structure mirrors this hierarchy and uses different classes of algorithms to accomplish different tasks.

6. CONCLUSION

In this paper, we presented our experience with modeling cryptographic components with the Clafer modeling language. We reported on the different design decisions we took as well as conceptual and language challenges we faced. Our experience with using Clafer has been positive so far, but we believe that some of the main issues we ran into can be overcome with support for a module system. We also discussed related variability modeling languages that may be suited for our needs. We believe that our reported experience is useful for variability-modeling language designers, as well as both researchers and practitioners who want to use Clafer or want to apply variability modeling in similar domains.

7. ACKNOWLEDGMENTS

Our reviewers pointed us to the work on taxonomies. We appreciate M. Antkiewicz's feedback and constant help with Clafer. Thanks to: J. Liang for clarifying the Choco-solver APIs, F. Günther for help with crypto. domain knowledge, and R. Kamath for the configurator. This work is funded by the DFG, project E1 within CRC 1119 CROSSING.

8. REFERENCES

[1] A. Murashkin. Automotive electronic/electric architecture modeling, design exploration and optimization using Clafer. Master's thesis, University of Waterloo, 2014.

[2] M. Rosenmüller, N. Siegmund, T. Thüm, and G. Saake. Multi-dimensional variability modeling. In *Proc. of the Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*. 2011.

[3] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In

Proc. of the IEEE/ACM Int'l Conference on Automated Software Engineering (ASE). 2010.

[4] K. Bąk, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wąsowski. Clafer: unifying class and feature modeling. *Software & Systems Modeling*, 2014.

[5] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proc. of the Conference on Computer and Communications Security (CCS)*, 2013.

[6] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory love Android: An analysis of android SSL (in)security. In *Proc. of the Conference on Computer and Communications Security (CCS)*, 2012.

[7] S. Arzt, S. Nadi, K. Ali, E. Bodden, S. Erdweg, and M. Mezini. Towards secure integration of cryptographic software. In *Proc. of the SIGPLAN Symposium on New Ideas in Programming and Reflections on Software at SPLASH (Onward!)*, 2015.

[8] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. 1996.

[9] D. Menascé. Security performance. *IEEE Internet Computing*, 7(3):84–87, May 2003.

[10] X. Wang and H. Yu. How to break md5 and other hash functions. In *Advances in Cryptology–EUROCRYPT 2005*. Springer, 2005.

[11] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[12] Alloy. http://alloy.mit.edu.

[13] Choco. http://www.emn.fr/z-info/choco-solver.

[14] M. Antkiewicz, K. Bąk, A. Murashkin, R. Olaechea, J. H. J. Liang, and K. Czarnecki. Clafer tools for product line engineering. In *Proc. of the Int'l Software Product Line Conference (SPLC) Co-located Workshops*. 2013.

[15] C. Percival and S. Josefsson. The scrypt password-based key derivation function. *Internet Engineering Task Force (IETF)*, 2012.

[16] National institute of standards and technology (NIST). secure hashing – approved algorithms. http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html.

[17] M. Antkiewicz. Clafer cheat sheet. http://t3-necsis.cs.uwaterloo.ca:8091/Clafer%20Cheat%20Sheet.

[18] Accessing Clafer models programmatically. http://www.clafer.org/2014/08/accessing-clafer-models-programmatically.html.

[19] U. Aßmann, S. Zschaler, and G. Wagner. Ontologies, meta-models, and the model-driven paradigm. In *Ontologies for software engineering and software technology*. Springer, 2006.

[20] C. Atkinson, M. Gutheil, and K. Kiko. On the relationship of ontologies and models. In *Proc. of the 2nd Workshop on MetaModelling (WoMM)*. 2006.

[21] K. Czarnecki, C. Hwan, P. Kim, and K. Kalleberg. Feature models are views on ontologies. In *Proc. of the Int'l Software Product Line Conference (SPLC)*, 2006.

[22] D. Kalibatiene and O. Vasilecas. Survey on ontology languages. In *Perspectives in Business Informatics Research*. Springer, 2011.

[23] K. Bąk. *Modeling and Analysis of Software Product Line Variability in Clafer*. PhD thesis, University of Waterloo, 2013.

[24] A. Gyrard, C. Bonnet, and K. Boudaoud. An ontology-based approach for helping to secure the etsi machine-to-machine architecture. In *Proc. of the IEEE Int'l Conference on the Internet of Things (iThings)*. IEEE, 2014.

[25] L. Cleophas, B. W. Watson, D. G. Kourie, and A. Boake. Tabasco: a taxonomy-based domain engineering method. In *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*. South African Institute for Computer Scientists and Information Technologists, 2005.

[26] H. Eichelberger and K. Schmid. A systematic analysis of textual variability modeling languages. In *Proc. of the Int'l Software Product Line Conference (SPLC)*. 2013.

[27] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, 76(12):1130 – 1143, 2011. Special Issue on Software Evolution, Adaptability and Variability.

[28] H. Eichelberger, S. E. Sharkawy, C. Kröher, and K. Shmid. INDENICA variability modeling language: Language specification (version 1.26). Technical report. http://projects.sse.uni-hildesheim.de/easy/docs/ivml_spec.pdf.

[29] M.-O. Reiser. Core concepts of the Compositional Variability Management Framework (CVM) – a practitioner's guide. Technical report. http://www.eecs.tu-berlin.de/fileadmin/f4/TechReports/2009/tr-2009-16.pdf.

[30] Clafer example models. http://t3-necsis.cs.uwaterloo.ca:8091/.

[31] M. H. ter Beek, A. Fantechi, and S. Gnesi. Applying the product lines paradigm to the quantitative analysis of collective adaptive systems. In *Proceedings of the 19th Int'l Conference on Software Product Line*, 2015.

[32] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A survey of variability modeling in industrial practice. In *Proc. of the Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*, 2013.

[33] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski. Three cases of feature-based variability modeling in industry. In *Model-Driven Engineering Languages and Systems*. Springer, 2014.

[34] M. Alférez, J. A. Galindo, M. Acher, and B. Baudry. Modeling variability in the video domain: Language and experience report. Technical report, 2014.

[35] A. Hubaux, Q. Boucher, H. Hartmann, R. Michel, and P. Heymans. Evaluating a textual feature modelling language: Four industrial case studies. In *Software Language Engineering*. Springer, 2011.

[36] D. R. Stinson. *Cryptography: theory and practice*. CRC press, 2005.