Learning-Contextual Variability Models

Paul Temple, Mathieu Acher, Jean-Marc Jézéquel, Olivier Barais

Learning Contextual Variability Models

Paul Temple,
Mathieu Acher,
Jean-Marc Jézéquel,
Olivier Barais
(Univ Rennes, IRISA, France)

“Modeling how contextual factors relate to the configuration space of a software system is most of the time a manual and error-prone task, highly dependent on expert knowledge. Machine learning techniques have the potential to automatically predict what are the acceptable software configurations of a given context. The key idea is to execute and observe a sample of software configurations within a sample of contexts, and then learn what factors of the context are likely to discard or activate some features of the software. As a result, software developers and product managers can automatically extract the rules that specialize highly-configurable systems for operating on specific contexts.”

Introduction

Highly configurable systems emerged in recent years to address the fact that there is no one-size-fits-all solution capable of meeting all possible expectations of users in every possible context. Thus, most modern software systems, including mobile apps, web servers, operating systems, or computer vision systems can nowadays be configured at compile time or at run-time to deliver the “right” functionality and performance for a given context of deployment. Conversely, some configurations could be very badly suited to some contexts: for instance, the activation of some features in mobile apps could dramatically increase energy consumption on specific mobile devices and operating systems.

The challenge we address in this article is to automatically map *software configurations* to *specific contexts* (and vice-versa). We consider that the context of a software system is itself a configurable entity. It is constituted of different concerns: execution environment (hardware, operating system, etc.), kinds of inputs to process, goals and performance to meet (execution time, quality of the result, etc.). Other factors such as country regulations or marketing strategies can also be part of this context and have an (indirect) influence on the software [1]. From this mapping, we expect that given a context configuration, there is at least one corresponding software configuration. Conversely we also want that for each possible software configuration, there is at least one corresponding context configuration. If that is not the case, we could prevent this particular software configuration to be selectable from the beginning.

Modeling variability

Since there are numerous features and complex interactions among these features, leading to a combinatorial explosion of possible contexts and software configurations, experts typically elaborate a variability model that formalizes how features can be combined. Variability information (e.g., optionality) and cross-tree constraints over features define the set of *valid* configurations. Numerical information can also be associated to features.

Let us take the example of a tracking vision system build on top of OpenCV (see http://opencv.org/).
A variability model is depicted in Figure 1. On the left-hand side, context configurations involve features such as Camera, Video, Position, Light Source, etc. Some features are mandatory (e.g., Camera or Video), optional (specific climate conditions), mutually exclusive (Fog or Heat haze), and some values are numeric (Vibrations or Noise level).
On the right-hand side, we model (a subset of) OpenCV software variability. For example, Confidence is a parameter of the Detect function. It can be tuned for internally influencing the results of subsequent computations.
In the example of Figure 1, the two variability models are separated and aggregated as in [1] since several contexts are involved. As suggested in [6], we could also elaborate a single variability model that integrates both software and context features. It would simplify the variability model and reduce the number of dependencies among features.

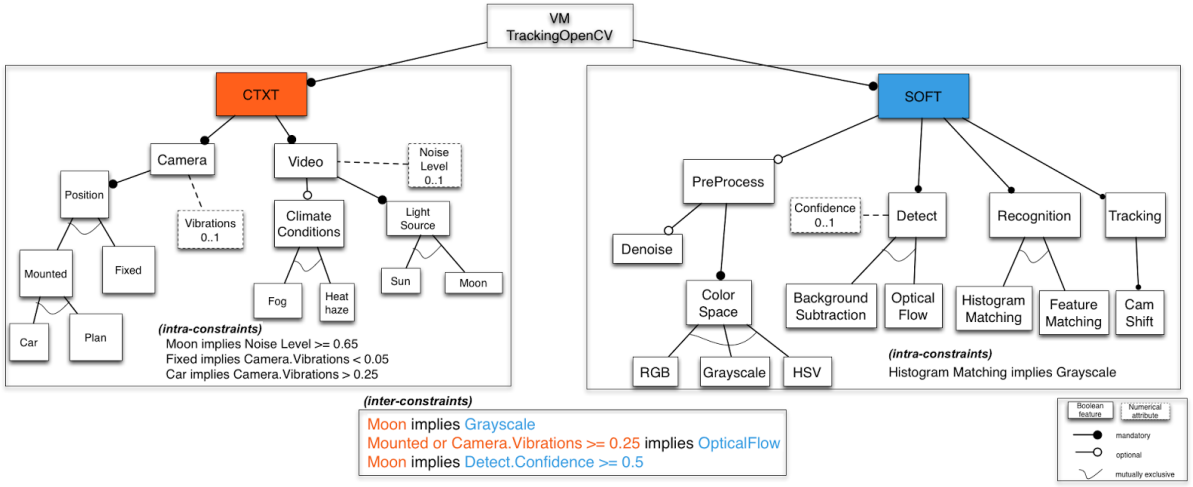

Figure 1: Modeling context/software configurations, intra-constraints inside software (or inside context), and inter-constraints between the context and software. The goal of our method is to discover such inter or intra-constraints through machine learning

Learning constraints of variability models

Whatever the selected modeling strategy, specifying how contextual factors affect the configuration space of a software system is a complex and labour-intensive activity, highly dependent on expert knowledge.
Specific *constraints* are usually specified for relating features of the context to features of the software. Figure 1 presents possible examples of such *inter-constraints* in the lower part.
Due to the huge number of configurations and complex relations between context and software, it is easy to forget or to wrongly specify a constraint, with possible adverse consequences when the software is deployed.
This article aims to address the problem of "*How to elaborate a variability model (including constraints) that can be configured for different contexts?"* with a novel and radical approach, based on machine learning and constraint solving.

Our key idea is to execute and observe a sample of software configurations within a sample of context configurations, and then learn what features of the context are likely to discard or activate some features of the software. As a result, software developers and product managers can extract the *inter-constraints* that relate context to software.
They can also learn *intra-constraints* within software features (or within context features).
In a sense, machine learning has the potential to replace a manual, error-prone, time-consuming, and subjective modeling effort with an automated process based on realistic measurements of software configurations in numerous contexts.

The case of configurable vision systems

The engineering of computer vision systems (e.g., for tracking objects of interest in a video, see Figure 1) is representative of the general problem previously exposed.
Video experts have to compose numerous highly-configurable algorithms given their specific goals, execution environments, and the assumptions they make about videos to analyse.
Overall, a video processing chain can operate in a variety of contexts. It can process outdoors or indoors videos with more or less noise, filmed during day or night; the software could be embedded in resource-constrained devices; the quality of service needed by the application also differs: sometimes fast execution is the major concern, sometimes strong guarantees in terms of precision and/or recall of objects of interests tracked in the video dominate.

Our experience shows that modeling context and software variability (and their mutual relation) is labour-intensive, subjective and error-prone, even with extensive knowledge [3].
With our method, we aim to synthesize the constraints of Figure 1 as well as identifying new ones.

Learning Contextual Variability Models

Figure 2 introduces the learning process. All steps can be automated except the user specification of what is an acceptable configuration (typically below or above a performance threshold value).

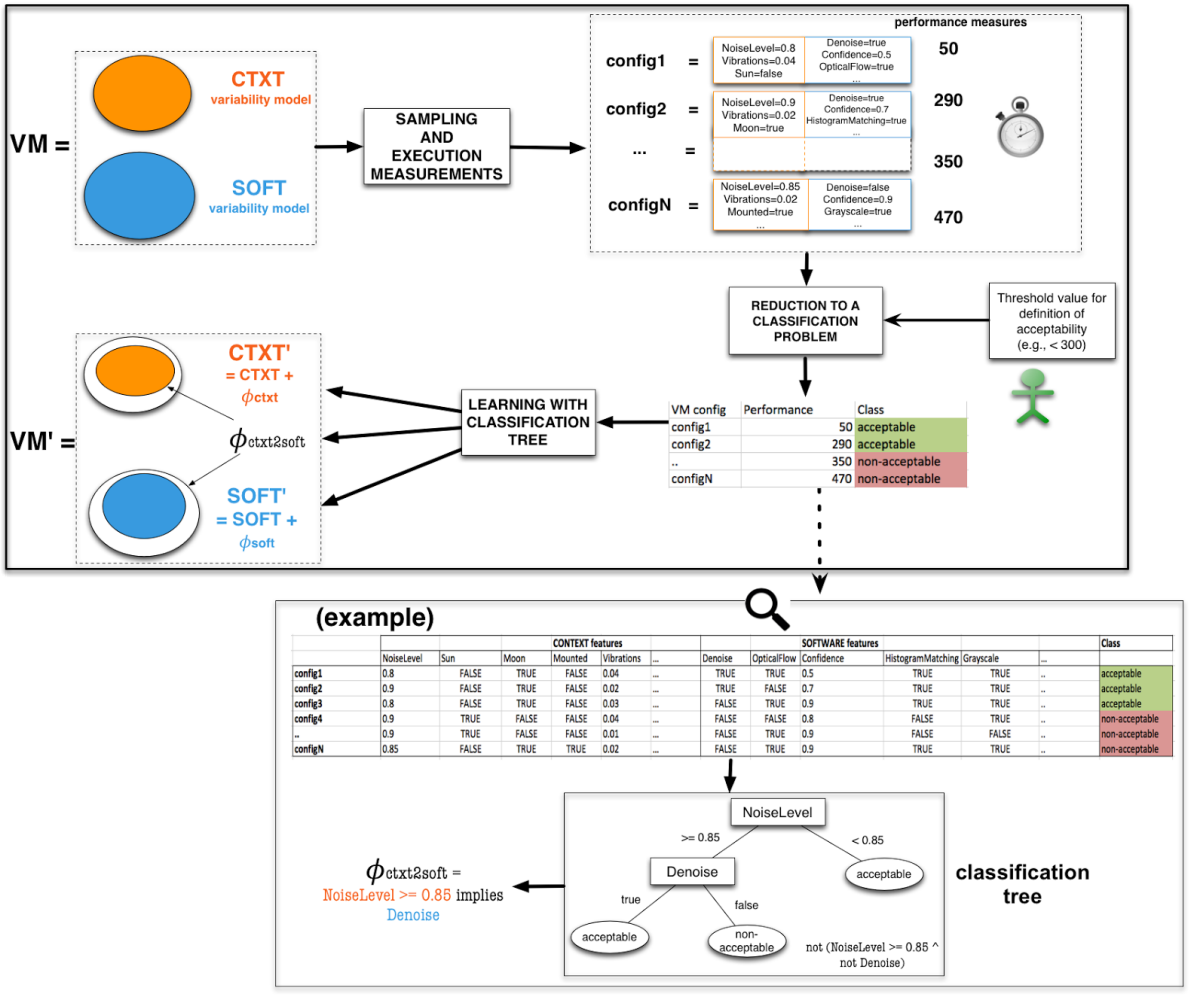

Figure 2: From a sample of configurations and measurements, we build a classification tree out of which we can extract constraints (an example of inter-constraint is given at the bottom).

First, we generate a sample of N configurations of the variability model *VM*. A configuration is composed of contextual features and software features. Numerous sampling strategies (e.g., random, t-wise) can be used to automatically select *valid* configurations [3, 4]. The number of configurations N in the sample can be controlled by the user.

Second, we execute and observe each software configuration within the context configuration. In our running example, a configuration of the tracking vision system is used to process a video with specific characteristics (see config1, config2, …, configN in Figure 2). Many properties of a software configuration can be measured: whether it crashes at runtime, whether it violates some invariants, or whether it meets a certain *performance*.

In the case of vision systems, we are mostly interested in measuring the accuracy and precision of the system to track objects, as well as the execution time.
We obtain a matrix with N configurations (see Figure 2).
Third, the user of our learning method specifies the threshold values for which the performance is considered as acceptable. Performance measurements are compared to the threshold value and we can classify configurations as acceptable or non-acceptable.
Finally, we address a *statistical binary classification* problem in which we can predict which context and software features (part of the configurations) lead to acceptable or non-acceptable configurations.

Statistical classification

The example at the bottom of Figure 2 illustrates how the classification works. We consider all features of configuration as predictor variables. We have a training sample of N configurations on *class* variables Y that take values {acceptable, non-acceptable}. Given training data (see the matrix at the bottom of Figure 2), we want to learn a classifier able to predict the values Y from new, previously unseen configuration values.

Many approaches have been developed for addressing (binary) classification problems. We selected *classification trees (CT),* a supervised machine learning technique that classifies data into classes [2]. CTs can handle Boolean and *numerical* values. Moreover, from CTs, we can extract human-readable rules or constraints expressed in propositional logics. Specifically, we first build the conjunction of all decisions that lead to leaves that classify configurations as “non-acceptable”; we then negate the resulting expression. Such constraints are eventually added in the variability model *VM’* to exclude non-acceptable configurations.

At the bottom of Figure 2, we give an example of a CT and an extracted constraint. Nodes test for the values of a certain feature (e.g., NoiseLevel). Edges correspond to the outcome of a test and connect to the next node (e.g., >= 0.85). Leaves of the tree predict the final outcome (“acceptable” or “non-acceptable”). We follow the paths leading to “non-acceptable”, build the conjunction of all decisions *NoiseLevel >= 0.85 and Denoise == false*, and negate the expression.

Initial results

Feasibility study. We have developed a *tracking* vision system by strictly following the approach illustrated throughout this article. Specifically, we modeled and implemented software variability in C++ using a subset of OpenCV.
To provide data appropriate to each context configuration, we synthesized videos with various properties like the noise level, the camera vibration, etc. It is also possible to use realistic benchmarks (e.g., COCO that provides thousands of annotated images with different kinds of objects and visual properties, see http://mscoco.org/home/).

We used a Lua video generator that synthesizes video variants together with their expected results (ground truths) [3]. In this way, we can measure the performance (precision and accuracy of the tracking of objects of interests, execution time, etc.) of tracking configurations in different contexts. Based on measurements, we have been able to specialize the configuration spaces and learn non-trivial constraints, similar to those in Figure 1.

System	Domain	Language	Variability (number of options)	Number of valid configurations	Performance objectives	Number of measurements needed to reach 80% accuracy (on average, for all perf. thresholds)	Corresponding % of measurements (wrt number of valid configs) needed to reach 80% accuracy
MOTIV industrial video generator	Video Processing	Lua	20 Boolean options and 88 numerical options	~10^100	noise, size	500	10^-98
Apache Web server	Web server	C	9 Boolean options	192	response rate	14	7.29
BerkeleyC	Database	C	18 Boolean options	2560	I/O time	26	1.02
BerkeleyJ	Database	Java	26 Boolean options	400	I/O time	9	2.25
LLVM	Compiler	C++	11 Boolean options	1024	optimization time	31	3.03
SQLite	Database	SQLite	39 Boolean options	10^6	time	901	0.1
Dune	Solver	C++	8 Boolean and 3 numerical options	2304	solving time	24	1.04
HIPAcc	Image Proc.	C++	31 Boolean and 2 numerical options	13485	solving time	135	1
HSMGP	Solver	N/A	11 Boolean and 3 numerical options	3456	time	35	1.01
JavaGC	Runtime Env.	C++	12 Boolean and 23 numerical options	10^31	time	1670	10^-28
x264	Video Processing	C	8 Boolean and 12 numerical options	10^27	energy, speed, size, time, watt	691	10^-25

Table 1: Experimental results across application domains

Case Study. We have specialized an industrial video generator written in Lua [3]. Our learning approach allows one to constrain the generator and only build video variants of a certain quality (size, noise frequency, etc.). Therefore we can get different variants of the video generator ready for numerous usages and contexts.
A sample of 500 configurations was used and we obtained a classification accuracy of 80% (see Table 1). Constraints have proven to be readable: domain experts were able to validate and incorporate them into the variability model [3].

Controlled experiments. We have considered 10 publicly available configurable systems (see Table 1) for which we have reused performance measurements (execution time, footprint, etc.) of configurations using benchmarks [4, 7]. We used our learning method to synthesize constraints in such a way we only retain software configurations meeting a certain performance objective. A practical application is that we can *specialize* configurable systems for targeting specific usages, customers, deployment scenarios, hardware settings, in short any contexts. For example, we can specialize the variability model of the x264 video encoder for achieving a “low energy consumption” (e.g., for embedding x264 in resource-constrained devices).
Empirical results are promising for two reasons. First, the learning phase can reach an accuracy greater than 80% on average for all performance thresholds and for all systems. We can thus narrow the space of possible configurations to a good approximation. Second, only a relative small training set is needed to achieve a high classification accuracy (see Table 1).

Overall, a reasonable number of measurements (from dozens to hundreds) can be sufficient to discover complex relationships that rightly discard a large portion of non-acceptable configurations, without over-constraining the configurable systems.

For more details, we invite the reader to consult online the source code and experimental data: http://learningconstraints.github.io as well as references [3] and [4].

Future Work

We are aware that there are classes of software systems and contexts for which the approach may be harder to engineer. A first difficulty is related to the development of procedures (oracles) for measuring software configurations in different contexts. It may be difficult to find the right data or to create the realistic contextual conditions. A second challenge is related to the complexity of the configuration space: e.g., how to scale for a large configurable system like the Linux kernel with 10000+ configuration options?
Despite some barriers, the idea of *learning* variability spaces is already applicable to many configurable systems. It could also be applied to dynamic software product lines [5] or self-adaptive systems capable of reconfiguring their behavior for addressing a variety of contexts.
We foresee that with the ever growing computation power and amount of data available, engineers will leverage machine learning to help elaborate contextual variability models.

References

[1] Herman Hartmann, Tim Trew: “Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains” SPLC’08

[2] W.-Y. Loh, “Classification and regression trees,” Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, vol. 1, no. 1, 2011.

[3] Paul Temple, José Angel Galindo Duarte, Mathieu Acher, and Jean-Marc Jézéquel. “Using Machine Learning to Infer Constraints for Product Lines” SPLC'16

[4] Paul Temple, Mathieu Acher, Jean-Marc A Jézéquel, Léo A Noel-Baron, and José A Galindo. “Learning-Based Performance Specialization of Configurable Systems” Research report, 2017 https://hal.archives-ouvertes.fr/hal-01467299

[5] Rafael Capilla, Jan Bosch “The Promise and Challenge of Runtime Variability” IEEE Computer 44(12): 93-95 (2011)

[6] Rafael Capilla, Oscar Ortiz, Mike Hinchey “Context Variability for Context-Aware Systems” IEEE Computer 47(2): 85-87 (2014)

[7] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, Andrzej Wasowski: “Variability-aware performance prediction: A statistical learning approach” ASE 2013

Short Bios

Paul Temple a PhD student in the DiverSE team at the University of Rennes 1 since September 2015.
He received a Master's degree in Computer Science from the University of Rennes 1 in 2013 and went in the e-payments and biometrics team (part of ENSICaen in Caen, FRANCE) during one year starting July 2014 as an engineer.
He is interested in Data and Image Processing, Software Engineering and Machine Learning.
He is now working on how to leverage machine learning techniques to ease the selection of a product derived from configurable systems w.r.t. users' requirements.

Dr. Mathieu Acher (http://mathieuacher.com) is an Associate Professor at University of Rennes 1, France. His research focuses on reverse engineering, modeling, reasoning, and understanding variability in various kinds of artefacts and domains. He is the main developer of FAMILIAR (http://familiar-project.github.io) for which he has designed and implemented novel automated operations. He has authored more than 80 peer-reviewed papers in international journals, conferences or workshops (e.g., ESE, STTT, SCP, ASE, ESEC/FSE, ISSTA, SPLC, CAiSE, FASE, IJCAI).
He is PC co-chair of Software Product Line Conference in 2017.

Dr. Jean-Marc Jézéquel is a Professor at the University of Rennes and Director of IRISA, one of the largest public research lab in Informatics in France.
He is also head of research of the French Cyber-defense Excellence Cluster and the director of the Rennes Node of EIT Digital. In 2016 he received the Silver Medal from CNRS.
His interests include model driven software engineering for software product lines, and specifically component based, dynamically adaptable systems with quality of service constraints, including security, reliability, performance, timeliness etc. He is the author of 4 books and of more than 250 publications in international journals and conferences. He was a member of the steering committees of the AOSD and MODELS conference series. He also served on the editorial boards of IEEE Computer, IEEE Transactions on Software Engineering, the Journal on Software and Systems, on the Journal on Software and System Modeling and the Journal of Object Technology. He received an engineering degree from Telecom Bretagne in 1986, and a Ph.D. degree in Computer Science from the University of Rennes, France, in 1989.

Dr. Olivier Barais (http://olivier.barais.fr) is a Full Professor at the University of Rennes 1, member of the DiverSE INRIA research team. He passed a PhD in computer science from the University of Lille 1, France in 2005. His research interests include Component Based Software Design, Model-Driven Engineering and Aspect Oriented Modeling. Olivier Barais has co-authored articles in conferences and journals such as SoSyM, IEEE Computer, ICSE, ASE, MoDELS, SPLC and CBSE.