Modeling Variability in the Video Domain: Language and Experience Report

Mauricio Alférez, José Angel Galindo Duarte, Mathieu Acher, Benoit Baudry

Modeling Variability in the Video Domain: Language and Experience Report

Mauricio Alférez[1], José A. Galindo, Mathieu Acher, Benoit Baudry

DiverSE team at INRIA Rennes
Campus Universitaire de Beaulieu
35042 Rennes cedex, France

Abstract

Context: Providers and consumers of computer vision algorithms need to test their solutions using realistic videos as input test data. The current practice is to find existing videos or to film them outdoors. However, the effort of manually collecting or filming videos in diverse scenarios is highly resource intensive and not economically viable. In an industrial project, we faced the challenge of providing more automation and control to produce video variants using a video generator. A key problem is to capture and exploit information of what can vary within a video.

Objective: This work seeks to describe the variability requirements in the video domain and to provide practical solutions for video variability modeling to support a generative approach for synthesizing video sequences.

Method: In this paper, we report about a new domain-specific variability modeling and configuration language, called VM, resulting from the close collaboration with industrial partners. We expose the requirements and advanced variability constructs required to characterize and realize variations of physical properties of a video (such as objects speed or illumination).

Results: The results of our experiments and industrial experience show that VM is effective to model complex variability information and can be exploited to synthesize video variants.

Conclusions: We concluded that basic variability mechanisms are useful but not enough, attributes and multi-features are of prior importance, and meta-information is relevant for efficient variability analysis. In addition, we questioned the existence of one-size-fits-all variability modeling solution applicable in any industry. Yet, some common needs for modeling variability are becoming apparent such as support for attributes and multi-features.

Keywords: VM, variability modeling, product line engineering, highly configurable systems, textual specification languages

1. Introduction

Many organizations have to manage variability, encoded as features or configuration options, to extend, change, customize or configure multiple kinds of artifacts (e.g., hardware devices, operating systems or user interfaces) [1].

We faced the challenge of synthesizing video sequence variants in an industrial project involving providers and consumers of vision algorithms. This challenge is very original given the kind of artifact that varies (videos) and its domain (video analysis). In this domain, practitioners need to obtain videos to test their video analyzers. Typically, video analyzers are complex systems as they assemble different vision algorithms, such as objects tracking and patterns detection, into a video processing chain. In this setting, the more diverse is the input set of videos, the better is the chance of characterizing a video analyzer in terms of performance, reliability, robustness or any other non-functional property.

The current practice to obtain videos is to collect or to film them. Our experience with three organizations developing or using video analyzers is that the current practice is not economically viable or sufficient for testing video analyzers. We give more specific details in Section 2, but essentially, high costs and complex logistics are required to film videos in real locations.

The difficulties of the current practice to obtain an in-

Email addresses: mauricio.alferez@inria.fr (Mauricio Alférez), jagalindo@inria.fr (José A. Galindo), mathieu.acher@inria.fr (Mathieu Acher), benoit.baudry@inria.fr (Benoit Baudry)
[1]Corresponding author

put set of videos brought our attention to more automation and control. In this paper we address two research questions:

- **RQ1:** What are the variability requirements in the video domain?
- **RQ2:** How to capture what can vary within a video and then automate the synthesis of variants?

This paper introduces a generative approach and reports on our experience for addressing these research questions in a collaborative project involving three organizations.

This paper provides the following contributions:

- Requirements for modeling variability requirements in the video domain.
- Practical solutions for video variability modeling represented by the VM language and supporting tool.
- Analytic and empiric results that show that VM fulfills its requirements and how meta-information in VM models reduces the time required to perform automated analysis.
- Discussion and analysis about our results and how the reader can learn from our experience.

A central finding is that we needed to design a new *language*, called VM, for modeling variability of videos and thus allowing the synthesis of video variants. We first learned that Boolean variability constructs are clearly not sufficient in the video domain as it has to deal with numeric parameters (also called *attributes*) and features appearing several times (also called *clones* [2] or *multi-features* [3]). For example, *speed* is an attribute of a vehicle that can take different values across videos, and *vehicle* is a multi-feature as each video can show several vehicles configured with different speed values. In addition to the support of attributes and multi-features, we added several language constructs in VM. For example, default values, deltas to discretize continuous domain values, objective functions to filter relevant configurations, multi-ranges for attributes domains, meta-information, etc.

Automated analysis tools [4, 5] exploit the information in VM models to generate testing configurations, i.e., values assigned to features and attributes that define a video. The information made possible by VM is crucial for two reasons. First, we can better *control* the way testing configurations are generated – precluding irrelevant videos while ensuring the covering of relevant testing scenarios. Second, solvers can better *scale* thanks to meta-information associated to features and attributes. We notably show in Section 4 that meta-information reduces the time to reason and produce configurations.

The design of VM has been influenced by the technical realization (i.e., the solution space) of variability. We jointly developed an end-to-end solution to generate testing configurations that are then fed to a video generator. The connection with the realization layer – through configuration files – validates the adequacy of VM. We are now able to synthesize thousands of video variants, something clearly impossible at the beginning of the project.

The point of the paper is not to present yet another variability language or to provide details on video analysis, generation, or automated feature models analysis operations. We rather want to highlight the specific requirements we faced in the video domain, leading to the design and reuse of existing (or novel) variability constructs. Our experience, as others [6, 7, 8], question the existence of a one-size-fits-all variability solution applicable in any industry.

Also, to answer to our industrial problems, we focus on the expressive power of VM and how its affects scalability of feature analysis operations (e.g., generation of a t-wise covering configurations [5]). Based on our use of VM in one particularly complex video sequences generator, we conclude that VM was expressive enough to cover complex variability in the video domain. In the case of having more users we would need to evaluate technical and psychological dimensions in language construction, for example, usability, learn-ability, tooling support, interchange, etc.

The remainder of this paper is organized as follows: Section 2 describes further the industrial problem of video generation. Section 3 covers a list of requirements and how they impacted the design of VM. Section 4 reports on practical considerations, empirical results about the adequacy and effects of the new constructs of VM, and compares VM to other existing variability languages. Section 5 discusses threats to validity. Section 6 describes other related work, and finally, section 7 presents concluding remarks and summarizes the lessons learned.

2. Industrial Motivation

We faced the challenge of synthesizing a high number of diverse video variants in an industrial project called MOTIV. The project aims to evaluate computer vision

algorithms such as those used for surveillance or rescue operations. A targeted scenario is usually as follows. First, airborne or land-based cameras capture on-the-fly videos. Then, a video processing chain analyzes videos to detect and track objects, for example, survivors in a natural disaster. Based on that information the operation manager triggers a rescue mission quickly based on the video analysis information.

Two companies are part of the MOTIV project as well as the DGA (the French governmental organization for defense procurement). The two companies develop and provide algorithms for video analysis. The diversity of scenarios and signal qualities poses a difficult problem for all the partners of MOTIV: which algorithms are best suited given a specific application? From the consumer side (the DGA), how to choose, select and combine the algorithms? From the provider side (the two companies), how to guarantee that the algorithms meet a large variety of situations? How to propose innovative solutions able to handle new situations?

Our partners need to collect videos to test their video analysis solutions and detection algorithms. Synthesizing a high diversity of videos is difficult as there are many ways in which a video can change (e.g., physical properties, types and number of objects, backgrounds). However, synthesizing videos is still more feasible than filming them in real environments. Our partners calculate that an initial input data set of 153000 videos (of 3 minutes each), corresponds to 320 days of video footage and requires 64 years of filming outdoors (working 2 hours a day). Note that these numbers were calculated at the starting point of the project based on the previous experiences of the partners.

A related problem to randomly synthesizing videos is that practitioners ignore what kinds of situations are covered or not by the set of videos. Therefore, it is not possible to ensure the quality of the test-suite for detection algorithms in all situations. Overall, more *automation* and *control* are needed to synthesize video variants and cover a diversity of testing scenarios.

The first step for automation support was taken by our partners. They created a video generator to produce customized videos based on user preferences that were hard-coded during the first versions. The left-hand side of Figure 1 shows this *old approach* where the only actors are the developers. They had to comment lines or modify variable values directly in the video generator code to change the physical properties and objects that appear in each video.

When the video generator was more stable, the developers decided to create configuration files to communicate input values instead of hard-coded them. In particular, they employed Lua configuration files which have a simple structure based the pattern *parameter* = *value*. Then, developers used Lua code and proprietary C++ libraries, developed by a MOTIV partner, to process those configuration files and execute algorithms to alter, add, remove or substitute elements in base videos[2].

The Lua configuration files used helped to decouple implementation from input data, however, their simplistic nature presented at least three drawbacks: i) they lack of relationships and constraints between configuration options; this information is very important to scope the family of videos and to exclude invalid configurations; ii) every change in the video generator requires to change n configuration files, thus, this task becomes error prone and tedious as the n value is high (about 500 in the first round of experiments according to our partners); and iii) they require that developers understand very well the implementation details of the video generator. Therefore, developers had to guarantee that the values defined in configuration files were between the limits allowed by the video generator and that each configuration is valid (e.g., there are not conflicts between configuration options and their values).

The right-hand side of Figure 1 shows the VM approach that improves the process to generate videos. The key idea is that developers and domain experts model variability in the video domain using a model written in VM. Then, the VM tool generates Lua configuration files and connects with the video generator to produce videos.

Figure 2 shows an example of a particular scene (part of a video video sequence). This scene has a countryside background and only one object.

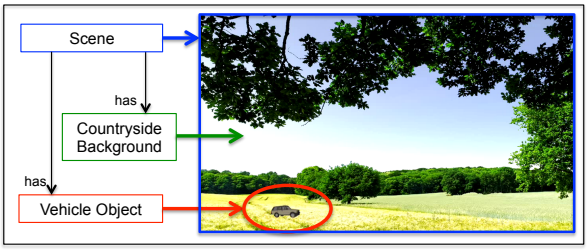

Figure 2: A video with countryside background and only one vehicle object.

[2]Lua is a widely used programming language (`http://www.lua.org/`). Details about the computer vision algorithms that synthesize video variants are out of the scope of the paper.

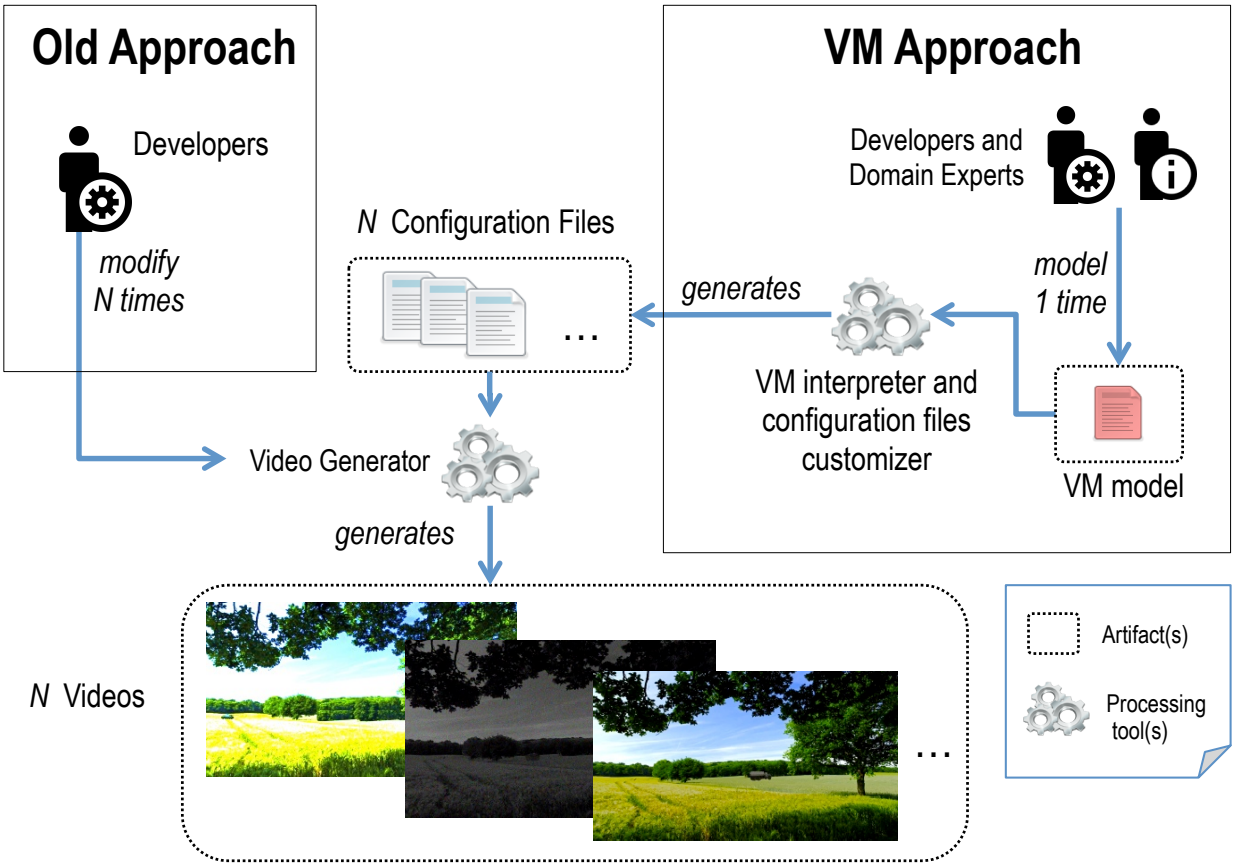

Figure 1: Old process compared to the VM-based process for video generation

3. Variability Modeling and Configuration Language and Tool –VM

VM is a textual, human readable and writable language and tool that supports extra variability modeling in the video domain and has a special focus on benefiting automated support scalability. VM was created in three iterations. Each one adds more complexity and functionality to the language and its tool support.

We employed project meetings to elicit requirements. During a period of about nine months, we had four large meetings, two individual meetings with the main developer of the video generator, and regular conversations with participants by email. Currently, the main direct user of VM in the MOTIV project is a team of three people from Inria, France.

Figure 3 presents an overview of the main characteristics (or solutions) supported by VM and the elicited requirements that originated them[3]. They follow a top-down decomposition where the more general characteristics appear on top. We used grey boxes to identify the novel characteristics offered by VM regarding existing variability languages (see Section 4.3 for more details). We mapped requirements to characteristics by writing requirement numbers (R#) at one side of characteristics numbers (C#) or inside a box with a dashed outline to group several characteristics. Readers can find the complete grammar of VM and the variability model online[4].

In the following, we will describe the three iterations based on requirements followed by the concrete solutions provided by VM. Most of the requirements are for the design of the language itself, however, there are also requirements for the tool that will process the models written in the language. For example, requirement R12 is concerned about the effects of the novel VM con-

[3]We prefer to use the terms "characteristic" and "solution" instead of "feature" when describing the properties of the VM itself, to avoid confusions with the features of a video sequence modeled using the VM language

[4]`https://github.com/ViViD-DiverSE/`

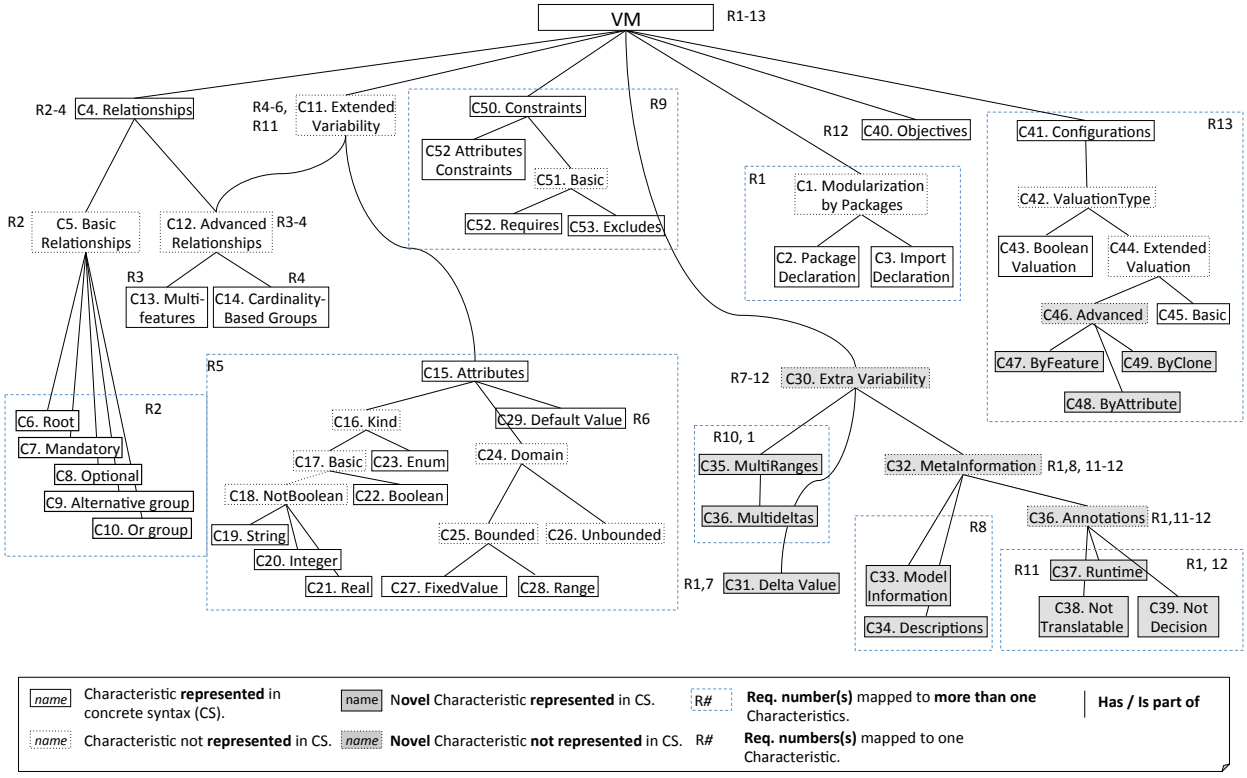

Figure 3: Main language features supported by VM.

structs regarding the performance of automated analysis operations in large and complex VM models.

3.1. Iteration 1: Basic variability modeling

R1. Be scalable. One of our main interests was to make a solution with the ability to handle a growing amount of configuration parameters in future video generator versions.

Two main issues motivated the scalability requirement. First, after adding or changing parameters in the video generator, the developers had to verify all the Lua configuration files to guarantee that the parameter values were between their limits and covered the situations that they wanted to test. This manual process is not scalable, time intensive and error prone.

Second, the MOTIV project participants are interested in automated analysis operations such as the generation of pair-wise coverage in video configurations. However, the time employed to perform complex automated analysis operations over attributed feature models may grow exponentially with the number of combinations between features and attribute values in not trivial feature models. Therefore, it is important to guarantee that the difficulty and time for managing the complexity of a video variability model goes up roughly linearly with increments in its size. In particular, our goal is that the time to perform automated analysis operations over a specification (especially, the pair-wise coverage operation) grows linearly or slower with larger (e.g., with more configuration parameters) or more complex specifications (e.g., with more cross-tree dependencies between configuration parameters).

→ **C1-C3. Modularization by Packages.** A way to support scalability in variability specifications (and in any software design in general) is to achieve a logical partitioning of a specification to be each part more manageable for the purpose of implementation, understanding and maintenance. Therefore, one part of the solution for the scalability (R1) was to provide modularization by packages to group configuration parameters, and import declaration to reference features and other elements defined in other packages.

Section 3.3 will present more solutions for the scalability requirement that deal with the challenge of specifying large domains for numeric attributes (introduced in the second iteration). In addition, the requirement R12 (“Benefit automated support”) motivates the use

language constructs that benefit automated analysis of the specifications written in VM.

R2. Organize common and variable configuration options. There are three issues that motivated this requirement: First, there was not a logical way to organize configuration options (a.k.a. *parameters*) apart from using comments to label groups of related options. That approach makes it impossible to establish relationships between options in different groups, and hierarchical dependencies between options in a group. For example "object" and "vehicle" appeared in the group labeled "scene_options", however, the 3-levels logical hierarchy *vehicle is an object* and *object is a type of scene option* was not explicit but necessary to understand the video domain.

Second, there was not a way to distinguish parameters that are common to all the videos from the ones that are optional. This issue can give room to unintended configurations. For example, "background" is a parameter that should have a valid value representing its type, however, nothing prevents to the developers of treating it as an option and assign it a "disabled" value by mistake.

Third, it is difficult to know which are the acceptable alternative variations that could be used as values for optional parameters. This information is important when generating video configurations automatically.

→ **C5. Basic Relationships.** Our solution for the issues related in requirement R2 was to define configuration options as *features* and relate them in a block called "Relationships" (Characteristic C4. "Relationships" in Figure 3). That block, exemplified in Listing 1, shows a features' hierarchy where the selection of a feature (the *child* feature) depends on the selection of a more general feature (the *parent* feature). On the other side, incompatibilities are expressed using groups of alternative features where the selection of a grouped feature may be incompatible with the selection of other grouped features. In this iteration, we only implemented basic features' relationships (mandatory and optional) and groups (oneOf and someOf).

The different types of relationships available in VM are summarized next:

C6. Root. Following traditional terminology from graph theory, a feature without a parent is called a *root* feature. In VM, each relationships block can have a root feature. "scene" (Listing 1, Line 2) is an example of root feature.

C7. Mandatory. Child feature is required. Corresponds to features that will be included in all possible video configurations such as "background" (Line 3).

C8. Optional. Child feature is not required. This corresponds to features that may be or may not be selected as part of a video configuration. Optional features use the symbol "?" before their name, for example "? objects" (Line 10).

C9. Alternative group. One of the sub-features must be selected. A alternative group is represented using the word "OneOf". For example, we specified in Lines 4-8 that developers can choose only one "background" between "urban", "countryside" and "desert".

C10. Or group. At least one of the sub-features must be selected. An Or group is represented using the word "someOf". For example, we specified in Lines 11-14 that "vehicle" and "man" are two not exclusive alternative kinds of objects that can be placed in a scene.

3.2. Iteration 2. Extended variability modeling

This iteration focuses on extended variability (Characteristic C11 in Figure 3).

R3. Limit the number of configurable copies. We saw that some parameters can have several configurable copies in the same configuration file. For example, in the Lua configuration files, the configuration parameter "vehicle" appears several times but with different suffixes (e.g., vehicle1, vehicle2, etc.). Currently, there is no way to know the maximum or minimum number of vehicle copies that can be enabled and configured in each video without studying carefully the video generator.

R4. Limit the number of selectable configuration options from a group. The video generator allows changing scenes during the execution of the video. For example, a video may show a vehicle that travels from a "countryside" to an urban" scene. The feature groups created in the iteration 1 cannot model this situation; "oneOf" is too restrictive and "someOf" is too loose and may allow up to three scenes, which is not viable as the videos will take too much time to generate. Therefore, it is necessary to limit the minimum and maximum numbers of selectable configuration options from a group.

→ **C12. Advanced Relationships.** The solution for requirements R3 and R4 are two advanced relationships (Characteristic C12 in Figure 3) in the relationships block: Muti-features and Cardinality-Based Groups.

→ **C13. Multi-Features.** Our solution for requirement R3 was to use cardinalities before each feature name to specify the minimum and maximum configurable copies it can have. To create a configuration of a video, a multi-feature and all its children features are cloned into copies, and each copy can be configured individually.

The specification of a multi-feature follows one of two patterns placed before a feature name:

```
1  Relationships:
2  scene { //mandatory root feature
3    background { //mandatory feature
4      oneOf { // XOr feature group
5        urban  // grouped feature
6        countryside //grouped feature
7        desert //grouped feature
8      }
9    }
10   ? objects { //optional feature
11     someOf { //Or feature group
12       [1..5] vehicle  //short way to express multi-features
13       cloneBetween 1 and 10 man /*readable but verbose way to express multi-features. It is
             equivalent to [1..10] man*/
14     }
15   }
16 }
```

Listing 1: Feature relationships example

(1) `[minVal..maxVal]` (Listing 1, Line 12), or
(2) `cloneBetween minVal and maxVal` (Line 13).

In both patterns, *minVal* and *maxVal* are the minimum and maximum number of allowed feature copies.

→ **C14. Cardinality-Based Groups.** Our solution for requirement R4 was to add cardinalities to groups to specify the selection of a minimum and maximum numbers of grouped features. The specification of a cardinality-based group follows one of the two patterns:
(1) `[minVal..maxVal]`, or
(2) `someBetween minVal and maxVal`.

R5. Represent Diverse Kinds of configuration options. The VM relationships created in the Iteration 1 were enough to model the most simple configuration options such as to activate or deactivate objects in the video. However, those "yes/no" options, represented as Boolean features, were not enough to describe the domain and range of possible values of other types configuration parameters, such as numbers or chains of characters.

→ **C15. Attributes.** Our solution to requirement R5 was the "Attributes" block that defines properties associated to the features expressed in the Relationships block. VM supports basic types (boolean and not boolean) and enumeration attributes (Characteristics C16-C23 in Figure 3). Listing 2 shows 6 examples of attributes of types integer (int), enumeration (enum), float point (real) and chain of chars (string).

→ **C24-C28. Bounded and Unbounded Attributes Domains.** There are two kinds of attributes in terms of their domains: bounded or unbounded. Bounded means that the attribute has a fixed value or that it may have one value between a range. Unbounded means that an attribute has not a defined minimum and maximum values.

The most basic bounded attribute is the one that has a fixed value. The value of those attributes can not be changed in any configuration and are comparable to constant values in programming languages. For example: "real man.speed = 10.5" means that if a video has one or more men objects, their speed is always 10.5.

The attributes of our running example are bounded. Line 2 stores a comment for each scene (@NT, @RT and deltas will be introduced in iteration 3). Line 3 defines vehicle speed as an integer number that ranges between 0 to 130. Line 4 defines man speed as a real number that ranges between 0.0 to 30.0. Line 5 means that "vehicle.identifier" receives only one of two possible values, "Hummer" or "AMX30" (these are just two of the available vehicles models). Line 6 means that "man.appearance_change" receives a floating point value normalized between 0.0 to 1.0, and finally, Line 7 defines an attribute "cost" that ranges between 0 to 1000 and is assigned to all the features.

R6. Establish the default values of configuration options. Developers want to create new video configuration files with the minimum of manual effort. For instance, they want to create partial configurations and want that the system completes the rest of the options automatically using predefined values assignments. However, developers did not have any explicit and organized way to establish the default values of configuration parameters.

Apart from supporting partial configurations, default values for attributes are key in automated analysis to define variables in automated problem solvers (e.g., CSPs).

→ **C29. Default Values.** Our solution for requirement R6 was to allow developers to establish a "default" value among a range of values and associate it to an attribute. For example, Line 3 defines that the vehicles' speed is 40, unless developers set other value in a con-

```
1 Attributes:
2 @NT string scene.comment
3 @RT int vehicle.speed [0..130] delta 5 default 40
4 real man.speed [0.0..30.0] delta 0.5 default 3.0
5 enum vehicle.identifier ["HummerH2","AMX30"]
6 real man.appearance_change [0.0 .. 1.0] delta 0.1 default 0.5
7 int *.cost [0 .. 1000] default 150
```

Listing 2: Feature attributes examples

figuration. In the MOTIV project, generation of video configurations is fully automated from the VM model. Therefore, default values are used during variables initialization using a CSP instance.

3.3. Iteration 3. Extra variability modeling

During this iteration we include most of the novel characteristics of VM that we call as "Extra variability" modeling (Characteristic C30 in Figure 3).

R7. Reduce domain values. Bounded attributes, reduce the number of possible values of an attribute, and therefore, the number of combinations of attributes values and features. However, even bounded attributes may have a value among an almost infinite range and it is necessary to specify which values are the most important. For example, the bounded attribute "real man.appearance_change [0.0..1.0]" includes many and too close values (e.g., 0.00011 and 0.00012) that are not differentiated by the human eye and it makes no sense to produce two different videos that vary only on those values. This requirement is related to the more general requirement R1 which addresses scalability and may be taken as a sub-requirement.

→ **C31. Delta values.** A solution for requirements R1 and R7 is a new construct called "delta". Each delta reduces the number of acceptable numeric values, therefore, "real man.appearance_change [0.0..1.0] delta 0.1" in the Listing 2, Line 6 will be interpreted as "enum man.appearance_change [0.0, 0.1, ... 0.9, 1.0]".

R8. Provide information about the specification and its elements. Developers wanted a more structured and standardized way to organize model information, such as version and name. Also, we saw necessary to provide information about the parameters. The goal of defining parameters is to agree on the meaning of each term. For example, the meaning of "signal_quality.chrominance_U_mean" was unknown for most of us while the attribute "man.appearance_change" could be interpreted in different ways, for example, the ability to change color, type, speed, trajectory, or the frequency in which he moves its legs or hands.

→ **C33-C34. Model information and descriptions.** Our solution to requirement R8 is the "Model information" and "Descriptions" blocks that add meaning to VM models.

```
1 @name "scene variability"
2 @version 1.0
3 @description "Part of the variability model
      related to the scene"
4 @author "DiverSE Team"
5 @email benoit.baudry@inria.fr
6 @organization "INRIA, Rennes, France"
7 @date "March, 2014"
```

Listing 3: Model information block example

Listing 3 shows an example of model information. Tags such as "@version", "@author" and "@date" are used to ease the classification of variability models in a repository or to implement simple but practical mechanisms of versioning control for VM models.

The Descriptions block contains a list of definitions of features, attributes or constraints expressed in natural language. Listing 4 show two examples, the first describes the feature "object" and the second describes the attribute "man.appearance_change".

VM helps to improve reference integrity using two rules: i) Only elements previously written in other blocks can be defined, ii) attributes must indicate the feature where they are contained using the containment designator ".", and iii) attributes that apply to more than one feature could use the wild-card "*" instead of the name of the feature (e.g., cost in Line 4).

R9. Deal with constraints. There is information about dependencies or incompatibilities between features that is difficult or impossible to express in the hierarchical decomposition of features captured in the Relationships block. Probably, the best-known examples of such constraints are "requires" and "excludes" constraints between features. Also, the addition of attributes and multi-features prompts the need to define more complex constraints.

→ **C50-C53. Constraints.** Our solution to the requirement R9 is the "Constraints" block that allows to write a wide spectrum of constraints between features and attributes. Listing 5 shows three examples of constraints, the first means that the selection of an urban

```
1 Descriptions:
2 feat objects is "..."
3 att objects.appearance_change is "..."
4 att *.cost is "the cost in miliseconds of adding a feature in a video sequence"
```

Listing 4: Descriptions block example

```
1 urban requires vehicle
2 countryside requires clonesOf man < 10
3 countryside -> vehicle.dust == vehicle.size
```

Listing 5: Constraint block example

background requires the selection of the feature vehicle. The second constraint means that the selection of a countryside background implies to include less than 10 different men along the video, and the last constraint specifies that the size of the dust cloud behind a vehicle in the countryside is equal the size of the vehicle.

Each constraint must be a valid expression that combines variables referencing features and attributes names, functions and operators. VM provides a list of operators whose syntax, semantics, and precedence are very similar to the Java language. For example, VM supports arithmetic (e.g., $*, +$), relational comparative (e.g., $>=, <$), equality ($==, !=$), logical operators (e.g., requires, $\&\&, ||$) and conditional (? :) operators.

VM also allows to use functions as parts of the constraints. As an example, lets imagine that we want to restrict the time taken to generate a video to be less than 2 hours measured in milliseconds. That constraint is written as "sum (*.cost) < 2*3600000". Functions can accept several parameters or may have equivalent operators (e.g., sum and +). VM supports logical (e.g., xor, or, and), arithmetic (e.g., sum, avg, max, min) and sets (e.g., clonesOf) functions.

R10. Deal with multiple ranges and priorities of values. Along the range of values of a particular physical measure there are too many possible values, but not all of them have the same importance to generate configurations. Therefore, values with less importance should not be considered in as many video configurations as in others. For example, domain experts advised that "luminance_mean" values above some levels will definitely make a video impossible to analyze, therefore, those values should not be included or included in less videos than other values. Therefore, it is necessary to make explicit the different allowed ranges of values as well as their importance in terms of its appearance frequency in video configurations.

→ **C35-C36. Multi-range and multi-delta values.** Our solution to requirement R10 and R1 is to extend the definition of attributes with information about the several allowed ranges of values. Each range can have a different delta that reduces the number of values considered in each range (a.k.a. *multi-delta*). Multi-deltas are necessary when not all values of an attribute are equally important for creating configurations. The rational is that in the delta should be small in ranges that are not too important, to consider just a few values in that range. For example, the attribute "luminance_deviation" has three ranges:
"real signal_quality.luminance_dev [0.0 .. 8.0] delta 1.0 [8.0 .. 32.0] delta 2.0 [32.0 .. 64.0] delta 4.0".
The first and last ranges are equally important for the domain experts and therefore, each one of them considers 8 values. The second range is a little bit more important than the others, therefore, it considers 12 values.

R11. Differentiate static and run-time variabilities. Some configuration options refer to changes in a video that will remain "as is" until the end of the video (static variability). However, there are other changes that are applied all along a video (run-time variability). The explicit distinction between these two kinds of variability is important to determine the binding time of each configuration option. This information is exploited to scope the spectrum of configurations, i.e., to only those that have static variability.

→ **C37. Run Time Annotation.** The solution for requirement R11 is to apply an annotation to indicate the binding time of each element. For the video domain we only consider run time but there are other binding times in other domains, e.g., load time, and link time. Runtime features and attributes will be those whose values can change during the execution time of the video. To decide if an element varies at runtime or not depends only on the domain experts.

A runtime feature or attribute is represented using the tag "@RT" or "@RunTime" before its name. For example, the "background" feature would be considered as runtime if it can be changed during the execution of a video: "@RunTime background". Listing 2, Line 3 shows that the speed of vehicles varies during the video between 0 and 130 km/h.

These annotations also support scalability (R1) as they can reduce the amount of variables in a constraint satisfaction problem (CSP) for generating a pair-wise,

so we do not introduce such variables in the CSP but consider them into the final configurations.

R12. Benefit automated support. VM models should be read by a processing tool to provide automated analysis operations (e.g., to check that the model is valid or return pair-wise covering), to generate Lua configurations files, and to control the generation of the videos related to those configurations. It is important to note that there are too many possible combinations of features and attribute values. Therefore, part of the challenge is to determine the criteria to define which are the most important combinations to generate. The objective is to find constructs in the language that have a positive effect in the performance of the automated analysis operations (e.g., to retrieve significant pair-wise coverage configurations, or prune not valid but too similar configurations).

→ **C38-C39. Not translatable and not decision annotations.** A solution for requirements R12 and R1 is the use of annotations that can be attached to features or attributes to add information about how to deal with them when interpreting a VM model.

The *not translatable annotation* means that a feature or attribute contains information that should not be considered when creating a CSP based on the VM model. A not translatable feature or attribute is represented by "@NT" or "@NotTranslatable" before its name. For example, in Line 2 of Listing 2 we considered the attribute "comment" of a scene as not translatable. This is also useful for pair-wise calculations using CSP because it reduces the numbers of variables.

The decision variables in a CSP solver are the ones that need to be *determined* to solve the problem. Typically, decision variables represent concrete features or attributes that are visible in a system. However, when using feature models and depending on the domain, there are some variables that do not really form part of the problem solution (e.g., abstract features or concepts). For example, in the MOTIV project, only the variables representing video attributes are part of the solution, therefore, features were taken as not decision variables. This is, a solution is distinguished from another depending on the value of the attributes. The *not decision annotation* means that a feature or attribute can have any value when translated to a CSP. A not decision feature or attribute is represented using the tag "@ND" or "@NotDecision" before its name. This reduces the complexity of the CSP calculation because ND tagged elements will exist in the problem but *the solver does not traverse all possible domain values*.

→ **C40. Objective functions.** It is necessary a translation between VM models and CSP solvers [4] to address requirement R12. With this translation, we can execute automated analysis operations such as *NumberOfProducts*, *ValidConfiguration*, or meeting different user criteria (a.k.a. *objectives*). Moreover, we can perform some testing operations, for example, the pair-wise operation considering attributes' values [5]. Listing 6 shows one objective called "generate_low_cost_configurations" in the "Objectives" block. This objective is defined as the minimization of the total "cost" of the features, which is calculated by the sum of the values of the attributes "cost".

```
1 objective generate_low_cost_configurations {
2   min (sum (*.cost))
3 }
```

Listing 6: Objective function example

R13. Support to define configurations. The VM tool should allow to read and to write combinations of features and attributes' values that characterize specific video configurations.

→ **C41-C49. Configurations.** Our solution for requirement R13 is the "Configurations" block. Configurations are used to characterize specific systems in terms of a combination of features and attributes' values.

In our project, the VM tool generates Lua configuration files directly from a VM model (e.g., the pair-wise covering configurations), thus, it is not necessary to manually write configurations in VM. However, developers use VM to write extra video configurations. These configurations can be next converted to Lua configuration files and read by the video generator using the VM tool.

VM provides two different ways to configure: i) boolean valuation, and ii) extended valuation.

→ **C43. Boolean Valuation.** This is the simplest type of valuation used to classify features as either selected (*activated*) or unselected (not *activated*). In VM, developers write the feature name inside a configuration to indicate that the feature is selected; otherwise, that feature will be considered not selected. Also, it is possible to use the operator "!" before a feature name to express not selection.

The configurations in Listing 7 are two simple, equivalent and valid configurations. They configure a video with a countryside background and without any objects appearing in the scene. There are several Boolean valuations in that configuration. For example, in Line 6 "countryside" is activated, while in Line 7, "objects" is deactivated. Mandatory features such as "scene" and "background" can be omitted (Lines 9 to 12) to make

```
configuration simple {
 scene
 scene.comment = "an empty countryside scene"
 background
  ! urban
  countryside
  ! objects
}
configuration simpleNotVerbose{
 scene.comment = "an empty countryside scene"
 countryside
}
```

Listing 7: Two simple and equivalent configuration of an scene

```
configuration advancedExtenedVal {
  feature-value pairs for attribute: cost {
    urban - 400,
    countryside - 250,
    objects - 1000
  //more pairs feature - value
  }
  attribute-value pairs for feature:
      signal_quality {
    luminance_mean = 72.55,
    luminance_dev = 48,
    chrominance_U_mean = 128
  //more rows signal_quality.attribute =
      value
  }
  attribute-value pairs for clone feature:
      vehicle clone: FirstAuto {
    identifier = "Hummer",
    speed = 50
  //more rows for FirstAuto.attribute = value
  }
}
configuration advancedExtendedValNotVerbose {
  cost {/*...*/}
  signal_quality {/*...*/}
  vehicle [FirstAuto] {/*...*/}
  //more rows for FirstAuto.attribute = value
}
```

Listing 8: Partial configurations illustrating extended valuation

configurations less verbose.

→ **C44. Extended Valuation.** This type of valuation is used with attributes, muti-features and cardinality-based groups.

- *C45. Basic*. For feature' attributes there is a very intuitive basic extended valuation based on the same patterns used to give values to attributes (Section 3.2). For example, Line 3 employs this basic extended valuation to assign a comment to the scene.

- *C46-49. Advanced*. This type of valuation is used to summarize and modularize otherwise repetitive and possibly, long and scattered valuations along each configuration. Listing 8 illustrates two different ways to write configurations using advanced valuation. Lines 1-19 show a more readable and suitable syntax to new users and Lines 20-24 show a shorter syntax suitable to experienced users.

Lines 2-7 address valuation grouped by an attribute to modularize in a block all the assignments feature-value. This way contrasts with the basic valuation that requires to follow the pattern "feature.cost = value" in many different places of the configuration.

Lines 8-13 address valuation grouped by feature and follows the same concept than valuation grouped by attribute. There is a difference between the previous two valuations in terms of the operators "-" and . Valuation by feature employs the "=" operator because it directly assigns a value to each attribute. In contrast, valuation by attribute has an indirect valuation to its attributes and therefore, it should use a different operator "-'. Finally, Lines 14-18 address valuation grouped by copy of a multi-feature and follow the same pattern than valuation grouped by feature.

4. Discussion and Evaluation

4.1. Practical Considerations

We discuss some practical considerations of applying VM. The goal is to help external readers or practitioners to evaluate if our particular experience and language are good for their purposes. Most of these considerations were proposed by Savolainen *et al.* [9] which are based on practical experiences and research carried out in cooperation with several companies – as also happened in our case.

Cost-Benefit. *"What is the optimal model in terms of cost-benefit when taking into account construction, usage, and maintenance?"* [9].

Construction can be divided in the creation of the language and the creation of the model of video variability: *Language infrastructure construction.* Creating a language, editor and interpreter is not for free since there are many tasks to support, such as parsing, autocompletion, syntax highlighting, etc. However, we discovered that new frameworks for language development make these tasks less complex and or even fully automated. For instance, we used Xtex [5] to generate the VM editor and parser, based only on the VM grammar definition. Using Xtext, we expended about one hour to create a working VM editor for the first iteration of VM, three days for the second iteration, and about four days for the third iteration. Admittedly, the most difficult part was to understand how to model and interpret nested expressions and operators precedence in the constraints block.

[5] http://www.eclipse.org/Xtext/

An extra effort for us was the connection with the Lua code and the configuration files for the VM model to be aligned with the schema of the configuration file exploited by the video generator. We needed technical exchanges (by emails), beyond meetings with video. This part took about one week, as it was a common effort between the creators of the language infrastructure and the creators of the video generator.

Model creation. This task was the one that took more time; it required to understand the domain, the requirements, and to discuss with video experts. We produced six different versions of the video variability model during a period of about nine months. These versions were made after four large meetings with all the project partners (these meetings focused on different topics apart from variability modeling, including administrative issues and technical issues in the video analysis domain), and two individual meetings with the main developer of the video generator.

Usage. Just a sketch of a feature model would be enough for communication; however, our project justified the construction of a language and tool to support not only communication but also enable video generation through scalable automated analysis.

Writing a valid configuration file of a video manually take around three minutes for the video generator expert. Therefore, to create an initial set of only 500 videos would take 25 hours $((500 * 3)/60)$, which does not consider the time to correct mistakes of creating invalid configurations. VM supports the process to generate not only 500 but also thousands of valid and diverse video configurations that guarantee some objectives (e.g., pair-wise coverage) in seconds. Section 4.2 complements the cost-benefit point with an evaluation of the benefits in terms of performance scalability of the new constructs proposed by VM.

Maintenance. In our particular experience, we did not experience significant maintenance costs associated to changes to the language grammar or the video variability model written in VM. On one side, Xtext helped to us maintain the language infrastructure code (e.g., parser, editor, etc.) by separating generated code from manual code. On the other side, we did not experienced major problems to update our video variability model since the video domain is stable and the only changes that we applied were increments in the specification.

As a conclusion for the cost-benefit practical consideration, we say that the costs of constructing, using and maintaining VM models are low compared with the benefits of producing automatically suitable videos to test complex video analyzers. Similar achievements were impossible before the introduction of variability techniques.

Stakeholders. *"Who puts effort into and who gains the benefits of the model? What knowledge about feature modeling methods in general and the product line in question do the stakeholders have?"* [9].

VM was developed mainly by a team composed of two people (a doctoral and postdoctoral researcher) and one lecturer at Inria, which knew about product lines, and some feature modeling methods. This team created the language infrastructure, implemented a translation from VM to a CSP (presented on a previous work [5]), and work to connect the VM tool with the video generator.

The main video generator developer is also a video expert that provided feedback for improving the VM design. In addition, he wrote an initial and not exhaustive description of the important aspects that may be varied in a video that were important to test a predefined set of video algorithms. Based on the description, the development team wrote the first version of the VM model and used that version to communicate with the rest of the partners in the following meetings. Stakeholders from the DGA provided comments that were addressed in the following iterations and model versions. However, the role of the members of the DGA was mainly to review that the video sequences synthesized were realistic.

Taking into account the variety of stakeholders, we took the decision of dividing the VM language by blocks, each one addressing a different concern. Video experts without too much technical expertise can focus on concerns described in the relationships, model information, definitions, and objectives blocks. Developers and video experts with a programming background can focus on adding annotations, constraints, deltas, or further defining the objectives and attributes blocks.

Correspondance. *"What elements of the product line does the feature model correspond to?"* [9].

1-to-1 mappings between features in the problem space and their realizations in the solution space ease their co-evolution. For example, many features in the video domain VM model have a 1-to-1 relationship with code modules that implemented the video generator. In a similar way, feature attributes tend to match input parameters of Lua functions.

Using 1-to-1 mappings is not a strict rule. In fact, we also modeled features that are not mapped to any specific module to group other features or attributes. For example, the feature "objects" does not map directly to any module, but helped to group conceptually the "vehicles" and "man" features that have concrete mappings to the code.

One important highlight regarding correspondence was that we decided not to use VM to model all possible variability in a video sequence. In particular, we decided not to model or provide constructs to determine the time and order in which events happen in a video sequence or the path of moving vehicles and people in an scene. Our partners already had a way to orchestrate events and to create and manage paths in predefined backgrounds. However, we are considering the integration of those aspects in future versions of VM.

Constraints. *"What do the constraints represent?"* [9]

VM addresses the challenge of managing and representing constraints through a set of functions and operations over features, attributes and sets of features (e.g., "ClonesOf"). Constraints are also important for *specializing* the VM model to specific testing scenarios. For instance, experts want to synthesize only videos with a specific background (such as desert or urban) or luminance; some values are thus fixed, but the other features or attributes are still subject to variations.

Notation. *"What constructs and representation should different stakeholders use?"* [9]

The VM language provides a *textual* notation for expressing variability. There are two major reasons. First, some of the partners are developers of video algorithms and are already familiar with textual content. In contrast to diagrammatic languages, participants continue to use well-established efficient tools in the industry such as code editors. Second, numerous attributes, meta-information, and cross-tree constraints have been specified; by construction they are textual information.

4.2. Reasoning Scalability

To validate some of the benefits introduced by VM, we now evaluate the automated analysis operation that retrieves a pair-wise coverage [10, 5]. The operation takes as input a VM model and generates some configurations (i.e., values for features and attributes) conforming to the constraints. Our goal is to study the effect of (1) @ND (for "not decidable") and (2) deltas (for varying the increment of a domain) on the performance of the operation. We expect to decrease the amount of time using meta-information (@ND and deltas).

Data. For the two experiments, we took the complete VM model of the MOTIV project as input. This model contains: i) 18 features containing different amount of attributes; and ii) a total of 84 attributes with ranges going from 0 to 120000. The size of the sum of all ranges represents 2161711 integer values. This model represents up to $2,0484 \cdot 10^{18}$ configurations. A key characteristic of this model is that most of the variability it represented as numeric attributes related to physical properties.

Experimental settings. The two experiments were executed in a Dell computer running an Intel i7 M 620 at 2.67GHz and 4 GBs of RAM. The operating system was Ubuntu 12.04, with a 1.7 open-JDK virtual machine. The implementation of the pair-wise operation internally relies on the Choco 2 solver.

Evaluating the effects of @ND. In the first experiment, we created ten groups, each one containing ten copies of the original model. Each group has a percentage of @ND tags, which went from 0 to 100 percent. The tags in each group were assigned to the attributes randomly. We report on the average time required by each group. The experiment hypothesis is that the use of @ND tags improves the performance of the pair-wise operation in the context of the MOTIV project.

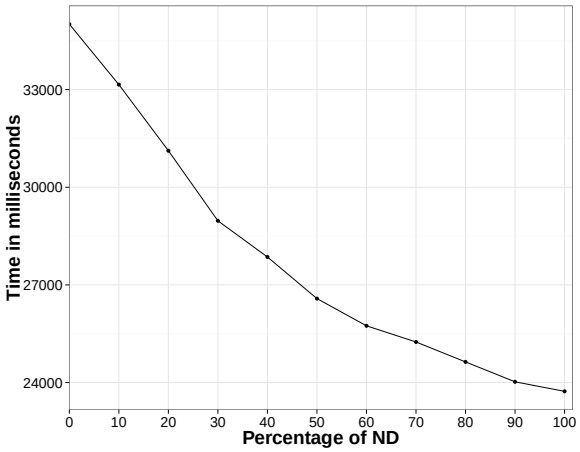

Figure 4: Time for obtaining a pair-wise coverage depending on the percentage of ND for the MOTIV feature model.

Figure 4 shows the results. The time varies around 10 seconds between the models containing 100% of @ND tags and the models with no tags. It represents an improvement of around 30% in the execution time. The improvement is significant. But the testing operation can still scale in a reasonable amount of time without @ND tags. At this step of the research, we thus cannot state that @ND tags are mandatory constructs in MOTIV for scaling but it matters and helps improve the time needed. This is, we suspect that it would be more useful for larger models, however, this experimentation is keep as future work.

Yet, we conjuncture that the VM model of the MOTIV project will grow in complexity and handle more attributes and features in the future. We expect to gain even more time reduction in future releases based on

the Figure 4 tendency. Another argument for @ND tags is that we generate only *relevant* configurations. Note that ND tags helps the solver to focus only in decision variables but this does not reduce the complexity of the problem in terms of variable domains.

Evaluating the effect of deltas. In the second experiment, we measured the impact of deltas usage in the pair-wise operation. Specifically, we compared the time required to execute the operation with and without deltas. When no deltas are specified, we consider an increment of "1" for integer ranges. We executed the testing operation with the deltas provided by our industrial partners. In this experiment no tags were used so the improvement of each language construct can be evaluated independently.

We first observed that the variable domains were reduced in 3400 integer units in the CSP when using the deltas optimization. The pair-wise operation without deltas took 38020 milliseconds. When enabling the deltas usage, the solver took 34772 milliseconds. This represent an improvement of 4 seconds. This experiment shows that the constructs of the language improves the scalability.

The improvement is noticeable but does not impose the presence of deltas at this step of the research. As for @ND tags, the potential of deltas may be more apparent with the growing complexity of the VM model. It also lies in the control of the configuration generation: practitioners can fine-tune the way values of attributes vary. We can also envision the combined use of deltas and @ND to reduce the amount of time and generation of relevant testable configurations.

Scalability improvements in random models. The major bias of our two first experiments is the population validity. Therefore, to extend our conclusions to models having different topologies and attributes nature, we performed the same operation over a set of models generated by Betty[11].

Figure 5 shows the time required by the operation depending in the amount of cross-tree constraints and the percentage of non decision tagged attributes. It is remarkable that the time required by the operation is reduced almost in the same percentage as the annotations introduced. Moreover, when models are big enough (e.g. 100 features) the time reduction is more than 30 minutes. This points out that the introduction of extra information is handy for providing better results when implementing automated analysis tools.

Threats to Validity. The main threats to validity to this experiment is related to the nature of the models randomly generated. While we used the Thum approach that tries to mimic real feature models. It is possible that

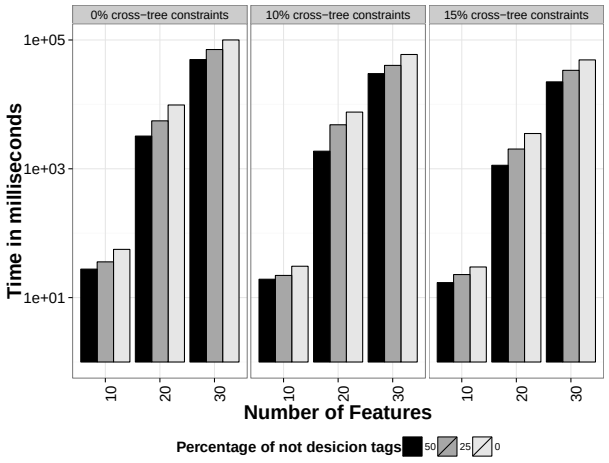

Figure 5: Time for obtaining a pair-wise coverage depending on the percentage of constraints and extra-information.

they do not cover all the properties of real models. Also, we only measured this improvement using one operation. A more extensive analysis will be done in future work, trying to compare the VM benefits within the set of 30 existing analysis operations [4]

4.3. Comparison with Existing Solutions

Numerous languages, being textual or graphical, have been designed to model variability. For instance, feature models have become more and more sophisticated since 1990 and their dialects have been detailed in comprehensive surveys, for example, by Schobbens *et al.* [12], Benavides *et al.* [4] and Eichelberger and Schmid [13]. Boolean constructs of feature models (as supported by FODA [14], FDL [15], XSFM [16], or Velvet [17]) are useful in the video domain, but not sufficient. New dialects (e.g., FAMA [18], TVL [19] and Clafer [20]) have emerged to overcome the expressiveness limitations of feature models, for instance, to deal with attributes or multi-features (R3 and R5).

Table 1 summarizes the comparison of VM with some representative languages in terms of the requirements that they address as a goal. The *key difference between VM and the other approaches* is the use of meta-information associated to features or attributes. For example, VM users can include: i) default values (R6), ii) deltas (R7), iii) elements definitions (R8), iv) multi-ranges and priorities –multi-deltas (R10), v) meta-information annotations such as "not translatable", "not decidable" (R12), and "runtime" (R11), and vi) objective functions (R12). As reported in Section 3, our industrial experience strongly motivates the introduction

of these new constructs. We also show the importance of the constructs in terms of reasoning scalability (see Section 4).

Currently, we are evaluating the applicability of VM as a generic attributed feature modeling language such as SXFM, TVL or FAMA. In the VM repository `https://github.com/ViViD-DiverSE/VM-Source` we have VM models that support the point that VM is expressive enough to encode existing boolean and attributed feature models found in the literature and public repositories (e.g., mobile media [26] and the SPLOT repository `www.splot-research.org`).

Berger *et al.* [7] studied the modeling of variability in the operating system domain (Linux, eCos, and FreeBSD are the subjects of the study). They showed that well-researched concepts of FODA feature models, comprising Boolean (optional) features, a hierarchy, group and cross-tree constraints, are used. They also identified domain-specific concepts beyond FODA feature models, such as: visibility conditions, derived features, derived defaults, and binding modes.

Dumitrescu *et al.* [6, 21] reported on their experience in an automotive model based systems engineering. Mussbacher *et al.* [22] propose an extension of the Aspect-oriented User Requirements Notation (AoURN) to support variability modeling. The outcome is a holistic reasoning framework based on goal modeling, feature modeling, and specification of scenarios. The framework has been applied on Via Verde, a real-world product family that aims to simplify the payment processes.

Requirement / Approach	FODA [14]	FDL [15]	SXFM [16]	Clafer [20]	TVL [19]	FAMA [18]	VM
R1. Be scalable	○	○	●	●	◑	●	●
R2. Organize conf. options	◑	●	●	●	●	●	●
R3. Limit n. of configurable copies	○	○	○	●	●	○	●
R4. Limit n. of options from a group	○	○	●	●	●	●	●
R5. Represent diverse conf. options	○	○	○	●	●	●	●
R6. Establish default values	○	○	○	○	○	●	●
R7. Reduce domain values.	○	○	○	○	○	○	●
R8. Provide info. of spec. and elem.	○	○	●	○	○	○	●
R9. Deal with constraints	◑	◑	◑	●	●	◑	●
R10. Deal with multi-ranges and pr.	○	○	○	○	○	◑	●
R11. Diff. static and run-time	○	○	○	○	○	○	●
R12. Benefit automated support	○	○	◑	◑	◑	◑	●
R13. Support to define configs.	○	○	○	◑	○	○	●

●addressed as goal, ◑addressed but with restrictions, ○not regarded as goal

Table 1: Summary of comparison between languages

Variability in industry. A recent survey reported that feature modeling is by far the most popular notation used in industry [8]. However no details are given on the specific language constructs used for modeling variability requirements. The industrial survey shows that pure::variants and GEARS are the most industrial tools used to model variability. They provide support for feature models but some adaptations are needed to cover all the requirements we faced in our industrial project. The most important is to provide reasoning support for extra variability and a mechanism to discretize multiple ranges of values defined in continuous domains.

Interestingly, a variety of notations is used in industry – most of industrial practitioners rely on several notations [8]. Studies of variability also show that modeling languages in open source systems all contained domain-specific, or even project-specific language constructs [7]. Our experience-report also highlights specific needs when modeling variability in the video domain. It questions the existence of a one-size-fits-all solution applicable in any industry without specific adaptations.

The Common Variability Language (CVL) (`http://www.omgwiki.org/variability/doku.php`), a recent proposal for OMG's standard, describes a comprehensive process for modeling software product lines. CVL includes the description of a variability abstraction model (VAM) that conceptually corresponds to a feature model with attributes and multi-features. The language VM is compatible with the VAM of CVL, but also comes with specific constructs (e.g., meta-information) and an associated reasoning support.

Domain-specific profiles and languages. One solution to address specific needs when modeling variability is the use of modeling profiles. These are particular ways to give a host language the feel of a domain-specific language. For example, Hofman *et al.* [23] extended UML Activities to represent and relate different kinds of "Features". We avoided to follow this approach as VM is intended to change independently of any particular host language.

Another alternative is the development of domain-specific languages (DSLs) [24, 25]. We want to highlight the fact that VM provides no specific construct to the domain of video (e.g., the language construct "scene" or the keyword "illumination"). Therefore, we can consider VM as a domain-specific language for variability modeling in general that offers adequate variability constructs *for* the video domain.

5. Threats To Validity

Two major external threats to the evaluation of scalability performance (see Section 4.2) are: *(1) population validity*, i.e., the model used in the experiments represents only one concrete instance of the problem. We consider that the feature model is realistic since several experts were involved in its design. Moreover, the result is not a contemplative model and has proved to be effective to synthesize videos variants. However, it is possible that the feature model does not reflect properly the same structure as other realistic models. It is also possible that the future evolution of the model changes its inherent complexity and influences the results. We plan to repeat the experiments with random models to claim that our techniques are applicable to other models. *(2) ecological validity*: analyses were executed 10-times and we report on averages to minimize the impact of third-party threads in the time being measured.

Another threat is that we evaluate the practical considerations (see Section 4.1) while being active participants of the project. To mitigate this threat, we structure the criteria according to an external evaluation framework [9]. Generalization of the observations of Section 4.1 (e.g., for the VM language or for the variability methodology) would require additional case studies and is premature at this stage of the research. The goal of Section 4.1 is thus more modest; we want to report on our specific industrial experience in a structured and disciplined way.

There may be variability languages we do not consider when comparing VM to existing solutions (see Section 4.3). To mitigate this threat, we consider recent comprehensive surveys [8, 13].

6. Related Works

Section 4 discusses state-of-the-art variability languages w.r.t. specific requirements we identified in an industrial project. In this section, we review other related works.

Variability and video domain. There is a plethora of work related to the domain of computer vision (and by extension to video analysis). Many video algorithms have been designed and benchmarked, and form the basis of many crucial applications of modern society. An original goal of the industrial project is to synthesize variants of videos with the intention of testing video algorithms. To the best of our knowledge, no generative approach, guided by a high-level variability specification and supported by automated techniques, has been proposed or developed in this domain.

Moisan *et al.* [27] and Acher *et al.* [28] proposed support to model the technical variability of video algorithms. The objective was to systematize the deployment of a customized video surveillance processing chain, suited to specific tasks (e.g., tracking of persons in an airport) and reconfigurable at runtime [27, 28]. In our industrial project, the goal and requirements are radically different: the challenge is to model the variability of videos – not of the algorithms. This key difference leaded us to design and use advanced variability language constructs. Acher *et al.* used only Boolean constructs for modeling variability [28].

Variability and reasoning support. Benavides *et al.* [4] made a survey of more than 20 years of automated analysis of feature models. Most of the reasoning operations apply on FODA feature models, i.e., with Boolean constructs. It called for more research devoted to the formalization, performance comparison and support of so-called extended feature models. Since then, advances have been made to support attributes and multi-features (also called *clone enabled features*), relying on either CSP solvers, BDD, SAT, or SMT solvers (e.g., see [29, 3]). An original and crucial aspect of our work is that we exploit meta-information over features and attributes when encoding VM models and generating test configurations. It has two merits: i) reducing the complexity of the constraint problem fed to the CSP solver, and ii) generating test configurations that contain only features and attributes *relevant* for specific video analysis scenarios.

In [5], we developed testing analysis operations operating over VM models (i.e., attributed feature models). Our previous work [5] focused on the testing operation. In this paper we comprehensively (1) describe the variability language and (2) report on our industrial experience. The effect of deltas and @ND ("not decision", see Section 4.2) on scalability performance had not been evaluated either.

7. Conclusions & Lessons Learned

In an industrial project, we faced the original challenge of synthesizing video variants. The goal is to test competing vision algorithms and thus determine what solutions are likely to fail or excel in specific settings. It is crucial for the partners of the project – being providers or consumers of the algorithms – to collect a comprehensive and suitable input of videos. The current practice, based on the manual elaboration of videos, is very costly in resources and cannot cover the diversity of targeted video analysis scenarios. We introduce a

generative approach and we address the following problem: What are the variability requirements in the video domain? How to capture what can vary within a video and then automate the synthesis of variants? This paper reported how specific requirements, encountered in the project and in the video domain, have shaped the design of a textual variability language (VM) with advanced constructs and reasoning support. We learned the following important lessons from our industrial experience:

1. Basic variability mechanisms *à la* FODA – Boolean (optional) features, hierarchy, group and cross-tree constraints – are useful but not enough;
2. Attributes and multi-features are of prior importance;
3. Meta-information is relevant for (1) performing efficient computer-aided analysis of VM models; (2) customizing the generation of testable configurations (e.g., to focus on specific attributes of features);
4. We detail the additional specific constructs (e.g. deltas, binding mode) we have added. Different iterations were needed for connecting VM to the video generator developed by the industrial partners and thus realizing a comprehensive solution [30];
5. Experts have reviewed 20+ variants we synthesized and judged that the video sequences are realistic. We are at the step of launching a very large-scale testing campaign over thousands of realistic variants – something clearly impossible at the beginning of the project (i.e., without variability support).

The point of the paper is not to present yet another variability language. We rather want to highlight the specific requirements we faced throughout the project, in the video domain, leading to the design and use of existing (or novel) variability constructs. Our experience, as others [6, 7, 8], question the existence of a one-size-fits-all variability solution applicable in any industry. Yet some common needs for modelling variability are becoming apparent (e.g., support for attributes and multi-features [3, 20, 29]).

Variability is gaining momentum in an increasing amount of domains and applications. The synthesis of video variants is an additional illustration. It perhaps explains the diversity of existing techniques, practices, tools, and languages for capturing variability requirements. We hope our experience report can further the understanding of variability and motivate innovative research for supporting variability practitioners.

Acknowledgements

This works was financed by the project MOTIV of the Direction Générale de l'Armement (DGA) - Ministère de la Défense, France. We also give thanks to Pierre Romenteau from InPixal (Rennes, France) for his feedback and his joint development for synthesizing video variants.

References

[1] M. Svahnberg, J. van Gurp, J. Bosch, A taxonomy of variability realization techniques: Research articles, Softw. Pract. Exper. 35 (8) (2005) 705–754. doi:http://dx.doi.org/10.1002/spe.v35:8.

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

[3] M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, Beyond boolean product-line model checking: dealing with feature attributes and multi-features, in: ICSE'13, 2013, pp. 472–481.

[4] D. Benavides, S. Segura, A. R. Cortés, Automated analysis of feature models 20 years later: A literature review, Inf. Syst. 35 (6) (2010) 615–636.

[5] J. A. Galindo, M. Alférez, M. Acher, B. Baudry, D. Benavides, A variability-based testing approach for synthesizing video sequences, in: ISSTA, To appear in 2014.

[6] C. Dumitrescu, R. Mazo, C. Salinesi, A. Dauron, Bridging the gap between product lines and systems engineering: an experience in variability management for automotive model based systems engineering, in: T. Kishi, S. Jarzabek, S. Gnesi (Eds.), SPLC, ACM, 2013, pp. 254–263.

[7] T. Berger, S. She, R. Lotufo, A. Wasowski, K. Czarnecki, A study of variability models and languages in the systems software domain, IEEE Trans. Software Eng. 39 (12) (2013) 1611–1640.

[8] Berger, Thorsten and Rublack, Ralf and Nair, Divya and Atlee, Joanne M. and Becker, Martin and Czarnecki, Krzysztof and Wasowski, Andrzej, A survey of variability modeling in industrial practice, in: VaMoS'13, ACM, 2013.

[9] J. Savolainen, M. Raatikainen, T. Männistö, Eight practical considerations in applying feature modeling for product lines, in: ICSR, 2011, pp. 192–206.

[10] C. Nie, H. Leung, A survey of combinatorial testing, ACM Comput. Surv. 43 (2) (2011) 11.

[11] S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, A. R. Cortés, Betty: benchmarking and testing on the automated analysis of feature models, in: VaMoS, 2012, pp. 63–71.

[12] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, Feature diagrams: A survey and a formal semantics, in: RE, 2006, pp. 136–145.

[13] H. Eichelberger, K. Schmid, A systematic analysis of textual variability modeling languages, in: SPLC, 2013, pp. 12–21.

[14] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented domain analysis (foda) feasibility study, Tech. rep., DTIC Document (1990).

[15] A. van Deursen, P. Klint, Domain-specific language design requires feature descriptions, Journal of Computing and Information Technology 10 (1) (2002) 1–17.

[16] M. Mendonça, M. Branco, D. D. Cowan, S.p.l.o.t.: software product lines online tools, in: OOPSLA Companion, 2009, pp. 761–762.

[17] M. Rosenmüller, N. Siegmund, T. Thüm, G. Saake, Multi-dimensional variability modeling, in: VaMoS, 2011, pp. 11–20.

[18] D. Benavides, P. Trinidad, A. R. Cortés, S. Segura, Fama, in: Systems and Software Variability Management, 2013, pp. 163–171.

[19] A. Classen, Q. Boucher, P. Heymans, A text-based approach to feature modelling: Syntax and semantics of tvl, Sci. Comput. Program. 76 (12) (2011) 1130–1143.

[20] K. Bak, K. Czarnecki, A. Wasowski, Feature and meta-models in clafer: Mixed, specialized, and coupled, in: SLE, 2010, pp. 102–122.

[21] C. Dumitrescu, P. Tessier, C. Salinesi, S. Gérard, A. Dauron, R. Mazo, Capturing variability in model based systems engineering, in: M. Aiguier, F. Boulanger, D. Krob, C. Marchal (Eds.), CSDM, Springer, 2013, pp. 125–139.

[22] G. Mussbacher, J. Araújo, A. Moreira, D. Amyot, Aourn-based modeling and analysis of software product lines, Software Quality Journal 20 (3-4) (2012) 645–687.

[23] P. Hofman, T. Stenzel, T. Pohley, M. Kircher, A. Bermann, Domain specific feature modeling for software product lines, in: SPLC (1), 2012, pp. 229–238.

[24] J. Gray, K. Fisher, C. Consel, G. Karsai, M. Mernik, J.-P. Tolvanen, Dsls: the good, the bad, and the ugly, in: G. E. Harris (Ed.), OOPSLA Companion, ACM, 2008, pp. 791–794.

[25] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, G. Wachsmuth, DSL Engineering - Designing, Implementing and Using Domain-Specific Languages, dslbook.org, 2013.

[26] A. S. Karatas, H. Oguztüzün, A. H. Dogru, From extended feature models to constraint logic programming, Sci. Comput. Program. 78 (12) (2013) 2295–2312.

[27] S. Moisan, J.-P. Rigault, M. Acher, P. Collet, P. Lahire, Run time adaptation of video-surveillance systems: A software modeling approach, in: International Conference on Computer Vision Systems (ICVS'11), 2011, pp. 203–212.

[28] M. Acher, P. Collet, P. Lahire, S. Moisan, J.-P. Rigault, Modeling variability from requirements to runtime, in: ICECCS, 2011, pp. 77–86.

[29] W. Zhang, H. Yan, H. Zhao, Z. Jin, A bdd-based approach to verifying clone-enabled feature models' constraints and customization, in: ICSR, 2008, pp. 186–199.

[30] M. Acher, M. Alférez, J. A. Galindo, P. Romenteau, B. Baudry, ViViD: A Variability-Based Tool for Synthesizing Video Sequences, in: SPLC'14 (tool demonstration track), 2014.